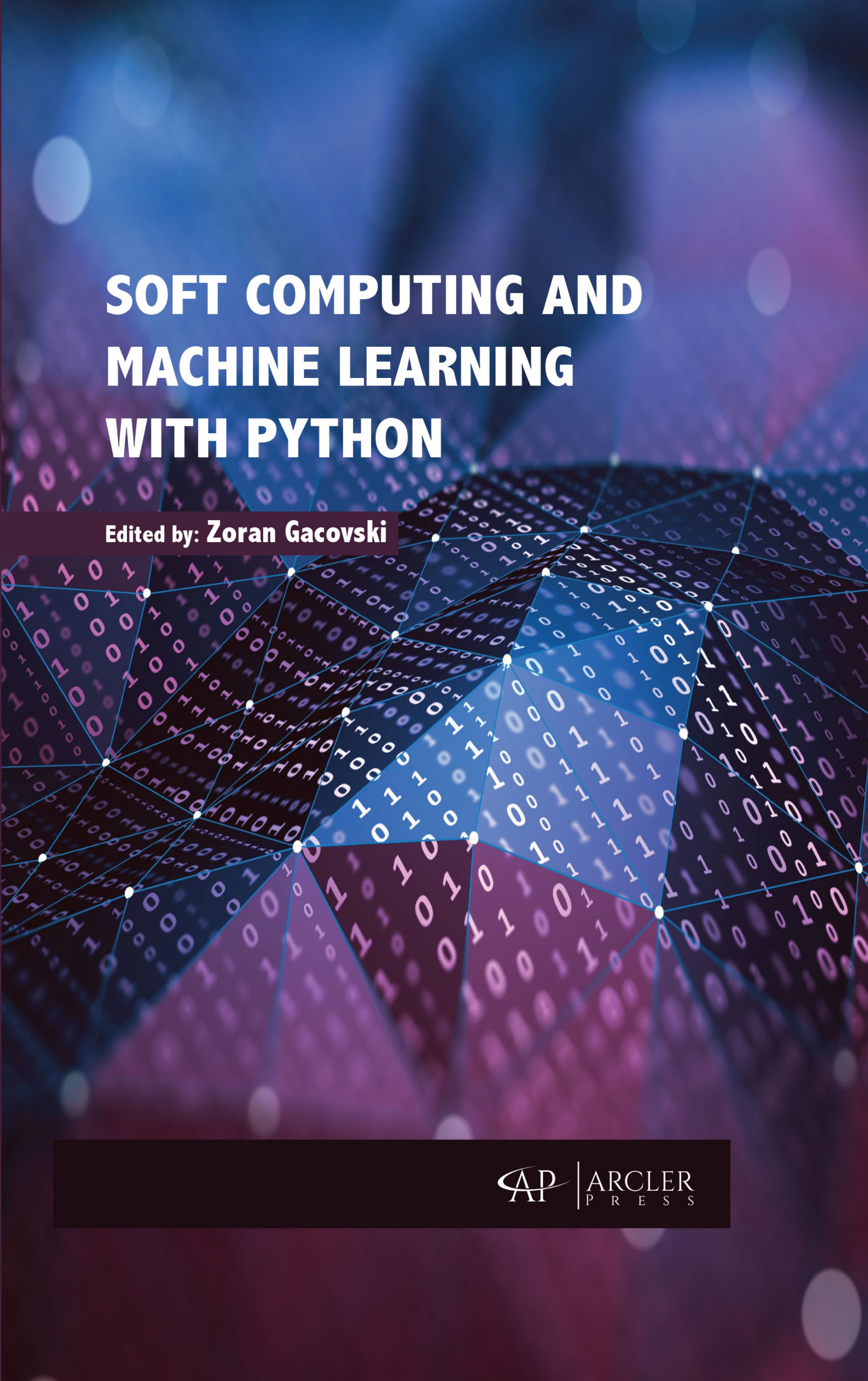


SOFT COMPUTING AND MACHINE LEARNING WITH PYTHON

Edited by: Zoran Gacovski

ARCLER PRESS

SOFT COMPUTING AND MACHINE LEARNING WITH PYTHON

SOFT COMPUTING AND MACHINE LEARNING WITH PYTHON

Edited by:

Zoran Gacovski

ARCLER
PRESS

www.arclerpress.com

Soft Computing and Machine Learning with Python

Zoran Gacovski

Arcler Press
2010 Winston Park Drive,
2nd Floor
Oakville, ON L6H 5R7
Canada
www.arclerpress.com
Tel: 001-289-291-7705
001-905-616-2116
Fax: 001-289-291-7601
Email: orders@arclereducation.com

e-book Edition 2019

ISBN: 978-1-77361-623-0 (e-book)

ISBN: 978-1-77361-500-4 (Hardcover)

DECLARATION

ABOUT THE EDITOR

Dr. Zoran Gacovski has earned his PhD degree at Faculty of Electrical engineering, Skopje. His research interests include Intelligent systems and Software engineering, fuzzy systems, graphical models (Petri, Neural and Bayesian networks), and IT security. He has published over 50 journal and conference papers, and he has been reviewer of renowned Journals.

In his career he was awarded by Fulbright postdoctoral fellowship (2002) for research stay at Rutgers University, USA. He has also earned best-paper award at the Baltic Olympiad for Automation control (2002), US NSF grant for conducting a specific research in the field of human-computer interaction at Rutgers University, USA (2003), and DAAD grant for research stay at University of Bremen, Germany (2008).

Currently, he is a professor in Computer Engineering at European University, Skopje, Macedonia.

TABLE OF CONTENTS

LIST OF CONTRIBUTORS

Taiwo Oladipupo Ayodele
University of Portsmouth United Kingdom

Manoel Fernando Alonso Gadi
Milton Keynes United Kingdom

Alair Pereira do Lago
Depart. de Ciencia de Computacao, Inst. de Matematica e Estatistica Universidade de Sao Paulo Brazil

Jorn Mehnen
Decision Engineering Centre, Cranfield University Cranfield, Bedfordshire MK43 0AL United Kingdom

Meherwar Fatima
Institute of CS & IT, The Women University Multan, Multan, Pakistan

Maruf Pasha
Department of Information Technology, Bahauddin Zakariya University, Multan, Pakistan

Nanxi Wang
Shanghai Starriver Bilingual School, Shanghai, China

Makhamisa Senekane
Faculty of Computing, Botho University-Maseru Campus, Maseru, Lesotho

Benedict Molibeli Taele
Department of Physics and Electronics, National University of Lesotho, Roma, Lesotho

Hudson F. Golino
Núcleo de Pós-Graduação, Pesquisa e Extensão, Faculdade Independente do Nordeste, Vitória da Conquista, Brazil

Cristiano Mauro Assis Gomes
Department of Psychology, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

Diego Andrade
Department of Psychology, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

John W. Shipman

Tom De Smedt
CLiPS Computational Linguistics Group University of Antwerp 2000 Antwerp, Belgium

Walter Daelemans
CLiPS Computational Linguistics Group University of Antwerp 2000 Antwerp, Belgium

Andreas C. M¨uller
Institute of Computer Science, Department VI University of Bonn Bonn, Germany

Sven Behnke
Institute of Computer Science, Department VI University of Bonn Bonn, Germany

Abhik Shah
Department of Chemical Engineering 3320 G.G. Brown Ann Arbor, MI 48103, USA

Peter Woolf
Department of Chemical Engineering 3320 G.G. Brown Ann Arbor, MI 48103, USA

Fabian Pedregosa
Parietal, INRIA Saclay Neurospin, Bat 145, CEA Saclay ˆ 91191 Gif sur Yvette – France

Gael Varoquaux
Parietal, INRIA Saclay Neurospin, Bat 145, CEA Saclay ˆ 91191 Gif sur Yvette – France

Alexandre Gramfort
Parietal, INRIA Saclay Neurospin, Bat 145, CEA Saclay ˆ 91191 Gif sur Yvette – France

Vincent Michel
Parietal, INRIA Saclay Neurospin, Bat 145, CEA Saclay ˆ 91191 Gif sur Yvette – France

Bertrand Thirion
Parietal, INRIA Saclay Neurospin, Bat 145, CEA Saclay ˆ 91191 Gif sur Yvette – France

Olivier Grisel
Nuxeo 20 rue Soleillet 75 020 Paris – France

Mathieu Blondel
Kobe University 1-1 Rokkodai, Nada Kobe 657-8501 – Japan

Peter Prettenhofer
Bauhaus-Universitat Weimar ¨ Bauhausstr. 11 99421 Weimar – Germany

Ron Weiss
Google Inc 76 Ninth Avenue New York, NY 10011 – USA

Vincent Dubourg
Clermont Universite, IFMA, EA 3867, LaMI ´ BP 10448, 63000 Clermont-Ferrand – France

Jake Vanderplas
Astronomy Department University of Washington, Box 351580 Seattle, WA 98195 – USA

Alexandre Passos
IESL Lab UMass Amherst Amherst MA 01002 – USA

David Cournapeau
Enthought 21 J.J. Thompson Avenue Cambridge, CB3 0FA – UK

Matthieu Brucher
Total SA, CSTJF avenue Larribau 64000 Pau – France

Matthieu Perrot
LNAO Neurospin, Bat 145, CEA Saclay ˆ 91191 Gif sur Yvette – France

Edouard Duchesnay
LNAO Neurospin, Bat 145, CEA Saclay ˆ 91191 Gif sur Yvette – France

Javier Escalada
Computer Science Department, University of Oviedo, Calvo Sotelo s/n, 33007 Oviedo, Spain

Francisco Ortin
Computer Science Department, University of Oviedo, Calvo Sotelo s/n, 33007 Oviedo, Spain

Ted Scully
Cork Institute of Technology, Computer Science Department, Rossa Avenue, Bishopstown, Cork, Ireland

Robert M. Nowak
Institute of Electronic Systems, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland

Debra Knisley
Institute for Quantitative Biology, East Tennessee State University, Johnson City, TN 37614-0663, USA
Department of Mathematics and Statistics, East Tennessee State University, Johnson City, TN 37614-0663, USA

Jeff Knisley
Institute for Quantitative Biology, East Tennessee State University, Johnson City, TN 37614-0663, USA
Department of Mathematics and Statistics, East Tennessee State University, Johnson City, TN 37614-0663, USA

Chelsea Ross
Department of Mathematics and Statistics, East Tennessee State University, Johnson City, TN 37614-0663, USA

Alissa Rockney
Department of Mathematics and Statistics, East Tennessee State University, Johnson City, TN 37614-0663, USA

LIST OF ABBREVIATIONS

AST	Abstract Syntax Tree
AIS	Artificial Immune Systems
ANN	Artificial Neural Network
BN	Bayesian Networks
CLES	Common language effect size
CAD	Computer Aided Diagnosis
CRF	Conditional Random Fields
DT	Decision Trees
DTG	Digital Technology Group
DBN	Dynamic Bayesian Networks
ERDF	European Regional Development Funds
FEP	Function entry points
GA	Genetic Algorithm
GIL	Global Interpreter Lock
GUI	Graphical user interface
GPU	Graphics processing unit
ILP	Inductive logic procedures
TDRI	Inductive Reasoning Developmental Test
JSON	JavaScript Object Notation
KS	Kolmogorov Smirnov
MIS	Management Information Systems
MAE	Mean absolute error
MSE	Mean squared error
TCM	Metacognitive Control Test
MFNN	Multilayer feed-forward neural network
MLP	Multilayer Perceptrons

NLP	Natural language processing
NN	Neural Networks
OOT	Out-Of-Time sample
PEBL	Python Environment for Bayesian Learning
QIP	Quantum Information Processing
QML	Quantum machine learning
ROC	Receiver Operating Characteristic
RMSE	Root mean squared error
SVC	Support vector classify
SVM	Support Vector Machine

PREFACE

A definition states that the machine learning is a discipline that allows the computers to learn without explicit programming. The challenge in machine learning is how to accurately (algorithmic) describe some kinds of tasks that people can easily solve (for example face recognition, speech recognition etc.).

Such algorithms can be defined for certain types of tasks, but they are very complex and / or require large knowledge base (e.g. machine translation-MT).

In many of the areas - data are continuously collected in order to get "some knowledge out of them"; for example - in medicine (patient data and therapy), in marketing (the users / customers and what they buy, what are they interested in, how products are rated etc.).

Data analysis of this scale requires approaches that will allow you to discover patterns and dependences among the data, that are neither known, nor obvious, but can be useful (data mining).

- Information retrieval - IR, is finding existing information as quickly as possible. For example, web browser - finds page within the (large) set of the entire WWW.
- Machine Learning - ML, is a set of techniques that generalize existing knowledge of the new information, as precisely as possible. An example is the speech recognition.
- Data mining - DM, primarily relates to the disclosure of something hidden within the data, some new dependence, which have not previously been known. Example is CRM - the customer analysis.

Python is high-level programming language that is very suitable for web development, programming of games, and data manipulation / machine learning applications. It is object-oriented language and interpreter as well, allowing the source code to execute directly (without compiling).

This edition covers machine learning theory and applications with Python, and includes chapters for soft computing theory, machine learning techniques/ applications, Python language details, and machine learning examples with Python.

Section 1 focuses on soft computing theory, describing machine learning overview, types of machine learning algorithms, and data mining with skewed data.

Section 2 focuses on machine learning techniques and applications, describing machine learning algorithms for disease diagnostic, bankruptcy prediction using machine learning, prediction of solar irradiation using quantum support vector machine and predicting academic achievement of high-school students.

Section 3 focuses on Python language details, describing Python 2.7 programming tutorial, pattern for Python, Pystruct - learning structured prediction in Python.

Section 4 focuses on machine learning with Python use cases, describing Python environment for Bayesian learning: inferring the structure of Bayesian Networks from knowledge and data, Scikit-learn: Machine Learning in Python, Efficient Platform for the Automatic Extraction of Patterns in native code, polyglot programming in applications used for genetic data analysis, and classifying multigraph models of secondary RNA structure using graph-theoretic descriptors.

SECTION I
SOFT COMPUTING THEORY

CHAPTER 1

MACHINE LEARNING OVERVIEW

Taiwo Oladipupo Ayodele

University of Portsmouth United Kingdom

MACHINE LEARNING OVERVIEW

Machine Learning according to Michie et al (D. Michie, 1994) is generally taken to encompass automatic computing procedures based on logical or binary operations that learn a task from a series of examples. Here we are just concerned with classification, and it is arguable what should come under the Machine Learning umbrella. Attention has focussed on decision-tree approaches, in which classification results from a sequence of logical steps. These are capable of representing the most complex problem given sufficient data (but this may mean an enormous amount!). Other techniques, such as genetic algorithms and inductive logic procedures (ILP), are currently under active development and in principle would allow us to deal with more general types of data, including cases where the number and type of attributes may vary, and where additional layers of learning are

Citation: Taiwo Oladipupo Ayodele (February 1st 2010). "Machine Learning Overview", New Advances in Machine Learning Yagang Zhang, IntechOpen, DOI: 10.5772/9374.

superimposed, with hierarchical structure of attributes and classes and so on. Machine Learning aims to generate classifying expressions simple enough to be understood easily by the human. They must mimic human reasoning sufficiently to provide insight into the decision process. Like statistical approaches, background knowledge may be exploited in development, but operation is assumed without human intervention.

To learn is:

- to gain knowledge, comprehension, or mastery of through experience or study or to gain knowledge (of something) or acquire skill in (some art or practice)
- to acquire experience of or an ability or a skill in
- to memorize (something), to gain by experience, example, or practice.

Machine Learning can be defines as a process of building computer systems that automatically improve with experience, and implement a learning process. Machine Learning can still be defined as learning the theory automatically from the data, through a process of inference, model fitting, or learning from examples:

- Automated extraction of useful information from a body of data by building good probabilistic models.
- Ideally suited for areas with lots of data in the absence of a general theory.

A major focus of machine learning research is to automatically produce models and a model is a pattern, plan, representation, or description designed to show the main working of a system, or concept, such as rules determinate rule for performing a mathematical operation and obtaining a certain result, a function from sets of formulae to formulae, and patterns (model which can be used to generate things or parts of a thing from data.

Learning is a MANY-FACETED PHENOMENON as described by Jaime et al (Jaime G. Carbonell, 1983) and also stated that Learning processes include the acquisition of new declarative knowledge, the development of motor and cognitive skills through instruction or practice, the organization of new knowledge into general, effective representations, and the discovery of new facts and theories through observation and experimentation. The study and computer modelling of learning processes in their multiple manifestations constitutes the subject matter of machine learning. Although machine learning has been a central concern in artificial intelligence since the early days when the idea of "self-organizing systems" was popular, the limitations inherent in the early neural network approaches led to a temporary decline in research volume. More recently, new symbolic methods and knowledge-intensive techniques have yielded promising results and these in turn have led to the current, revival in machine learning research. This book examines some basic methodological issues, existing techniques, proposes a classification of machine learning techniques, and provides a historical review of the major research directions.

Machine Learning according to Michie et al (D. Michie, 1994) is generally taken to encompass automatic computing procedures based on logical or binary operations that learn a task from a series of examples. Here we are just concerned with classification, and it is arguable what should come under the Machine Learning umbrella. Attention has focussed on decision-tree approaches, in which classification results from a sequence of logical steps. These are capable of representing the most complex problem given sufficient data (but this may mean an enormous amount!). Other techniques, such as genetic algorithms and inductive logic procedures (ILP), are currently under active development and in principle would allow us to deal with more general types of data, including cases where the number and type of attributes may vary, and where additional layers of learning are superimposed, with hierarchical structure of attributes and classes and so on. Machine Learning aims to generate classifying expressions simple enough to be understood easily by the human. They must mimic human reasoning sufficiently to provide insight into the decision process. Like statistical approaches, background knowledge may be exploited in development, but operation is assumed without human intervention. Machine learning has always been an integral part of artificial intelligence according to Jaime et al (Jaime G. Carbonell, 1983), and its methodology has evolved in concert, with the major concerns of the field. In response to the difficulties of encoding ever increasing volumes of knowledge in model AI systems, many researchers have recently turned their attention to machine learning as a means to overcome the knowledge acquisition bottleneck. This book presents a taxonomic analysis of machine learning organized primarily by learning strategies and secondarily by knowledge representation and application areas. A historical survey out lining the development of various approaches to machine learning is presented from early neural networks to present knowledge-intensive techniques.

The Aim of Machine Learning

The field of machine learning can be organized around three primary research Areas:

- ***Task-Oriented Studies***: The development and analysis of learning systems oriented toward solving a predetermined set, of tasks (also known as the "engineering approach").
- ***Cognitive Simulation***: The investigation and computer simulation of human learning processes (also known as the "cognitive modelling approach")
- ***Theoretical Analysis***: the theoretical exploration of the space of possible learning methods and algorithms independent application domain.

Although many research efforts strive primarily towards one of these objectives, progress in on objective often lends to progress in another. For example, in order to investigate the space of possible learning methods, a reasonable starting point may be to consider the only known example of robust learning behaviour, namely humans (and perhaps other biological systems) Similarly, psychological investigations of human learning may held by theoretical analysis that may suggest various possible learning models. The need to acquire a particular form of knowledge in stone task-oriented study may itself spawn new theoretical analysis or pose the question: "how do humans acquire this specific skill (or knowledge)?" The existence of these mutually supportive objectives reflects the entire field of artificial intelligence where expert system research, cognitive simulation, and theoretical studies provide some (cross-fertilization of problems and ideas (Jaime G. Carbonell, 1983).

Applied Learning Systems

At, present, instructing a computer or a computer-controlled robot, to perform a task requires one to define a complete and correct, algorithm for that task, and then laboriously program the algorithm into a computer. These activities typically involve a tedious and time-consuming effort by specially trained personnel. Present-day computer systems cannot truly learn to perform a task through examples or by analogy to a similar, previous-solved task. Nor can they improve significantly on the basis of past, mistakes or acquire new abilities by observing and imitating experts. Machine learning research strives to open the possibility of instructing computers in such new ways, and thereby promises to ease the burden of hand-programming growing volumes of increasingly complex information into the computers of tomorrow. The rapid expansion of application and availability of computers today makes this possibility even more attractive and desirable.

Knowledge Acquisition

When approaching a task-oriented knowledge acquisition task, one must be aware that, the resultant computer system must interact with humans, and therefore should closely parallel human abilities. The traditional argument that an engineering approach need not reflect human or biological performance and is not, truly applicable to machine learning. Since airplane, a successful result on an almost pure engineering approach, better little resemblance to their biological counterparts, one may argue that applied knowledge acquisition systems could be equally divorced from any consideration of human capabilities. This argument does not apply here because airplanes need not interact, with or understand birds Learning machines, on the other hand, will have to interact, with the people who make use of them, and consequently the concept and skills they acquire- if not necessarily their internal mechanism and must be understandable to human.

Machine Learning as a Science

The clear contender for a cognitive invariant in human is the learning mechanism which is the ability facts, skills and more abstractive concepts. Therefore understanding human learning well enough to reproduce aspect of that learning behaviour in a computer system is, in itself, a worthy scientific goal. Moreover, the computer can render substantial assistance to cognitive psychology, in that it may be used to test the consistency and completeness of learning theories and enforce a commitment to the fine-structure processlevel detail that precludes meaningless tautological or untestable theories (Bishop, 2006).

The study of human learning processes is also of considerable practical significance. Gaining insights into the principles underlying human learning abilities is likely to lead to more effective educational techniques. Machine learning research is all about developing intelligent computer assistant or a computer tutoring systems and many of these goals are shared within the machine learning fields. According to Jaime et al (Jaime G. Carbonell, 1983) who stated computer tutoring are starting to incorporate abilities to infer models of student competence from observed performance. Inferring the scope of a student's knowledge and skills in a particular area allows much more effective and individualized tutoring of the student (Sleeman, 1983).

The basic scientific objective of machine learning is the exploration of possible learning mechanisms, including the discovery of different induction algorithms, the scope of theoretical limitations of certain method seems to be the information that must be available to the learner, the issue of coping with imperfect training data and the creation of general techniques applicable in many task domains. There is not reason to believe that human learning methods are the only possible mean of acquiring knowledge and skills. In fact, common sense suggests that human learning represents just one point in an uncharted space of possible learning methods- a point that through the evolutionary process is particularly well suited to cope with the general physical environment in which we exist. Most theoretical work in machine learning are centred on creation, characterization and analysis of general learning methods, with the major emphasis on analyzing generality and performance rather than psychological plausibility.

Whereas theoretical analysis provides a means of exploring the space of possible learning methods, the task-oriented approach provides a vehicle to test and improve the performance of functional learning systems and testing applied learning systems, one can determine the cost-effectiveness trade-offs and limitations of particular approaches to learning. In this way, individual data points in the space possible learning systems are explored and the space itself becomes better understood.

Knowledge Acquisition and Skill Refinement: There are two basic form of learning:

1) Knowledge Acquisition
2) Skill refinement

When it is said that someone learned mathematics, it means that this person acquired concepts of mathematics, understood the meaning and their relationship to each other as well as to the world. The importance of learning in this case is acquisition of knowledge, including the description and models of physical systems and their behaviours, incorporating a variety of representations from simple intrusive mental model models, examples and images to completely test mathematical equations and physical laws. A person is said to have learned more if this knowledge explains a broader scope of situations, is more accurate, and is better able to predict the behaviour of the typical world (Allix, 2003). This form of learning is typically to a large variety of situations and is generally learned knowledge acquisition.

Therefore, knowledge acquisition is defined as learning a new task coupled with the ability to apply the information in the effective manner.

The second form of learning is the gradual improvement of motor and cognitive skills through practice- Learning by practice. Learning such as:

- Learning to drive a car
- Learning to play keyboard
- Learning to ride a bicycle
- Learning to play piano

If one acquire all textbook knowledge on how to perform these aforementioned activities, this represent the initial phase in developing the required skills. So, the major part of the learning process consists of taming the acquired skill, and improving the mental or motor coordination or learning coordination by repeated practice and correction of deviations from desired behaviour. This form of learning often called skill taming. This differs in many ways from knowledge acquisition. Where knowledge acquisition may be a conscious process whose result is the creation of new representative knowledge structures and mental models, and skill taming is learning from example or learning from repeated practice without concerted conscious effort. Jamie (Jaime G. Carbonell, 1983) explained that most human learning appears to be a mixture of both activities, with intellectual endeavours favouring the former and motor coordination tasks favouring the latter. Present machine learning research focuses on the knowledge acquisition aspect, although some investigations, specifically those concerned with learning in problem-solving and transforming declarative instructions into effective actions, touch on aspects of both types of learning. Whereas knowledge acquisition clearly belongs in the realm of artificial intelligence research, a case could be made that skill refinement comes closer to non-symbolic processes such as those studied in adaptative control system. Hence, perhaps both forms of learning- (knowledge based and refinement learning) can be captured in artificial intelligence models.

Classification of Machine Learning

There are several areas of machine learning that could be exploited to solve the problems of email management and our approach implemented unsupervised machine learning method. Uunsupervised learning is a method of machine learning whereby the algorithm is presented with examples from the input space only and a model is fit to these observations. For example, a clustering algorithm would be a form of unsupervised learning. "Unsupervised learning is a method of machine learning where a model is fit to observations. It is distinguished from supervised learning by the fact that there is no a priori output. In unsupervised learning, a data set of input objects is gathered. Unsupervised learning then typically treats input objects as a set of random variables. A joint density model is then built for the data set. The problem of unsupervised learning involved learning patterns in the input when no specific output values are supplied" according to Russell (Russell, 2003).

In the unsupervised learning problem, we observe only the features and have no measurements of the outcome. Our task is rather to describe how the data are organized or clustered". Hastie (Trevor Hastie, 2001) explained that "In unsupervised learning or clustering there is no explicit teacher, and the system forms clusters or "natural groupings" of the input patterns. "Natural" is always defined explicitly or implicitly in the clustering system itself; and given a particular set of patterns or cost function, different clustering algorithms lead to different clusters. Often the user will set the hypothesized number of different clusters ahead of time, but how should this be done? How do we avoid inappropriate representations?" according to Duda (Richard O. Duda, 2000).

There are various categories in the field of artificial intelligence. The classifications of machine learning systems are:

- ***Supervised Machine Learning***: Supervised learning is a machine learning technique for learning a function from training data. The training data consist of pairs of input objects (typically vectors), and desired outputs. The output of the function can be a continuous value (called regression), or can predict a class label of the input object (called classification). The task of the supervised learner is to predict the value of the function for any valid input object after having seen a number of training examples (i.e. pairs of input and target output). To achieve this, the learner has to generalize from the presented data to unseen situations in a "reasonable" way (see inductive bias). (Compare with unsupervised learning.)

Supervised learning is a machine learning technique whereby the algorithm is first presented with training data which consists of examples which include both the inputs and the desired outputs; thus enabling it to learn a function. The learner should then be able to generalize from the presented data to unseen examples." by Mitchell (Mitchell, 2006). Supervised learning also implies we are given a training set of (X, Y) pairs by a "teacher". We know (sometimes only approximately) the values of f for the m samples in the training set, ≡ we assume that if we can find a hypothesis, h, that closely agrees with f for the members of ≡ then this hypothesis will be a good guess for f especially if ≡ is large. Curvefitting is a simple example of supervised learning of a function. Suppose we are given the values of a two-dimensional function. f, at the four sample points shown by the solid circles in Figure 9. We want to fit these four points with a function, h, drawn from the set, H, of second-degree functions. We show there a two-dimensional parabolic surface above the $x_1 . x_2$, plane that fits the points. This parabolic function, h, is our hypothesis about the function f, which produced the four samples. In this case, h = f at the four samples, but we need not have required exact matches. Read more in section 3.1.

- Unsupervised Machine Learning: Unsupervised learning is a type of machine learning where manual labels of inputs are not used. It is distinguished from supervised learning approaches which learn how to perform a task, such as classification or regression, using a set of human prepared examples. .Unsupervised learning means we are only given the Xs and some (ultimate) feedback function on our performance. We simply have a training set of vectors without function values of them. The problem in this case, typically, is to partition the training set into subsets, $\equiv_1 \ldots \equiv_R$, in some appropriate way.

Classification of Machine Learning

Classification of machine learning system could be implemented along many different dimensions and we have chosen these two dimensions:

- Inference Learning: This is a form of classification on the basis of underlying strategy that is involved. These strategies will depend on the amount of inference the learning system performs on the information provided to the system.

Now learning strategies are distinguished by the amount of inference the learner performs on the information provided. So, if a computer system performs email classification for example, it knowledge increases but this may not perform any inference on the new information, this means all cognitive efforts is on the part of the analyst or programmer. But if the machine learning classifier independently discovers new theories or adopt new concepts, this will perform a very substantial inference. This is what is called deriving knowledge from example or experiments or by observation. An example is: When a student wants to solve statistical problems in a text book – this is a process that involves inference but the solution is not to discover a brand new formula without guidance from a teacher or text book. So, as the amount of inference that the learner is capable of performing increases, the burdens placed on the teacher or on external environ decreases. According to Jaime (Jaime G. Carbonell, 1983) , (Anil Mathur, 1999) who stated that it is much more difficult to teach a person by explaining each steps in a complex task than by showing that person the way that similar tasks are usually done. It more difficult yet to programme a computer to perform a complex task than to instruct a person to perform the task; as programming requires explicit specification of all prerequisite details, whereas a person receiving instruction can use prior knowledge and common sense to fill in most mundane details.

- *Knowledge Representation*: This is a form of skill acquire by the learner on the basis of the type of representation of the knowledge.

Existing Learning Systems

There are many other existing learning systems that employ multiple strategies and knowledge representations and some have been applied to more than one. In the knowledge based machine learning method, no inference is used but the learner display the transformation of knowledge in varieties of ways:

- *Learning by being programmed*: When an algorithm or code is written to perform specific task. E.g. a code is written as a guessing game for the type of animal. Such a programme could be modified by external entity.
- *Learning by memorisation*: This is by memorising given facts or data with no inference drawn from the incoming information or data.
- *Learning from examples*: This is a special case of inductive learning. Given a set of examples and counterexamples of a concept, the learner induces a general concept description that describes all of the positive examples and none of the counterexamples. Learning from examples is a method has been heavily investigated in artificial intelligence field. The amount of inference perform by the learner is much greater than in learning

from instructions, (Anil Mathur, 1999), (Jaime G. Carbonell, 1983).

- *Learning from Observation*: This is an unsupervised learning approach and is a very general form of inductive learning that includes discovery systems, theory formation tasks, the creation of classification criteria to form taxonomic hierarchies and similar task to be performed without benefit of an external teacher. Unsupervised learning requires the learner to perform more inference than any approach as previously explained. The learner is not provided with a set if data or instance of a particular concept. The above classification of learning strategies should help one to compare various learning systems in terms of their underlying mechanisms, in terms of the available external source of information and in terms of the degree to which they reply on preorganised knowledge. Read more in section 3.2.

Machine Learning Applications

The other aspect for classifying learning systems is the area of application which gives a new dimension for machine learning. Below are areas to which various existing learning systems have been applied. They are:

1) Computer Programming
2) Game playing (chess, poker, and so on)
3) Image recognition, Speech recognition
4) Medical diagnosis
5) Agriculture, Physics
6) Email management, Robotics
7) Music
8) Mathematics
9) Natural Language Processing and many more.

REFERENCES

1. Allix, N. M. (2003, April). Epistemology And Knowledge Management Concepts And Practices. Journal of Knowledge Management Practice .
2. Alpaydin, E. (2004). Introduction to Machine Learning. Massachusetts, USA: MIT Press.
3. Anderson, J. R. (1995). Learning and Memory. Wiley, New York, USA.
4. Anil Mathur, G. P. (1999). Socialization influences on preparation for later life. Journal of Marketing Practice: Applied Marketing Science , 5 (6,7,8), 163 - 176.
5. Ashby, W. R. (1960). Design of a Brain, The Origin of Adaptive Behaviour. John Wiley and Son.
6. Batista, G. &. (2003). An Analysis of Four Missing Data Treatment Methods for Suppervised Learning. Applied Artificial Intelligence , 17, 519-533.
7. Bishop, C. M. (1995). Neural Networks for Pattern Recognition. Oxford, England: Oxford University Press.
8. Bishop, C. M. (2006). Pattern Recognition and Machine Learning (Information Science and Statistics). New York, New York: Springer Science and Business Media.
9. Block H, D. (1961). The Perceptron: A Model of Brian Functioning. 34 (1), 123-135.
10. Carling, A. (1992). Introducing Neural Networks . Wilmslow, UK: Sigma Press.
11. D. Michie, D. J. (1994). Machine Learning, Neural and Statistical Classification. Prentice Hall Inc.
12. Fausett, L. (19994). Fundamentals of Neural Networks. New York: Prentice Hall.
13. Forsyth, R. S. (1990). The strange story of the Perceptron. Artificial Intelligence Review , 4 (2), 147-155.
14. Friedberg, R. M. (1958). A learning machine: Part, 1. IBM Journal , 2-13.
15. Ghahramani, Z. (2008). Unsupervised learning algorithms are designed to extract structure from data. 178, pp. 1-8. IOS Press.
16. Gillies, D. (1996). Artificial Intelligence and Scientific Method. OUP Oxford.
17. Haykin, S. (19994). Neural Networks: A Comprehensive Foundation.

New York: Macmillan Publishing.

18. Hodge, V. A. (2004). A Survey of Outlier Detection Methodologies. Artificial Intelligence Review, 22 (2), 85-126.
19. Holland, J. (1980). Adaptive Algorithms for Discovering and Using General Patterns in Growing Knowledge Bases Policy Analysis and Information Systems. 4 (3). Hunt, E. B. (1966). Experiment in Induction.
20. Ian H. Witten, E. F. (2005). Data Mining Practical Machine Learning and Techniques (Second edition ed.). Morgan Kaufmann.
21. Jaime G. Carbonell, R. S. (1983). Machine Learning: A Historical and Methodological Analysis. Association for the Advancement of Artificial Intelligence , 4 (3), 1-10.
22. Kohonen, T. (1997). Self-Organizating Maps.
23. Luis Gonz, l. A. (2005). Unified dual for bi-class SVM approaches. Pattern Recognition , 38 (10), 1772-1774.
24. McCulloch, W. S. (1943). A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophysics , 115-133.
25. Mitchell, T. M. (2006). The Discipline of Machine Learning. Machine Learning Department technical report CMU-ML-06-108, Carnegie Mellon University.
26. Mooney, R. J. (2000). Learning Language in Logic. In L. N. Science, Learning for Semantic Interpretation: Scaling Up without Dumbing Down (pp. 219-234). Springer Berlin / Heidelberg.
27. Mostow, D. (1983). Transforming declarative advice into effective procedures: a heuristic search cxamplc In I?. S. Michalski,. Tioga Press.
28. Nilsson, N. J. (1982). Principles of Artificial Intelligence (Symbolic Computation / Artificial Intelligence). Springer.
29. Oltean, M. (2005). Evolving Evolutionary Algorithms Using Linear Genetic Programming. 13 (3), 387 - 410 .
30. Orlitsky, A., Santhanam, N., Viswanathan, K., & Zhang, J. (2005). Convergence of profile based estimators. Proceedings of International Symposium on Information Theory. Proceedings. International Symposium on, pp. 1843 - 1847. Adelaide, Australia: IEEE.
31. Patterson, D. (19996). Artificial Neural Networks. Singapore: Prentice Hall. R. S. Michalski, T. J. (1983). Learning from Observation: Conceptual Clustering. TIOGA Publishing Co.

32. Rajesh P. N. Rao, B. A. (2002). Probabilistic Models of the Brain. MIT Press.
33. Rashevsky, N. (1948). Mathematical Biophysics:Physico-Mathematical Foundations of Biology. Chicago: Univ. of Chicago Press.
34. Richard O. Duda, P. E. (2000). Pattern Classification (2nd Edition ed.).
35. Richard S. Sutton, A. G. (1998). Reinforcement Learning. MIT Press.
36. Ripley, B. (1996). Pattern Recognition and Neural Networks. Cambridge University Press.
37. Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain . Psychological Review , 65 (6), 386-408.
38. Russell, S. J. (2003). Artificial Intelligence: A Modern Approach (2nd Edition ed.). Upper Saddle River, NJ, NJ, USA: Prentice Hall.
39. Ryszard S. Michalski, J. G. (1955). Machine Learning: An Artificial Intelligence Approach (Volume I). Morgan Kaufmann .
40. Ryszard S. Michalski, J. G. (1955). Machine Learning: An Artificial Intelligence Approach.
41. Selfridge, O. G. (1959). Pandemonium: a paradigm for learning. In The mechanisation of thought processes. H.M.S.O., London. London.
42. Sleeman, D. H. (1983). Inferring Student Models for Intelligent CAI. Machine Learning. Tioga Press.
43. Tapas Kanungo, D. M. (2002). A local search approximation algorithm for k-means clustering. Proceedings of the eighteenth annual symposium on Computational geometry (pp. 10-18). Barcelona, Spain : ACM Press.
44. Timothy Jason Shepard, P. J. (1998). Decision Fusion Using a Multi-Linear Classifier . In Proceedings of the International Conference on Multisource-Multisensor Information Fusion.
45. Tom, M. (1997). Machibe Learning. Machine Learning, Tom Mitchell, McGraw Hill, 1997: McGraw Hill.
46. Trevor Hastie, R. T. (2001). The Elements of Statistical Learning. New york, NY, USA: Springer Science and Business Media.
47. Widrow, B. W. (2007). Adaptive Inverse Control: A Signal Processing Approach. Wiley-IEEE Press.

48. Y. Chali, S. R. (2009). Complex Question Answering: Unsupervised Learning Approaches and Experiments. Journal of Artificial Intelligent Research , 1-47.

49. Yu, L. L. (2004, October). Efficient feature Selection via Analysis of Relevance and Redundacy. JMLR , 1205-1224.

50. Zhang, S. Z. (2002). Data Preparation for Data Mining. Applied Artificial Intelligence. 17, 375 - 381.

CHAPTER 2

TYPES OF MACHINE LEARNING ALGORITHMS

Taiwo Oladipupo Ayodele

University of Portsmouth United Kingdom

MACHINE LEARNING: ALGORITHMS TYPES

Machine learning algorithms are organized into taxonomy, based on the desired outcome of the algorithm. Common algorithm types include:

- Supervised learning --- where the algorithm generates a function that maps inputs to desired outputs. One standard formulation of the supervised learning task is the classification problem: the learner is required to learn (to approximate the behavior of) a function which maps a vector into one of several classes by looking at several input-output examples of the function.
- Unsupervised learning --- which models a set of inputs: labeled examples are not available.

Citation: Taiwo Oladipupo Ayodele (February 1st 2010). "Types of Machine Learning Algorithms", New Advances in Machine Learning Yagang Zhang, IntechOpen, DOI: 10.5772/9385.

- Semi-supervised learning --- which combines both labeled and unlabeled examples to generate an appropriate function or classifier.
- Reinforcement learning --- where the algorithm learns a policy of how to act given an observation of the world. Every action has some impact in the environment, and the environment provides feedback that guides the learning algorithm.
- Transduction --- similar to supervised learning, but does not explicitly construct a function: instead, tries to predict new outputs based on training inputs, training outputs, and new inputs.
- Learning to learn --- where the algorithm learns its own inductive bias based on previous experience.

The performance and computational analysis of machine learning algorithms is a branch of statistics known as computational learning theory.

Machine learning is about designing algorithms that allow a computer to learn. Learning is not necessarily involves consciousness but learning is a matter of finding statistical regularities or other patterns in the data. Thus, many machine learning algorithms will barely resemble how human might approach a learning task. However, learning algorithms can give insight into the relative difficulty of learning in different environments.

Supervised Learning Approach

Supervised learning is fairly common in classification problems because the goal is often to get the computer to learn a classification system that we have created. Digit recognition, once again, is a common example of classification learning. More generally, classification learning is appropriate for any problem where deducing a classification is useful and the classification is easy to determine. In some cases, it might not even be necessary to give predetermined classifications to every instance of a problem if the agent can work out the classifications for itself. This would be an example of unsupervised learning in a classification context.

Supervised learning often leaves the probability for inputs undefined. This model is not needed as long as the inputs are available, but if some of the input values are missing, it is not possible to infer anything about the outputs. Unsupervised learning, all the observations are assumed to be caused by latent variables, that is, the observations is assumed to be at the end of the causal chain. Examples of supervised learning and unsupervised learning are shown in the figure 1 below:

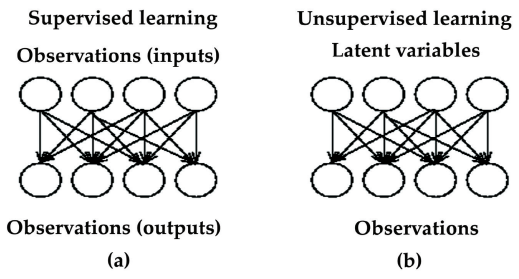


Figure 1: Examples of Supervised and Unsupervised Learning.

Supervised learning is the most common technique for training neural networks and decision trees. Both of these techniques are highly dependent on the information given by the pre-determined classifications. In the case of neural networks, the classification is used to determine the error of the network and then adjust the network to minimize it, and in decision trees, the classifications are used to determine what attributes provide the most information that can be used to solve the classification puzzle. We'll look at both of these in more detail, but for now, it should be sufficient to know that both of these examples thrive on having some "supervision" in the form of pre-determined classifications.

Inductive machine learning is the process of learning a set of rules from instances (examples in a training set), or more generally speaking, creating a classifier that can be used to generalize from new instances. The process of applying supervised ML to a realworld problem is described in Figure F. The first step is collecting the dataset. If a requisite expert is available, then s/he could suggest which fields (attributes, features) are the most informative. If not, then the simplest method is that of "brute-force," which means measuring everything available in the hope that the right (informative, relevant) features can be isolated. However, a dataset collected by the "brute-force" method is not directly suitable for induction. It contains in most cases noise and missing feature values, and therefore requires significant pre-processing according to Zhang et al (Zhang, 2002).

The second step is the data preparation and data pre-processing. Depending on the circumstances, researchers have a number of methods to choose from to handle missing data (Batista, 2003). Hodge et al (Hodge, 2004) , have recently introduced a survey of contemporary techniques for outlier (noise) detection. These researchers have identified the techniques' advantages and disadvantages. Instance selection is not only used to handle noise but to cope with the infeasibility of learning from very large datasets. Instance selection in these datasets is an optimization problem that attempts to maintain the mining quality while minimizing the sample size. It reduces data and enables a data mining algorithm to function and work effectively with very large datasets. There is a variety of procedures for sampling instances from a large dataset. See figure 2 below.

Feature subset selection is the process of identifying and removing as many irrelevant and redundant features as possible (Yu, 2004) . This reduces the dimensionality of the data and enables data mining algorithms to operate faster and more effectively. The fact that many features depend on one another often unduly influences the accuracy of supervised ML classification models. This problem can be addressed by constructing new features from the basic feature set. This technique is called feature construction/ transformation. These newly generated features may lead to the creation of more concise and accurate classifiers. In addition, the discovery of meaningful features contributes to better comprehensibility of the produced classifier, and a better understanding of the learned concept.Speech recognition using hidden Markov models and Bayesian networks relies on some elements of supervision as well in order to adjust parameters to, as usual, minimize the error on the given inputs.Notice something important here: in the classification problem, the goal of the learning algorithm is to minimize the error with respect to the given inputs. These inputs, often called the "training set", are the examples from which the agent tries to learn. But learning the training set well is not necessarily the best thing to do. For instance, if I tried to teach you exclusive-or, but only showed you combinations consisting of one true and one false, but never both false or both true, you might learn the rule that the answer is always true. Similarly, with machine learning algorithms, a common problem is over-fitting the data and essentially memorizing the training set rather than learning a more general classification technique. As you might imagine, not all training sets have the inputs classified correctly. This can lead to problems if the algorithm used is powerful enough to memorize even the apparently "special cases" that don't fit the more general principles. This, too, can lead to over fitting, and it is a challenge to find algorithms that are both powerful enough to learn complex functions and robust enough to produce generalisable results.

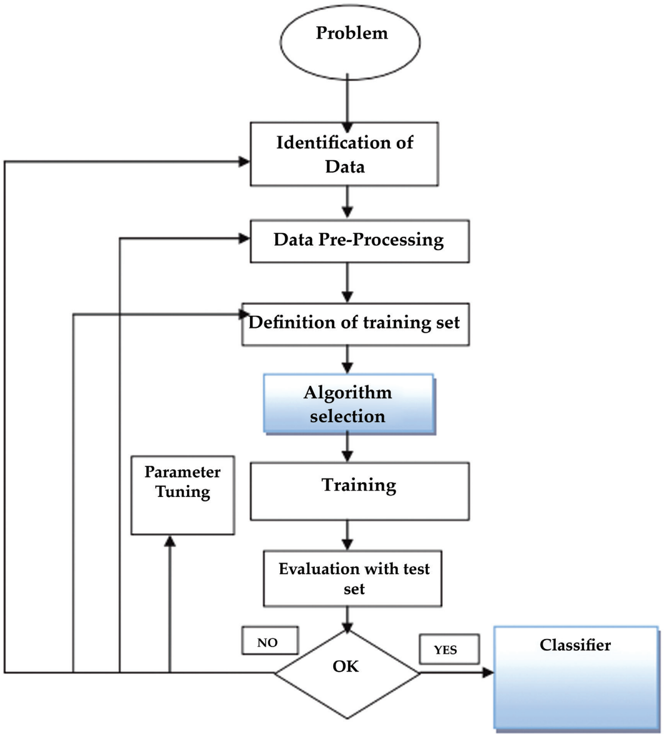


Figure 2: Machine Learning Supervise Process.

Unsupervised Learning

Unsupervised learning seems much harder: the goal is to have the computer learn how to do something that we don't tell it how to do! There are actually two approaches to unsupervised learning. The first approach is to teach the agent not by giving explicit categorizations, but by using some sort of reward system to indicate success. Note that this type of training will generally fit into the decision problem framework because the goal is not to produce a classification but to make decisions that maximize rewards. This approach nicely generalizes to the real world, where agents might be rewarded for doing certain actions and punished for doing others. Often, a form of reinforcement learning can be used for unsupervised learning, where the agent bases its actions on the previous rewards and punishments without necessarily even learning any information about the exact ways that its actions affect the world. In a way, all of this information is unnecessary because by learning a reward function, the agent simply knows what to do without any processing because it knows the exact reward it expects to achieve for each action it could take. This can be extremely beneficial in cases where calculating every possibility is very time consuming (even if all of the transition probabilities between world states were known). On the other hand, it can be very time consuming to learn by, essentially, trial and error. But this kind of learning can be powerful because it assumes no pre-discovered classification of examples. In some cases, for example, our classifications may not be the best possible. One striking exmaple is that the conventional wisdom about the game of backgammon was turned on its head when a series of computer programs (neuro-gammon and TD-gammon) that learned through unsupervised learning became stronger than the best human chess players merely by playing themselves over and over. These programs discovered some principles that surprised the backgammon experts and performed better than backgammon programs trained on pre-classified examples. A second type of unsupervised learning is called clustering. In this type of learning, the goal is not to maximize a utility function, but simply to find similarities in the training data. The assumption is often that the clusters discovered will match reasonably well with an intuitive classification. For instance, clustering individuals based on demographics might result in a clustering of the wealthy in one group and the poor in another. Although the algorithm won't have names to assign to these clusters, it can produce them and then use those clusters to assign new examples into one or the other of the clusters. This is a data-driven approach that can work well when there is sufficient data; for instance, social information filtering algorithms, such as those that Amazon.com use to recommend books, are based on the principle of finding similar groups of people and then assigning new users to groups. In some cases, such as with social information filtering, the information about other members of a cluster (such as what books they read) can be sufficient for the algorithm to produce meaningful results. In other cases, it may be the case that the clusters are merely a useful tool for a human analyst. Unfortunately, even unsupervised learning suffers from the problem of overfitting the training data. There's no silver bullet to avoiding the problem because any algorithm that can learn from its inputs needs to be quite powerful.

Unsupervised learning algorithms according to Ghahramani (Ghahramani, 2008) are designed to extract structure from data samples. The quality of a structure is measured by a cost function which is usually minimized to infer optimal parameters characterizing the hidden structure in the data. Reliable and robust inference requires a guarantee that extracted structures are typical for the data source, i.e., similar structures have to be extracted from a second sample set of the same data source. Lack of robustness is known as over fitting from the statistics and the machine learning literature. In this talk I characterize the over fitting phenomenon for a class of histogram clustering models which play a prominent role in information retrieval, linguistic and computer vision applications. Learning algorithms with robustness to sample fluctuations are derived from large deviation results and the maximum entropy principle for the learning process.

Unsupervised learning has produced many successes, such as world-champion calibre backgammon programs and even machines capable of driving cars! It can be a powerful technique when there is an easy way to assign values to actions. Clustering can be useful when there is enough data to form clusters (though this turns out to be difficult at times) and especially when additional data about members of a cluster can be used to produce further results due to dependencies in the data. Classification learning is powerful when the classifications are known to be correct (for instance, when dealing with diseases, it's generally straight-forward to determine the design after the fact by an autopsy), or when the classifications are simply arbitrary things that we would like the computer to be able to recognize for us. Classification learning is often necessary when the decisions made by the algorithm will be required as input somewhere else. Otherwise, it wouldn't be easy for whoever requires that input to figure out what it means. Both techniques can be valuable and which one you choose should depend on the circumstances--what kind of problem is being solved, how much time is allotted to solving it (supervised learning or clustering is often faster than reinforcement learning techniques), and whether supervised learning is even possible.

Algorithm Types

In the area of supervised learning which deals much with classification. These are the algorithms types:

- Linear Classifiers
  - Logical Regression
  - Naïve Bayes Classifier
  - Perceptron

    - Support Vector Machine
- Quadratic Classifiers
- K-Means Clustering
- Boosting
- Decision Tree
    - Random Forest
- Neural networks
- Bayesian Networks

Linear Classifiers: In machine learning, the goal of classification is to group items that have similar feature values, into groups. Timothy et al (Timothy Jason Shepard, 1998) stated that a linear classifier achieves this by making a classification decision based on the value of the linear combination of the features. If the input feature vector to the classifier is a real vector $\vec{x}$, then the output score is

$$y = f(\vec{w} \cdot \vec{x}) = f\left(\sum_j w_j x_j\right),$$

where $\vec{w}$ is a real vector of weights and f is a function that converts the dot product of the two vectors into the desired output. The weight vector $\vec{w}$ is learned from a set of labelled training samples. Often f is a simple function that maps all values above a certain threshold to the first class and all other values to the second class. A more complex f might give the probability that an item belongs to a certain class.

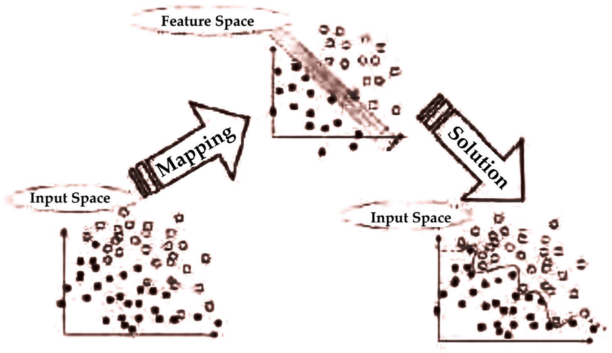
For a two-class classification problem, one can visualize the operation of a linear classifier as splitting a high-dimensional input space with a hyperplane: all points on one side of the hyper plane are classified as "yes", while the others are classified as "no". A linear classifier is often used in situations where the speed of classification is an issue, since it is often the fastest classifier, especially when $\vec{x}$ is sparse. However, decision trees can be faster. Also, linear classifiers often work very well when the number of dimensions in $\vec{x}$ is large, as in document classification, where each element in $\vec{x}$ is typically the number of counts of a word in a document (see document-term matrix). In such cases, the classifier should be wellregularized.

- ***Support Vector Machine***: A Support Vector Machine as stated by Luis et al (Luis Gonz, 2005) (SVM) performs classification by constructing an Ndimensional hyper plane that optimally separates the data into two categories. SVM models are closely related to neural networks. In fact, a SVM model using a sigmoid kernel function is equivalent to a twolayer, perceptron neural network.

Support Vector Machine (SVM) models are a close cousin to classical multilayer perceptron neural networks. Using a kernel function, SVM's are an alternative training method for polynomial, radial basis function and multi-layer perceptron classifiers in which the weights of the network are found by solving a quadratic programming problem with linear constraints, rather than by solving a non-convex, unconstrained minimization problem as in standard neural network training.

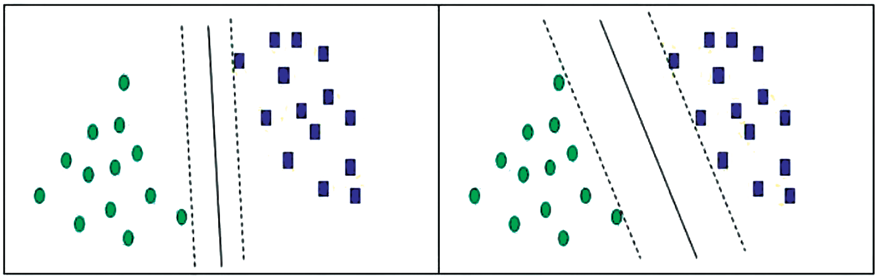
In the parlance of SVM literature, a predictor variable is called an attribute, and a transformed attribute that is used to define the hyper plane is called a feature. The task of choosing the most suitable representation is known as feature selection. A set of features that describes one case (i.e., a row of predictor values) is called a vector. So the goal of SVM modelling is to find the optimal hyper plane that separates clusters of vector in such a way that cases with one category of the target variable are on one side of the plane and cases with the other category are on the other size of the plane. The vectors near the hyper plane are the support vectors. The figure below presents an overview of the SVM process.

The SVM algorithm



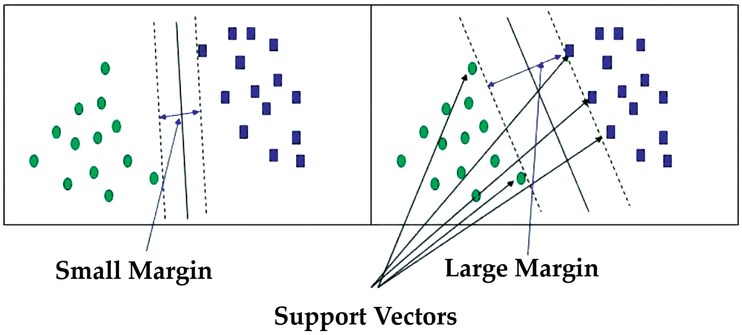
A Two-Dimensional Example

Before considering N-dimensional hyper planes, let's look at a simple 2-dimensional example. Assume we wish to perform a classification, and our data has a categorical target variable with two categories. Also assume that there are two predictor variables with continuous values. If we plot the data points using the value of one predictor on the X axis and the other on the Y axis we might end up with an image such as shown below. One category of the target variable is represented by rectangles while the other category is represented by ovals.



In this idealized example, the cases with one category are in the lower left corner and the cases with the other category are in the upper right corner; the cases are completely separated. The SVM analysis attempts to find a 1-dimensional hyper plane (i.e. a line) that separates the cases based on their target categories. There are an infinite number of possible lines; two candidate lines are shown above. The question is which line is better, and how do we define the optimal line.

The dashed lines drawn parallel to the separating line mark the distance between the dividing line and the closest vectors to the line. The distance between the dashed lines is called the margin. The vectors (points) that constrain the width of the margin are the support vectors. The following figure illustrates this.



An SVM analysis (Luis Gonz, 2005) finds the line (or, in general, hyper plane) that is oriented so that the margin between the support vectors is maximized. In the figure above, the line in the right panel is superior to the line in the left panel.

If all analyses consisted of two-category target variables with two predictor variables, and the cluster of points could be divided by a straight line, life would be easy. Unfortunately, this is not generally the case, so SVM must deal with (a) more than two predictor variables, (b) separating the points with non-linear curves, (c) handling the cases where clusters cannot be completely separated, and (d) handling classifications with more than two categories.

In this chapter, we shall explain three main machine learning techniques with their examples and how they perform in reality. These are:

- K-Means Clustering
- Neural Network
- Self Organised Map

K-Means Clustering

The basic step of k-means clustering is uncomplicated. In the beginning we determine number of cluster K and we assume the centre of these clusters. We can take any random objects as the initial centre or the first K objects in sequence can also serve as the initial centre. Then the K means algorithm will do the three steps below until convergence.

Iterate until stable (= no object move group):

1. Determine the centre coordinate
2. Determine the distance of each object to the centre
3. Group the object based on minimum distance

The Figure 3 shows a K- means flow diagram

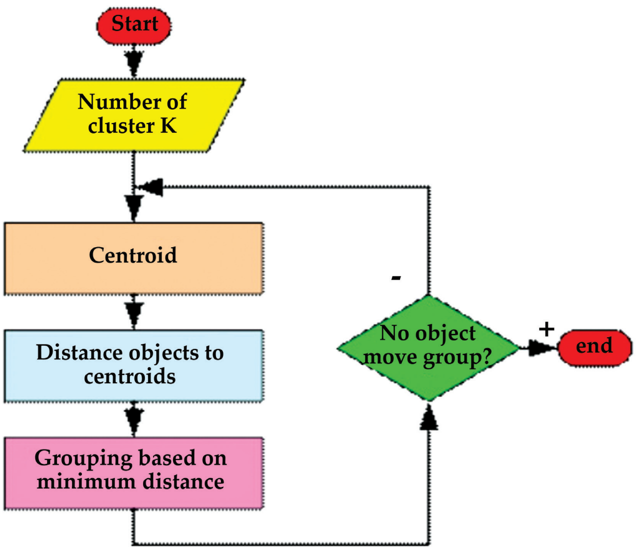

Start
Number of cluster K
Centroid
Distance objects to centroids
Grouping based on minimum distance
No object move group?
-
+
end

Figure 3: K-means iteration.

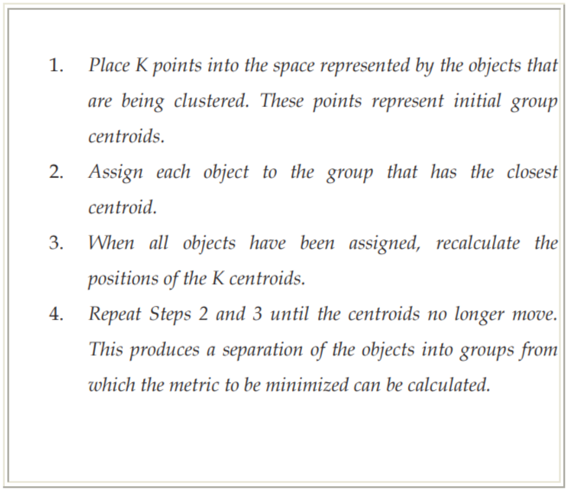
K-means (Bishop C. M., 1995) and (Tapas Kanungo, 2002) is one of the simplest unsupervised learning algorithms that solve the well known clustering problem. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed a priori. The main idea is to define k centroids, one for each cluster. These centroids shoud be placed in a cunning way because of different location causes different result. So, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate k new centroids as barycenters of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new centroid. A loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done. In other words centroids do not move any more.

Finally, this algorithm aims at minimizing an objective function, in this case a squared error function. The objective function

$$\pm = \sum_{j-1}^{k} \sum_{i-1}^{n} \left\| {}_{i}^{(j)} - {}_{j} \right\|^{2}$$

where $\left\| x_i^{(j)} - c_j \right\|^2$ is a chosen distance measure between a data point $x_i^{(j)}$ and the cluster centre c_j, is an indicator of the distance of the n data points from their respective cluster centres.

The algorithm in figure 4 is composed of the following steps:



1. *Place K points into the space represented by the objects that are being clustered. These points represent initial group centroids.*
2. *Assign each object to the group that has the closest centroid.*
3. *When all objects have been assigned, recalculate the positions of the K centroids.*
4. *Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the objects into groups from which the metric to be minimized can be calculated.*

Although it can be proved that the procedure will always terminate, the k-means algorithm does not necessarily find the most optimal configuration, corresponding to the global objective function minimum. The algorithm is also significantly sensitive to the initial randomly selected cluster centres. The k-means algorithm can be run multiple times to reduce this effect. K-means is a simple algorithm that has been adapted to many problem domains. As we are going to see, it is a good candidate for extension to work with fuzzy feature vectors.

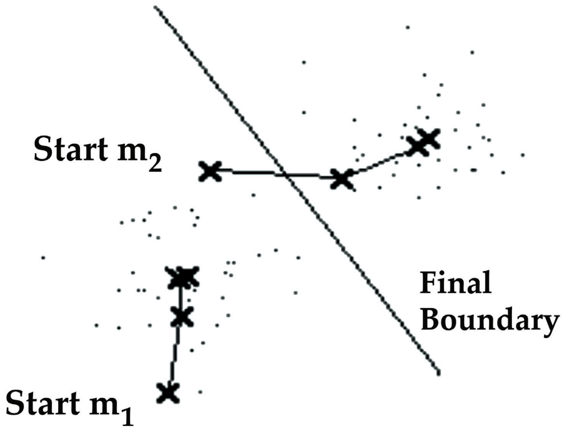
An example

Suppose that we have n sample feature vectors $x_1, x_2, ..., x_n$ all from the same class, and we know that they fall into k compact clusters, k < n. Let mi be the mean of the vectors in cluster i. If the clusters are well separated, we can use a minimum-distance classifier to separate them. That is, we can say that x is in cluster i if $\| x - m_i \|$ is the minimum of all the k distances. This suggests the following procedure for finding the k means:

- Make initial guesses for the means $m_1, m_2, ..., m_k$
- Until there are no changes in any mean

- Use the estimated means to classify the samples into clusters
- For i from 1 to k
- Replace m_i with the mean of all of the samples for cluster i
- end_for
- end_until

Here is an example showing how the means m_1 and m_2 move into the centers of two clusters.



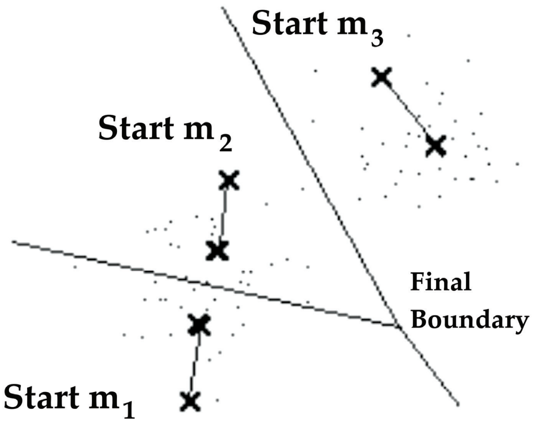
This is a simple version of the k-means procedure. It can be viewed as a greedy algorithm for partitioning the n samples into k clusters so as to minimize the sum of the squared distances to the cluster centers. It does have some weaknesses:

- The way to initialize the means was not specified. One popular way to start is to randomly choose k of the samples.
- The results produced depend on the initial values for the means, and it frequently happens that suboptimal partitions are found. The standard solution is to try a number of different starting points.
- It can happen that the set of samples closest to m_i is empty, so that m_i cannot be updated. This is an annoyance that must be handled in an implementation, but that we shall ignore.
- The results depend on the metric used to measure $\| x - m_i \|$. A popular solution is to normalize each variable by its standard

deviation, though this is not always desirable.

- The results depend on the value of k.

This last problem is particularly troublesome, since we often have no way of knowing how many clusters exist. In the example shown above, the same algorithm applied to the same data produces the following 3-means clustering. Is it better or worse than the 2-means clustering?



Unfortunately there is no general theoretical solution to find the optimal number of clusters for any given data set. A simple approach is to compare the results of multiple runs with different k classes and choose the best one according to a given criterion

Neural Network

Neural networks (Bishop C. M., 1995) can actually perform a number of regression and/or classification tasks at once, although commonly each network performs only one. In the vast majority of cases, therefore, the network will have a single output variable, although in the case of many-state classification problems, this may correspond to a number of output units (the post-processing stage takes care of the mapping from output units to output variables). If you do define a single network with multiple output variables, it may suffer from cross-talk (the hidden neurons experience difficulty learning, as they are attempting to model at least two functions at once). The best solution is usually to train separate networks for each output, then to combine them into an ensemble so that they can be run as a unit. Neural methods are:

- ***Multilayer Perceptrons***: This is perhaps the most popular network architecture in use today, due originally to Rumelhart and McClelland (1986) and discussed at length in most neural network textbooks (e.g., Bishop, 1995). This is the type of network discussed briefly in previous sections: the units each perform a biased weighted sum of their inputs and pass this activation level through a transfer function to produce their output, and the units are arranged in a layered feed forward topology. The network thus has a simple interpretation as a form of inputoutput model, with the weights and thresholds (biases) the free parameters of the model. Such networks can model functions of almost arbitrary complexity, with the number of layers, and the number of units in each layer, determining the function complexity. Important issues in Multilayer Perceptrons (MLP) design include specification of the number of hidden layers and the number of units in these layers (Bishop C. M., 1995), (D. Michie, 1994).

The number of input and output units is defined by the problem (there may be some uncertainty about precisely which inputs to use, a point to which we will return later. However, for the moment we will assume that the input variables are intuitively selected and are all meaningful). The number of hidden units to use is far from clear. As good a starting point as any is to use one hidden layer, with the number of units equal to half the sum of the number of input and output units. Again, we will discuss how to choose a sensible number later.

- ***Training Multilayer Perceptrons***: Once the number of layers, and number of units in each layer, has been selected, the network's weights and thresholds must be set so as to minimize the prediction error made by the network. This is the role of the training algorithms. The historical cases that you have gathered are used to automatically adjust the weights and thresholds in order to minimize this error. This process is equivalent to fitting the model represented by the network to the training data available. The error of a particular configuration of the network can be determined by running all the training cases through the network, comparing the actual output generated with the desired or target outputs. The differences are combined together by an error function to give the network error. The most common error functions are the sum squared error (used for regression problems), where the individual errors of output units on each case are squared and summed together, and the cross entropy functions (used for maximum likelihood classification).

In traditional modeling approaches (e.g., linear modeling) it is possible to algorithmically determine the model configuration that absolutely minimizes this error. The price paid for the greater (non-linear) modeling power of neural networks is that although we can adjust a network to lower its error, we can never be sure that the error could not be lower still.

A helpful concept here is the error surface. Each of the N weights and thresholds of the network (i.e., the free parameters of the model) is taken to be a dimension in space. The N+1th dimension is the network error. For any possible configuration of weights the error can be plotted in the N+1th dimension, forming an error surface. The objective of network training is to find the lowest point in this many-dimensional surface.

In a linear model with sum squared error function, this error surface is a parabola (a quadratic), which means that it is a smooth bowl-shape with a single minimum. It is therefore "easy" to locate the minimum.

Neural network error surfaces are much more complex, and are characterized by a number of unhelpful features, such as local minima (which are lower than the surrounding terrain, but above the global minimum), flat-spots and plateaus, saddle-points, and long narrow ravines.

It is not possible to analytically determine where the global minimum of the error surface is, and so neural network training is essentially an exploration of the error surface. From an initially random configuration of weights and thresholds (i.e., a random point on the error surface), the training algorithms incrementally seek for the global minimum. Typically, the gradient (slope) of the error surface is calculated at the current point, and used to make a downhill move. Eventually, the algorithm stops in a low point, which may be a local minimum (but hopefully is the global minimum).

- ***The Back Propagation Algorithm***: The best-known example of a neural network training algorithm is back propagation (Haykin, 19994), (Patterson, 19996), (Fausett, 19994). Modern second-order algorithms such as conjugate gradient descent and Levenberg-Marquardt (see Bishop, 1995; Shepherd, 1997) (both included in ST Neural Networks) are substantially faster (e.g., an order of magnitude faster) for many problems, but back propagation still has advantages in some circumstances, and is the easiest algorithm to understand. We will introduce this now, and discuss the more advanced algorithms later. In back propagation, the gradient vector of the error surface is calculated. This vector points along the line of steepest descent from the current point, so we know that if we move along it a “short” distance, we will decrease the error. A sequence of such moves (slowing as we near the bottom) will eventually find a minimum of some sort. The difficult part is to decide how large the steps should be.

Large steps may converge more quickly, but may also overstep the solution or (if the error surface is very eccentric) go off in the wrong direction. A classic example of this in neural network training is where the algorithm progresses very slowly along a steep, narrow, valley, bouncing from one side across to the other. In contrast, very small steps may go in the correct direction, but they also require a large number of iterations. In practice, the step size is proportional to the slope (so that the algorithm settles down in a minimum) and to a special constant: the learning rate. The correct setting for the learning rate is application-dependent, and is typically chosen by experiment; it may also be time-varying, getting smaller as the algorithm progresses.

The algorithm is also usually modified by inclusion of a momentum term: this encourages movement in a fixed direction, so that if several steps are taken in the same direction, the algorithm “picks up speed”, which gives it the ability to (sometimes) escape local minimum, and also to move rapidly over flat spots and plateaus.

The algorithm therefore progresses iteratively, through a number of epochs. On each epoch, the training cases are each submitted in turn to the network, and target and actual outputs compared and the error calculated. This error, together with the error surface gradient, is used to adjust the weights, and then the process repeats. The initial network configuration is random, and training stops when a given number of epochs elapses, or when the error reaches an acceptable level, or when the error stops improving (you can select which of these stopping conditions to use).

- ***Over-learning and Generalization***: One major problem with the approach outlined above is that it doesn’t actually minimize the error that we are really interested in - which is the expected error the network will make when new cases are submitted to it. In other words, the most desirable property of a network is its ability to generalize to new cases. In reality, the network is trained to minimize the error on the training set, and short of having a perfect and infinitely large training set, this is not the same thing as minimizing the error on the real error surface - the error surface of the underlying and unknown model (Bishop C. M., 1995).

The most important manifestation of this distinction is the problem of over-learning, or over-fitting. It is easiest to demonstrate this concept using polynomial curve fitting rather than neural networks, but the concept is precisely the same.

A polynomial is an equation with terms containing only constants and powers of the variables. For example:

$y=2x+3$

$y=3x^2+4x+1$

Different polynomials have different shapes, with larger powers (and therefore larger numbers of terms) having steadily more eccentric shapes. Given a set of data, we may want to fit a polynomial curve (i.e., a model) to explain the data. The data is probably noisy, so we don't necessarily expect the best model to pass exactly through all the points. A low-order polynomial may not be sufficiently flexible to fit close to the points, whereas a high-order polynomial is actually too flexible, fitting the data exactly by adopting a highly eccentric shape that is actually unrelated to the underlying function. See figure 4 below.

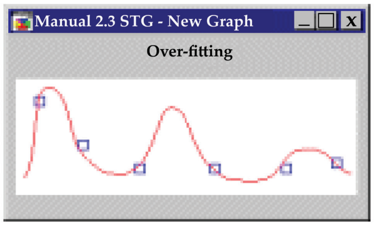


Figure 4: High-order polynomial sample.

Neural networks have precisely the same problem. A network with more weights models a more complex function, and is therefore prone to over-fitting. A network with less weight may not be sufficiently powerful to model the underlying function. For example, a network with no hidden layers actually models a simple linear function. How then can we select the right complexity of network? A larger network will almost invariably achieve a lower error eventually, but this may indicate over-fitting rather than good modeling.

The answer is to check progress against an independent data set, the selection set. Some of the cases are reserved, and not actually used for training in the back propagation algorithm. Instead, they are used to keep an independent check on the progress of the algorithm. It is invariably the case that the initial performance of the network on training and selection sets is the same (if it is not at least approximately the same, the division of cases between the two sets is probably biased). As training progresses, the training error naturally drops, and providing training is minimizing the true error function, the selection error drops too. However, if the selection error stops dropping, or indeed starts to rise, this indicates that the network is starting to overfit the data, and training should cease. When over-fitting occurs during the training process like this, it is called over-learning. In this case, it is usually advisable to decrease the number of hidden units and/or hidden layers, as the network is over-powerful for the problem at hand. In contrast, if the network is not sufficiently powerful to model the underlying function, over-learning is not likely to occur, and neither training nor selection errors will drop to a satisfactory level.

The problems associated with local minima, and decisions over the size of network to use, imply that using a neural network typically involves experimenting with a large number of different networks, probably training each one a number of times (to avoid being fooled by local minima), and observing individual performances. The key guide to performance here is the selection error. However, following the standard scientific precept that, all else being equal, a simple model is always preferable to a complex model, you can also select a smaller network in preference to a larger one with a negligible improvement in selection error.

A problem with this approach of repeated experimentation is that the selection set plays a key role in selecting the model, which means that it is actually part of the training process. Its reliability as an independent guide to performance of the model is therefore compromised - with sufficient experiments, you may just hit upon a lucky network that happens to perform well on the selection set. To add confidence in the performance of the final model, it is therefore normal practice (at least where the volume of training data allows it) to reserve a third set of cases - the test set. The final model is tested with the test set data, to ensure that the results on the selection and training set are real, and not artifacts of the training process. Of course, to fulfill this role properly the test set should be used only once - if it is in turn used to adjust and reiterate the training process, it effectively becomes selection data!

This division into multiple subsets is very unfortunate, given that we usually have less data than we would ideally desire even for a single subset. We can get around this problem by resampling. Experiments can be conducted using different divisions of the available data into training, selection, and test sets. There are a number of approaches to this subset, including random (monte-carlo) resampling, cross-validation, and bootstrap. If we make design decisions, such as the best configuration of neural network to use, based upon a number of experiments with different subset examples, the results will be much more reliable. We can then either use those experiments solely to guide the decision as to which network types to use, and train such networks from scratch with new samples (this removes any sampling bias); or, we can retain the best networks found during the sampling process, but average their results in an ensemble, which at least mitigates the sampling bias.

To summarize, network design (once the input variables have been selected) follows a number of stages:

- Select an initial configuration (typically, one hidden layer with the number of hidden units set to half the sum of the number of input and output units).
- Iteratively conduct a number of experiments with each configuration, retaining the best network (in terms of selection error) found. A number of experiments are required with each configuration to avoid being fooled if training locates a local minimum, and it is also best to resample.
- On each experiment, if under-learning occurs (the network doesn't achieve an acceptable performance level) try adding more neurons to the hidden layer(s). If this doesn't help, try adding an extra hidden layer.
- If over-learning occurs (selection error starts to rise) try removing hidden units (and possibly layers).
- Once you have experimentally determined an effective configuration for your networks, resample and generate new networks with that configuration.
- ***Data Selection***: All the above stages rely on a key assumption. Specifically, the training, verification and test data must be representative of the underlying model (and, further, the three sets must be independently representative). The old computer science adage "garbage in, garbage out" could not apply more strongly

than in neural modeling. If training data is not representative, then the model's worth is at best compromised. At worst, it may be useless. It is worth spelling out the kind of problems which can corrupt a training set:

The future is not the past. Training data is typically historical. If circumstances have changed, relationships which held in the past may no longer hold. All eventualities must be covered. A neural network can only learn from cases that are present. If people with incomes over $100,000 per year are a bad credit risk, and your training data includes nobody over $40,000 per year, you cannot expect it to make a correct decision when it encounters one of the previously-unseen cases. Extrapolation is dangerous with any model, but some types of neural network may make particularly poor predictions in such circumstances.

A network learns the easiest features it can. A classic (possibly apocryphal) illustration of this is a vision project designed to automatically recognize tanks. A network is trained on a hundred pictures including tanks, and a hundred not. It achieves a perfect 100% score. When tested on new data, it proves hopeless. The reason? The pictures of tanks are taken on dark, rainy days; the pictures without on sunny days. The network learns to distinguish the (trivial matter of) differences in overall light intensity. To work, the network would need training cases including all weather and lighting conditions under which it is expected to operate - not to mention all types of terrain, angles of shot, distances...

Unbalanced data sets. Since a network minimizes an overall error, the proportion of types of data in the set is critical. A network trained on a data set with 900 good cases and 100 bad will bias its decision towards good cases, as this allows the algorithm to lower the overall error (which is much more heavily influenced by the good cases). If the representation of good and bad cases is different in the real population, the network's decisions may be wrong. A good example would be disease diagnosis. Perhaps 90% of patients routinely tested are clear of a disease. A network is trained on an available data set with a 90/10 split. It is then used in diagnosis on patients complaining of specific problems, where the likelihood of disease is 50/50. The network will react over-cautiously and fail to recognize disease in some unhealthy patients. In contrast, if trained on the "complainants" data, and then tested on "routine" data, the network may raise a high number of false positives. In such circumstances, the data set may need to be crafted to take account of the distribution of data (e.g., you could replicate the less numerous cases, or remove some of the numerous cases), or the network's decisions modified by the inclusion of a loss matrix (Bishop C. M., 1995). Often, the best approach is to ensure even representation of different cases, then to interpret the network's decisions accordingly.

Self Organised Map

Self Organizing Feature Map (SOFM, or Kohonen) networks are used quite differently to the other networks. Whereas all the other networks are designed for supervised learning tasks, SOFM networks are designed primarily for unsupervised learning (Haykin, 19994), (Patterson, 19996), (Fausett, 19994) (Whereas in supervised learning the training data set contains cases featuring input variables together with the associated outputs (and the network must infer a mapping from the inputs to the outputs), in unsupervised learning the training data set contains only input variables. At first glance this may seem strange. Without outputs, what can the network learn? The answer is that the SOFM network attempts to learn the structure of the data.

Also Kohonen (Kohonen, 1997) explained one possible use is therefore in exploratory data analysis. The SOFM network can learn to recognize clusters of data, and can also relate similar classes to each other. The user can build up an understanding of the data, which is used to refine the network. As classes of data are recognized, they can be labelled, so that the network becomes capable of classification tasks. SOFM networks can also be used for classification when output classes are immediately available - the advantage in this case is their ability to highlight similarities between classes.

A second possible use is in novelty detection. SOFM networks can learn to recognize clusters in the training data, and respond to it. If new data, unlike previous cases, is encountered, the network fails to recognize it and this indicates novelty.

A SOFM network has only two layers: the input layer, and an output layer of radial units (also known as the topological map layer). The units in the topological map layer are laid out in space - typically in two dimensions (although ST Neural Networks also supports onedimensional Kohonen networks).

SOFM networks (Patterson, 19996) are trained using an iterative algorithm. Starting with an initially-random set of radial centres, the algorithm gradually adjusts them to reflect the clustering of the training data. At one level, this compares with the sub-sampling and KMeans algorithms used to assign centres in SOM network and indeed the SOFM algorithm can be used to assign centres for these types of networks. However, the algorithm also acts on a different level.

The iterative training procedure also arranges the network so that units representing centres close together in the input space are also situated close together on the topological map. You can think of the network's topological layer as a crude two-dimensional grid, which must be folded and distorted into the N-dimensional input space, so as to preserve as far as possible the original structure. Clearly any attempt to represent an N-dimensional space in two dimensions will result in loss of detail; however, the technique can be worthwhile in allowing the user to visualize data which might otherwise be impossible to understand.

The basic iterative Kohonen algorithm simply runs through a number of epochs, on each epoch executing each training case and applying the following algorithm:

- Select the winning neuron (the one who's centre is nearest to the input case);
- Adjust the winning neuron to be more like the input case (a weighted sum of the old neuron centre and the training case).

The algorithm uses a time-decaying learning rate, which is used to perform the weighted sum and ensures that the alterations become more subtle as the epochs pass. This ensures that the centres settle down to a compromise representation of the cases which cause that neuron to win. The topological ordering property is achieved by adding the concept of a neighbourhood to the algorithm. The neighbourhood is a set of neurons surrounding the winning neuron. The neighbourhood, like the learning rate, decays over time, so that initially quite a large number of neurons belong to the neighbourhood (perhaps almost the entire topological map); in the latter stages the neighbourhood will be zero (i.e., consists solely of the winning neuron itself). In the Kohonen algorithm, the adjustment of neurons is actually applied not just to the winning neuron, but to all the members of the current neighbourhood.

The effect of this neighbourhood update is that initially quite large areas of the network are "dragged towards" training cases - and dragged quite substantially. The network develops a crude topological ordering, with similar cases activating clumps of neurons in the topological map. As epochs pass the learning rate and neighbourhood both decrease, so that finer distinctions within areas of the map can be drawn, ultimately resulting in finetuning of individual neurons. Often, training is deliberately conducted in two distinct phases: a relatively short phase with high learning rates and neighbourhood, and a long phase with low learning rate and zero or near-zero neighbourhoods.

Once the network has been trained to recognize structure in the data, it can be used as a visualization tool to examine the data. The Win Frequencies Datasheet (counts of the number of times each neuron wins when training cases are executed) can be examined to see if distinct clusters have formed on the map. Individual cases are executed and the topological map observed, to see if some meaning can be assigned to the clusters (this usually involves referring back to the original application area, so that the relationship between clustered cases can be established). Once clusters are identified, neurons in the topological map are labelled to indicate their meaning (sometimes individual cases may be labelled, too). Once the topological map has been built up in this way, new cases can be submitted to the network. If the winning neuron has been labelled with a class name, the network can perform classification. If not, the network is regarded as undecided.

SOFM networks also make use of the accept threshold, when performing classification. Since the activation level of a neuron in a SOFM network is the distance of the neuron from the input case, the accept threshold acts as a maximum recognized distance. If the activation of the winning neuron is greater than this distance, the SOFM network is regarded as undecided. Thus, by labelling all neurons and setting the accept threshold appropriately, a SOFM network can act as a novelty detector (it reports undecided only if the input case is sufficiently dissimilar to all radial units).

SOFM networks as expressed by Kohonen (Kohonen, 1997) are inspired by some known properties of the brain. The cerebral cortex is actually a large flat sheet (about 0.5m squared; it is folded up into the familiar convoluted shape only for convenience in fitting into the skull!) with known topological properties (for example, the area corresponding to the hand is next to the arm, and a distorted human frame can be topologically mapped out in two dimensions on its surface).

Grouping Data Using Self Organise Map

The first part of a SOM is the data. Above are some examples of 3 dimensional data which are commonly used when experimenting with SOMs. Here the colours are represented in three dimensions (red, blue, and green.) The idea of the self-organizing maps is to project the n-dimensional data (here it would be colour and would be 3 dimensions) into something that be better understood visually (in this case it would be a 2 dimensional image map).

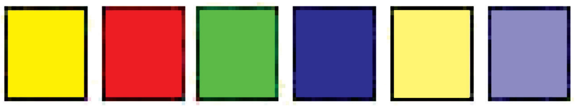


Figure 5: Sample Data.

In this case one would expect the dark blue and the greys to end up near each other on a good map and yellow close to both the red and the green. The second components to SOMs are the weight vectors. Each weight vector has two components to them which I have here attempted to show in the image below. The first part of a weight vector is its data. This is of the same dimensions as the sample vectors and the second part of a weight vector is its natural location. The good thing about colour is that the data can be shown by displaying the color, so in this case the color is the data, and the location is the x,y position of the pixel on the screen.

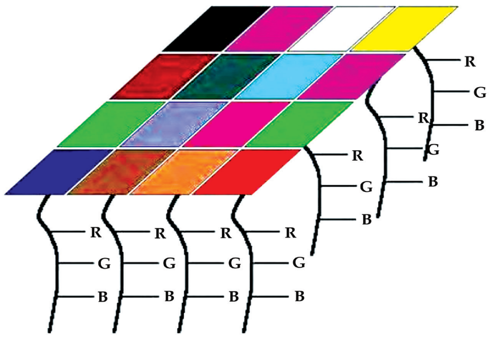


Figure 6: 2D Array Weight of Vector.

In this example, 2D array of weight vectors was used and would look like figure 5 above. This picture is a skewed view of a grid where you have the n-dimensional array for each weight and each weight has its own unique location in the grid. Weight vectors don’t necessarily have to be arranged in 2 dimensions, a lot of work has been done using SOMs of 1 dimension, but the data part of the weight must be of the same dimensions as the sample vectors.Weights are sometimes referred to as neurons since SOMs are actually neural networks. SOM Algorithm. The way that SOMs go about organizing themselves is by competeting for representation of the samples. Neurons are also allowed to change themselves by learning to become more like samples in hopes of winning the next competition. It is this selection and learning process that makes the weights organize themselves into a map representing similarities.

So with these two components (the sample and weight vectors), how can one order the weight vectors in such a way that they will represent the similarities of the sample vectors? This is accomplished by using the very simple algorithm shown here.

```
Initialize Map
For t from 0 to 1

Randomly select a sample
Get best matching unit
Scale neighbors
Increase t a small amount

End for
```

Figure 7: A Sample SOM Algorithm.

The first step in constructing a SOM is to initialize the weight vectors. From there you select a sample vector randomly and search the map of weight vectors to find which weight best represents that sample. Since each weight vector has a location, it also has neighbouring weights that are close to it. The weight that is chosen is rewarded by being able to become more like that randomly selected sample vector. In addition to this reward, the neighbours of that weight are also rewarded by being able to become more like the chosen sample vector. From this step we increase t some small amount because the number of neighbours and how much each weight can learn decreases over time. This whole process is then repeated a large number of times, usually more than 1000 times.

In the case of colours, the program would first select a color from the array of samples such as green, then search the weights for the location containing the greenest color. From there, the colour surrounding that weight are then made more green. Then another color is chosen, such as red, and the process continues. They processes are:

A. Initializing the Weights

Here are screen shots of the three different ways which decided to initialize the weight vector map. We should first mention the palette here. In the java program below there are 6 intensities of red, blue, and green displayed, it really does not take away from the visual experience. The actual values for the weights are floats, so they have a bigger range than the six values that are shown in figure 7 below.

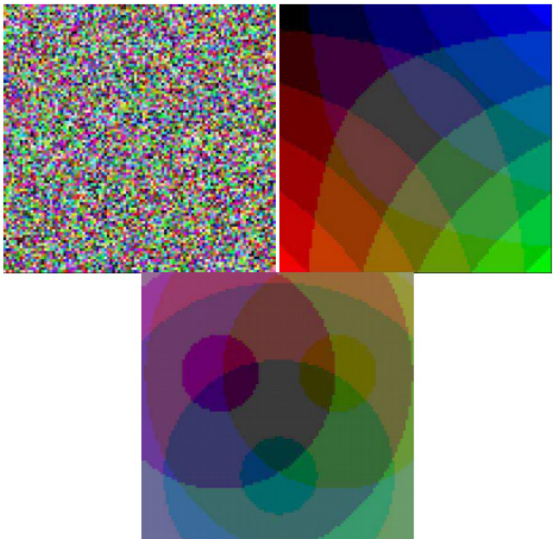


Figure 8: Weight Values.

There are a number of ways to initialize the weight vectors. The first you can see is just give each weight vector random values for its data. A screen of pixels with random red, blue, and green values is shown above on the left. Unfortunately calculating SOMs according to Kohonen (Kohonen, 1997) is very computationally expensive, so there are some variants of initializing the weights so that samples that you know for a fact are not similar start off far away. This way you need less iteration to produce a good map and can save yourself some time.

Here we made two other ways to initialize the weights in addition to the random one. This one is just putting red, blue, green, and black at all four corners and having them slowly fade toward the center. This other one is having red, green, and blue equally distant from one another and from the center.

B. Get Best Matching Unit

This is a very simple step, just go through all the weight vectors and calculate the distance from each weight to the chosen sample vector. The weight with the shortest distance is the winner. If there are more than one with the same distance, then the winning weight is chosen randomly among the weights with the shortest distance. There are a number of different ways for determining what distance actually means mathematically. The most common method is to use the Euclidean distance:

$$\sqrt{\sum_{i=0}^{n} x_i^2}$$

where x[i] is the data value at the ith data member of a sample and n is the number of dimensions to the sample vectors.

In the case of colour, if we can think of them as 3D points, each component being an axis. If we have chosen green which is of the value (0,6,0), the color light green (3,6,3) will be closer to green than red at (6,0,0).

Light green = Sqrt((3-0)^2+(6-6)^2+(3-0)^2) = 4.24
Red = Sqrt((6-0)^2+(0-6)^2+(0-0)^2) = 8.49

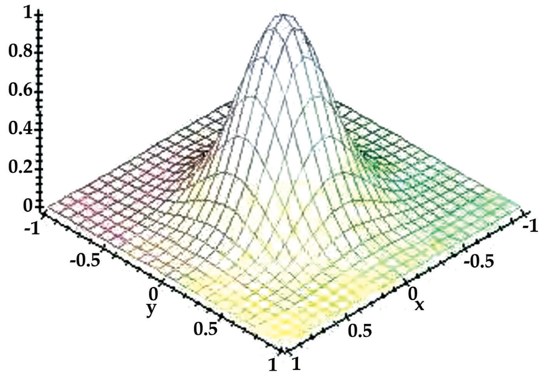
So light green is now the best matching unit, but this operation of calculating distances and comparing them is done over the entire map and the wieght with the shortest distance to the sample vector is the winner and the BMU. The square root is not computed in the java program for speed optimization for this section.

C. Scale Neighbors

1. Determining Neighbors

There are actually two parts to scaling the neighboring weights: determining which weights are considered as neighbors and how much each weight can become more like the sample vector. The neighbors of a winning weight can be determined using a number of different methods. Some use concentric squares, others hexagons, I opted to use a gaussian function where every point with a value above zero is considered a neighbor.

As mentioned previously, the amount of neighbors decreases over time. This is done so samples can first move to an area where they will probably be, then they jockey for position. This process is similar to coarse adjustment followed by fine tuning. The function used to decrease the radius of influence does not really matter as long as it decreases, we just used a linear function.



$$f := (x, y) \rightarrow \mathrm{e}^{\left(-6.666666667\,\sqrt{x^{2}+y^{2}}^{\,2}\right)}$$

Figure 9: A graph of SOM Neighbour's determination.

Figure 8 above shows a plot of the function used. As the time progresses, the base goes towards the centre, so there are less neighbours as time progresses. The initial radius is set really high, some value near the width or height of the map.

2. Learning

The second part to scaling the neighbours is the learning function. The winning weight is rewarded with becoming more like the sample vector. The neighbours also become more like the sample vector. An attribute of this learning process is that the farther away the neighbour is from the winning vector, the less it learns. The rate at which the amount a weight can learn decreases and can also be set to whatever you want. I chose to use a gaussian function. This function will return a value ranging between 0 and 1, where each neighbor is then changed using the parametric equation. The new color is:

*Current color**(1.-*t*) + *sample vector***t*

So in the first iteration, the best matching unit will get a t of 1 for its learning function, so the weight will then come out of this process with the same exact values as the randomly selected sample.

Just as the amount of neighbors a weight has falls off, the amount a weight can learn also decreases with time. On the first iteration, the winning weight becomes the sample vector since t has a full range of from 0 to 1. Then as time progresses, the winning weight becomes slightly more like the sample where the maximum value of t decreases. The rate at which the amount a weight can learn falls of linearly. To depict this visually, in the previous plot, the amount a weight can learn is equivalent to how high the bump is at their location. As time progresses, the height of the bump will decrease. Adding this function to the neighbourhood function will result in the height of the bump going down while the base of the bump shrinks.

So once a weight is determined the winner, the neighbours of that weight is found and each of those neighbours in addition to the winning weight change to become more like the sample vector.

Determining the Quality of SOMs

Below is another example of a SOM generated by the program using 500 iterations in figure 9. At first glance you will notice that similar colour is all grouped together yet again. However, this is not always the case as you can see that there are some colour who are surrounded by colour that are nothing like them at all. It may be easy to point this out with colour since this is something that we are familiar with, but if we were using more abstract data, how would we know that since two entities are close to each other means that they are similar and not that they are just there because of bad luck?

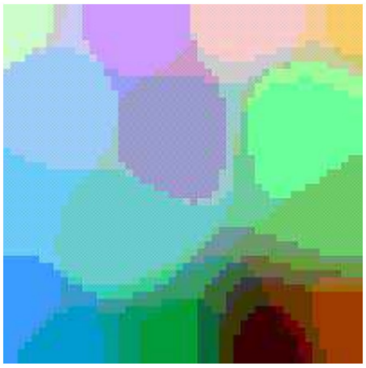


Figure 10: SOM Iteration.

There is a very simple method for displaying where similarities lie and where they do not. In order to compute this we go through all the weights and determine how similar the neighbors are. This is done by calculating the distance that the weight vectors make between the each weight and each of its neighbors. With an average of these distances a color is then assigned to that location. This procedure is located in Screen.java and named public void update_bw().

If the average distance were high, then the surrounding weights are very different and a dark color is assigned to the location of the weight. If the average distance is low, a lighter color is assigned. So in areas of the center of the blobs the colour are the same, so it should be white since all the neighbors are the same color. In areas between blobs where there are similarities it should be not white, but a light grey. Areas where the blobs are physically close to each other, but are not similar at all there should be black. See Figure 8 below

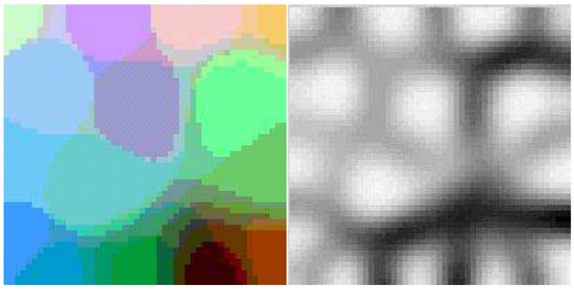


Figure 11: A sample allocation of Weight in Colour.

As shown above, the ravines of black show where the colour may be physically close to each other on the map, but are very different from each other when it comes to the actual values of the weights. Areas where there is a light grey between the blobs represent a true similarity. In the pictures above, in the bottom right there is black surrounded by colour which are not very similar to it. When looking at the black and white similarity SOM, it shows that black is not similar to the other colour because there are lines of black representing no similarity between those two colour. Also in the top corner there is pink and nearby is a light green which are not very near each other in reality, but near each other on the colored SOM. Looking at the black and white SOM, it clearly shows that the two not very similar by having black in between the two colour.

With these average distances used to make the black and white map, we can actually assign each SOM a value that determines how good the image represents the similarities of the samples by simply adding these averages.

Advantages and Disadvantages of SOM

Self organise map has the following advantages:

- Probably the best thing about SOMs that they are very easy to understand. It's very simple, if they are close together and there is grey connecting them, then they are similar. If there is a black ravine between them, then they are different. Unlike Multidimensional Scaling or N-land, people can quickly pick up on how to use them in an effective manner.
- Another great thing is that they work very well. As I have shown you they classify data well and then are easily evaluate for their own quality so you can actually calculated how good a map is and how strong the similarities between objects are.

These are the disadvantages:

- One major problem with SOMs is getting the right data. Unfortunately you need a value for each dimension of each member of samples in order to generate a map. Sometimes this simply is not possible and often it is very difficult to acquire all of this data so this is a limiting feature to the use of SOMs often referred to as missing data.
- Another problem is that every SOM is different and finds different similarities among the sample vectors. SOMs organize sample data so that in the final product, the samples are usually surrounded by similar samples, however similar samples are not always near each other. If you have a lot of shades of purple, not always will you get one big group with all the purples in that cluster, sometimes the clusters will get split and there will be two groups of purple. Using colour we could tell that those two groups in reality are similar and that they just got split, but with most data, those two clusters will look totally unrelated. So a lot of maps need to be constructed in order to get one final good map.
- The final major problem with SOMs is that they are very computationally expensive which is a major drawback since as the dimensions of the data increases, dimension reduction visualization techniques become more important, but unfortunately then time to compute them also increases. For calculating that black and white similarity map, the more neighbours you use to calculate the distance the better similarity map you will get, but the number of distances the algorithm needs to compute increases exponentially.

REFERENCES

1. Allix, N. M. (2003, April). Epistemology And Knowledge Management Concepts And Practices. Journal of Knowledge Management Practice .
2. Alpaydin, E. (2004). Introduction to Machine Learning. Massachusetts, USA: MIT Press.
3. Anderson, J. R. (1995). Learning and Memory. Wiley, New York, USA.
4. Anil Mathur, G. P. (1999). Socialization influences on preparation for later life. Journal of Marketing Practice: Applied Marketing Science , 5 (6,7,8), 163 - 176.
5. Ashby, W. R. (1960). Design of a Brain, The Origin of Adaptive Behaviour. John Wiley and Son.
6. Batista, G. &. (2003). An Analysis of Four Missing Data Treatment Methods for Suppervised Learning. Applied Artificial Intelligence , 17, 519-533.
7. Bishop, C. M. (1995). Neural Networks for Pattern Recognition. Oxford, England: Oxford University Press.
8. Bishop, C. M. (2006). Pattern Recognition and Machine Learning (Information Science and Statistics). New York, New York: Springer Science and Business Media.
9. Block H, D. (1961). The Perceptron: A Model of Brian Functioning. 34 (1), 123-135.
10. Carling, A. (1992). Introducing Neural Networks . Wilmslow, UK: Sigma Press.
11. D. Michie, D. J. (1994). Machine Learning, Neural and Statistical Classification. Prentice Hall Inc.
12. Fausett, L. (19994). Fundamentals of Neural Networks. New York: Prentice Hall.
13. Forsyth, R. S. (1990). The strange story of the Perceptron. Artificial Intelligence Review , 4 (2), 147-155.
14. Friedberg, R. M. (1958). A learning machine: Part, 1. IBM Journal , 2-13.
15. Ghahramani, Z. (2008). Unsupervised learning algorithms are designed to extract structure from data. 178, pp. 1-8. IOS Press.
16. Gillies, D. (1996). Artificial Intelligence and Scientific Method. OUP Oxford.

17. Haykin, S. (19994). Neural Networks: A Comprehensive Foundation. New York: Macmillan Publishing.
18. Hodge, V. A. (2004). A Survey of Outlier Detection Methodologies. Artificial Intelligence Review , 22 (2), 85-126.
19. Holland, J. (1980). Adaptive Algorithms for Discovering and Using General Patterns in Growing Knowledge Bases Policy Analysis and Information Systems. 4 (3).
20. Hunt, E. B. (1966). Experiment in Induction.
21. Ian H. Witten, E. F. (2005). Data Mining Practical Machine Learning and Techniques (Second edition ed.). Morgan Kaufmann.
22. Jaime G. Carbonell, R. S. (1983). Machine Learning: A Historical and Methodological Analysis. Association for the Advancement of Artificial Intelligence , 4 (3), 1-10.
23. Kohonen, T. (1997). Self-Organizating Maps.
24. Luis Gonz, l. A. (2005). Unified dual for bi-class SVM approaches. Pattern Recognition , 38 (10), 1772-1774.
25. McCulloch, W. S. (1943). A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophysics , 115-133.
26. Mitchell, T. M. (2006). The Discipline of Machine Learning. Machine Learning Department technical report CMU-ML-06-108, Carnegie Mellon University.
27. Mooney, R. J. (2000). Learning Language in Logic. In L. N. Science, Learning for Semantic Interpretation: Scaling Up without Dumbing Down (pp. 219-234). Springer Berlin / Heidelberg.
28. Mostow, D. (1983). Transforming declarative advice into effective procedures: a heuristic search cxamplc In I?. S. Michalski,. Tioga Press.
29. Nilsson, N. J. (1982). Principles of Artificial Intelligence (Symbolic Computation / Artificial Intelligence). Springer.
30. Oltean, M. (2005). Evolving Evolutionary Algorithms Using Linear Genetic Programming. 13 (3), 387 - 410 .
31. Orlitsky, A., Santhanam, N., Viswanathan, K., & Zhang, J. (2005). Convergence of profile based estimators. Proceedings of International Symposium on Information Theory. Proceedings. International Symposium on, pp. 1843 - 1847. Adelaide, Australia: IEEE.
32. Patterson, D. (19996). Artificial Neural Networks. Singapore: Prentice Hall.

33. R. S. Michalski, T. J. (1983). Learning from Observation: Conceptual Clustering. TIOGA Publishing Co.
34. Rajesh P. N. Rao, B. A. (2002). Probabilistic Models of the Brain. MIT Press.
35. Rashevsky, N. (1948). Mathematical Biophysics:Physico-Mathematical Foundations of Biology. Chicago: Univ. of Chicago Press.
36. Richard O. Duda, P. E. (2000). Pattern Classification (2nd Edition ed.).
37. Richard S. Sutton, A. G. (1998). Reinforcement Learning. MIT Press.
38. Ripley, B. (1996). Pattern Recognition and Neural Networks. Cambridge University Press.
39. Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain . Psychological Review , 65 (6), 386-408.
40. Russell, S. J. (2003). Artificial Intelligence: A Modern Approach (2nd Edition ed.). Upper Saddle River, NJ, NJ, USA: Prentice Hall.
41. Ryszard S. Michalski, J. G. (1955). Machine Learning: An Artificial Intelligence Approach (Volume I). Morgan Kaufmann .
42. Ryszard S. Michalski, J. G. (1955). Machine Learning: An Artificial Intelligence Approach.
43. Selfridge, O. G. (1959). Pandemonium: a paradigm for learning. In The mechanisation of thought processes. H.M.S.O., London. London.
44. Sleeman, D. H. (1983). Inferring Student Models for Intelligent CAI. Machine Learning. Tioga Press. s : ACM Press.
45. Timothy Jason Shepard, P. J. (1998). Decision Fusion Using a Multi-Linear Classifier . In Proceedings of the International Conference on Multisource-Multisensor Information Fusion.
46. Tom, M. (1997). Machibe Learning. Machine Learning, Tom Mitchell, McGraw Hill, 1997: McGraw Hill.
47. Trevor Hastie, R. T. (2001). The Elements of Statistical Learning. New york, NY, USA: Springer Science and Business Media.
48. Widrow, B. W. (2007). Adaptive Inverse Control: A Signal Processing Approach. Wiley-IEEE Press.
49. Y. Chali, S. R. (2009). Complex Question Answering: Unsupervised Learning Approaches and Experiments. Journal of Artificial Intelligent Research , 1-47.

50. Yu, L. L. (2004, October). Efficient feature Selection via Analysis of Relevance and Redundacy. JMLR , 1205-1224.

51. Zhang, S. Z. (2002). Data Preparation for Data Mining. Applied Artificial Intelligence. 17, 375 - 381.

CHAPTER
3

DATA MINING WITH SKEWED DATA

Manoel Fernando Alonso Gadi[1], Alair Pereira do Lago[2] , and Jorn Mehnen[3]

[1]Milton Keynes United Kingdom

[2]Depart. de Ciencia de Computacao, Inst. de Matematica e Estatistica Universidade de Sao Paulo Brazil

[3]Decision Engineering Centre, Cranfield University Cranfield, Bedfordshire MK43 0AL United Kingdom

In this chapter, we explore difficulties one often encounters when applying machine learning techniques to real-world data, which frequently show skewness properties. A typical example from industry where skewed data is an intrinsic problem is fraud detection in finance data. In the following we provide examples, where appropriate, to facilitate the understanding of data mining of skewed data. The topics explored include but are not limited to: data preparation, data cleansing, missing values, characteristics

Citation: Manoel Fernando Alonso Gadi, Alair Pereira do Lago and Jorn Mehnen (February 1st 2010). "Data Mining with Skewed Data", New Advances in Machine Learning Yagang Zhang, IntechOpen, DOI: 10.5772/9382.

construction, variable selection, data skewness, objective functions, bottom line expected prediction, limited resource situation, parametric optimisation, model robustness and model stability.

INTRODUCTION

In many contexts like in a new e-commerce website, fraud experts start investigation procedures only after a user makes a claim. Rather than working reactively, it would be better for the fraud expert to act proactively before a fraud takes place. In this e-commerce example, we are interested in classifying sellers into legal customers or fraudsters. If a seller is involved in a fraudulent transaction, his/her license to sell can be revoked by the e-business. Such a decision requires a degree of certainty, which comes with experience. In general, it is only after a fraud detection expert has dealt with enough complains and enough data that he/she acquired a global understanding of the fraud problem. Quite often, he/she is exposed to a huge number of cases in a short period of time. This is when automatic procedures, commonly computer based, can step in trying to reproduce expert procedures thus giving experts more time to deal with harder cases. Hence, one can learn from fraud experts and build a model for fraud. Such a model requires fraud evidences that are commonly present in fraudulent behavior. One of the difficulties of fraud detection is that fraudsters try to conceal themselves under a "normal" behavior. Moreover, fraudsters rapidly change their modus operandi once it is discovered. Many fraud evidences are illegal and justify a more drastic measure against the fraudster. However, a single observed indicator is often not strong enough to be considered a proof and needs to be evaluated as one variable among others. All variables taken together can indicate high probability of fraud. Many times, these variables appear in the literature by the name of characteristics or features. The design of these characteristics to be used in a model is called characteristics extraction or feature extraction.

DATA PREPARATION

Characteristics Extraction

One of the most important tasks on data preparation is the conception of characteristics. Unfortunately, this depends very much on the application (See also the discussions in Section 4 and 5). For fraud modelling for instance, one starts from fraud expert experience, determine significant characteristics as fraud indicators, and evaluates them. In this evaluation, one is interested in measuring how well these characteristics:

- covers (is present in) the true fraud cases;
- and how clearly they discriminate fraud from non-fraud behavior.

In order to cover as many fraud cases as possible, one may verify how many of them are covered by the characteristics set. The discrimination power of any of these characteristics can be evaluated by their odds ratio. If the probability of the event (new characteristics) in each of two compared classes (fraud and non-fraud in our case) are p_f (first class) and p_n (second class), then the odds ratio is:

$$OR = \frac{p_f/(1-p_f)}{p_n/(1-p_n)} = \frac{p_f(1-p_n)}{p_n(1-p_f)}.$$

An odds ratio equals to 1 describes the characteristics as equally probable in both classes (fraud and non-fraud). The more this ratio is greater/less than 1, the more likely this characteristic is in the first/second class than in the other one.

Data Cleansing

In many manuals on best practice in model development, a chapter on data consistency checks, or data cleansing, is present. The main reason for this is to avoid waisting all the effort applied in the model development stage, because of data inconsistency invalidating the dataset in use.

Here we understand data consistency checks as being a set of expert rules to check whether a characteristic follows an expected behaviour. These expert rules can be based on expert knowledge or common sense. For example, a common error when filling in the date-of-birth section in a credit card application form is to put the current year instead of the year the person was actually born. In most countries an under sixteen year old can not have a credit card. Therefore, an easy way of checking this inconsistency is to simply calculate the applicant's age and check if it falls within a valid range. With more information available, more complex checks can be applied, such as, e.g. matching name with gender or street name with post code. In some cases the model developer has access to reports stored in Management Information Systems (MIS). If that is the case, it is a highly recommended idea to calculate key population indicators and compare these to portfolio reports. For example, in a credit card industry environment one can check the volume of applications; accept rate, decline rate and take-up rate and others. It can also be useful to check the univariate distribution of each variable including the percentage of outliers, missing and miscellaneous values.

Having identified the characteristics that contain errors, the next step is to somehow fix the inconsistencies or minimise their impact in the final model. Here we list, in the form of questions, some good practices in data cleansing used by the industry that can sometimes improve model performance, increase generalisation power and finally, but no less important, make models less vulnerable to fraud and faults.

1. Is it possible to fix the errors by running some codes on the dataset? Sometimes wrong values have a one-to-one mapping to the correct values. Therefore, the best strategy is to make the change in the development dataset and to carry on with the development. It is important that these errors are fixed for the population the model will be applied to as well. This is because both developing and applying populations must be consistent, otherwise fixing the inconsistency would worsen the model performance rather than improving it;
2. Is a small number of attributes1 (less than 5%) impacting only few rows (less than 5%)? In this case, one can do a bivariate analysis to determine if it is possible to separate these values into a default (or fault) group. Another option is to drop the rows. However, this tactic might turn out to be risky (see section about missing values);
3. Is the information value of the problematic attribute(s) greater than for the other attributes combined? Consider dropping this characteristic and demand fixing;
4. Is it possible to allow outliers? Simply dropping them might be valid if there are few or there are invalid values. Change their values to the appropriate boundary could also be valid. For example, if an acceptable range for yearly income is [1,000;100,000] MU2 and an applicant has a yearly income of 200,000 MU then it should be changed to 100,000 MU. This approach is often referred to as truncated or censored modelling Schneider (1986).
5. Finally, in an industry environment, when an MIS is available, one can check for the acceptance rate or number of rows to be similar to the reports? It is very common for datasets to be

corrupted after transferring them from a Mainframe to Unix or Windows machines.

DATA SKEWNESS

A dataset for modelling is perfectly balanced when the percentage of occurrence of each class is 100/n, where n is the number of classes. If one or more classes differ significantly from the others, this dataset is called skewed or unbalanced. Dealing with skewed data can be very tricky. In the following sections we explore, based on our experiments and literature reviews, some problems that can appear when dealing with skewed data. Among other things, the following sections will explain the need for stratified sampling, how to handle missing values carefully and how to define an objective function that takes the different costs for each class into account.

Missing Values

Missing values are of little importance when dealing with balanced data, but can become extremely harmful or beneficial when dealing with skewed data. See how the example below looks harmful at first glance, but indeed exposes a very powerful characteristic.

Table 1 shows an example of a characteristic called Transaction Amount. By looking at the first line of the table one may conclude that the number of missing values is small (1.98%) and decide not to investigate any further. Breaking it down into fraudulent and legitimate transactions, one can see that 269 (32.5%) data items whose values are missings are frauds, which is nearly 9 times bigger than the overall fraud rate in our dataset (1,559/41,707 = 3.74% see Table 2).

Table 1: Fraud/legitimate distribution

Population	# transaction	# missing	% missing
Total	41707	825	1.98%
Fraud	1559	269	17.26%
Legitimate	40148	556	1.39%

Investigating even further, by analysing the fraud rates by ranges as shown in table 2, one can see that the characteristic being analysed really helps to predict fraud; on the top of this, missing values seem to be the most powerful attribute for this characteristic.

Table 2: Transaction amount bivariate

Trans. amount	# frauds	# trans.	Fraud rate
Missings	269	825	32.61%
0,100	139	14639	0.95%
101,500	235	10359	2.27%
501,1000	432	8978	4.81%
1001,5000	338	4834	6.99%
5001,+inf	146	2072	7.05%
Total	1559	41707	3.74%

When developing models with balanced data, in most cases one can argue that it is good practice to avoid giving prediction to missing values (as a separate attribute or dummy), especially, if this attribute ends up with dominating the model. However, when it comes to unbalanced data, especially with fraud data, some specific value may have been intentionally used by the fraudster in order to bypass the system's protection. In this case, one possible explanation could be a system failure, where all international transaction are not being correctly currency converted when passed to the fraud prevention system. This loophole may have been found by some fraudster and exploited. Of course, this error would have passed unnoticed had one not paid attention to any missing or common values in the dataset.

DERIVED CHARACTERISTICS

New or derived characteristics construction is one of, if not the, most important part of modelling. Some important phenomena mapped in nature are easily explained using derived variables. For example, in elementary physics speed is a derived variable of space over time. In data mining, it is common to transform date of birth into age or, e.g., year of study into primary, secondary, degree, master, or doctorate. Myriad ways exist to generate derived characteristics. In the following we give three typical examples:

1. Transformed characteristics: transform characteristics to gain either simplicity or generalisation power. For example, date of birth into age, date of starting a relationship with a company into time on books, and years of education into education level;
2. Time series characteristics: a new characteristic built based on a group of historical months of a given characteristic. Examples

are average balance of a bank account within the last 6 months, minimum balance of a bank account within the last 12 months, and maximum days in arrears3 over the last 3 months;

3. Interaction: variable combining two or more different characteristics (of any type) in order to map interesting phenomena. For example, average credit limit utilization = average utilization / credit limit.

CATEGORISATION (GROUPING)

Categorisation (discretising, binning or grouping) is any process that can be applied to a characteristic in order to turn it into categorical values Witten & Franku (2005). For example, let us suppose that the variable age ranges from 0 to 99 and all values within this interval are possible. A valid categorisation in this case could be:

1. category 1: if age is between 1 and 17;
2. category 2: if age is between 18 and 30;
3. category 3: if age is between 31 and 50;
4. category 4: if age is between 51 and 99.

Among others, there are three main reasons for categorising a characteristic: firstly, to increase generalisation power; secondly, to be able to apply certain types of methods, such as, e.g. a Generalised Linear Model4 (GLM) Witten & Franku (2005), or a logistic regression using Weight of Evidence5 (WoE) formulations Agterberg et al. (1993); thirdly, to add stability to the model by getting rid of small variations causing noise. Categorisation methods include:

1. *Equal width*: corresponds to breaking a characteristic into groups of equal width. In the age example we easily break age into 5 groups of 20 decimals in each: 0-19, 20-39, 40-59, 60-79, 80-99.
2. *Percentile*: this method corresponds to breaking the characteristic into groups of equal volume, or percentage, of occurrences. Note that in this case groups will have different widths. In some cases breaking a characteristic into many groups may not be possible because occurrences are concentrated. A possible algorithm in pseudo code to create percentile groups is:

```
Nc <- Number of categories to be created
Nr <- Number of rows
Size <- Nr/Nc
Band_start [0] <- Minimum (Value (characteristic[0..Nr]))
//Dataset needs to be sorted by the characteristic to be grouped
For j = 1 .. Nc {
For i = 1 .. Size {
Value_end <- Value(characteristic[i+Nc*Size])
}
Band_end[j] <- Value_end
Band_start[j+1] <- Value_end
}
```

3. *Bivariate grouping*: this method corresponds to using the target variable to find good breaking points for the ranges of each group. It is expected that, in doing so, groups created using a bivariate process have a lower drop in information value, whilst it can improve the generalisation by reducing the number of attributes. One can do this in a spreadsheet by recalculating the odds and information value every time one collapses neighbouring groups with either similar odds, non-monotonic odds or a too small population percentage.

Next, we present one possible process of grouping the characteristic age using a bivariate grouping analysis. For visual simplicity the process starts with groups of equal width, each containing 10 units (see Table 3). The process consists of eliminating intervals without monotonic odds, grouping similar odds and guaranteeing a minimal percentage of individuals in each group.

Table 3: Age bivariate step 1/4

Age	goods	bads	Odds	%tot
0-9	100	4	25.00	1.83%
10-19	200	10	20.00	3.70%
20-29	600	50	12.00	11.46%
30-39	742	140	5.30	15.55%
40-49	828	160	5.18	17.42%
50-59	1000	333	3.00	23.50%
60-69	500	125	4.00	11.02%
70-79	300	80	3.75	6.70%
80-89	200	100	2.00	5.29%
90-99	100	100	1.00	3.53%
Total	4570	1102	4.15	100.00%

Table 4: Age bivariate step 2/4

Age	goods	bads	Odds	%tot
0-9	100	4	25.00	1.83%
10-19	200	10	20.00	3.70%
20-29	600	50	12.00	11.46%
30-39	742	140	5.30	15.55%
40-49	828	160	5.18	17.42%
50-79	**1800**	**538**	**3.35**	**41.22%**
80-89	200	100	2.00	5.29%
90-99	100	100	1.00	3.53%
Total	4570	1102	4.15	100.00%

The result of the first step, eliminating intervals without monotonic odds can be seen in Table 4. Here bands 50-59 (odds of 3.00), 60-69 (odds of 4.00) and 70-79 (odds of 3.75) have been merged, as shown in boldface. One may notice that merging bands 50-59 and 60-69 would result in a group with odds of 3.28; hence resulting in the need to merge with band 70-79 to yield monotonic odds.

Table 5: Age bivariate step 3/4

Age	goods	bads	Odds	%tot
0-9	100	4	25.00	1.83%
10-19	200	10	20.00	3.70%
20-29	600	50	12.00	11.46%
30-49	**1575**	**300**	**5.23**	**32.97%**
50-79	1800	538	3.35	41.22%
80-89	200	100	2.00	5.29%
90-99	100	100	1.00	3.53%
Total	4570	1102	4.15	100.00%

By using, for example, 0.20 as the minimum allowed odds difference, Table 5 presents the result of step two where bands 30-39 (odds of 5.30) and 40-49 (odds of 5.18) have been merged. This is done to increase model stability. One may notice that odds retrieved from the development become expected odds in a future application of the model. Therefore, these values will vary around the expectation. By grouping these two close odds, one tries to avoid that a reversal in odds may happen by pure random variation.

Table 6: Age bivariate step 4/4

Age	goods	bads	Odds	%tot
0-19	**300**	**14**	**21.43**	**5.54%**
20-29	600	50	12.00	11.46%
30-49	1575	300	5.23	32.97%
50-79	1800	538	3.35	41.22%
80-89	200	100	2.00	5.29%
90-99	100	100	1.00	3.53%
Total	4570	1102	4.15	100.00%

For the final step, if we assume 2% to be be the minimum allowed percentage of the population in each group. This forces band 0-9 (1.83% of total) to be merged with one of its neighbours; in this particular case, there is only the option to merge with band 10-19. Table 6 shows the final result of the bivariate grouping process after all steps are finished.

SAMPLING

As computers become more and more powerful, sampling, to reduce the sample size for model development, seems to be losing attention and importance. However, when dealing with skewed data, sampling methods remain extremely important Chawla et al. (2004); Elkan (2001). Here we present two reasons to support this argument.

First, to help to ensure that no over-fitting happens in the development data, a sampling method can be used to break the original dataset into training and holdout samples. Furthermore, a stratified sampling can help guarantying that a desirable factor has similar percentage in both training and holdout samples. In our work Gadi et al. (2008b), for example, we executed a random sampling process to select multiple splits of 70% and 30%, as training and holdout samples. However, after evaluating the output datasets we decided to redo the sampling process using stratified sampling by fraud/legitimate flag.

Second, to improve the model prediction, one may apply an over- or under- sampling process to take the different cost between classes into account. Cost-sensitive procedure Elkan (2001) replicates (oversampling) the minority (fraud) class according to its cost in order to balance different costs for false positives and false negatives. In Gadi et al. (2008a) we achieved interesting results by applying a cost-sensitive procedure.

Two advantages of a good implementation of a cost-sensitive procedure are: first, it can enable changes in cut-off to the optimal cut-off, For example, in fraud detection, if the cost tells one, a cost-sensitive procedure will consider a transaction with as little as 8% of probability of fraud as a potential fraud to be investigated; second, if the cost-sensitive procedure considers cost per transaction, such an algorithm may be able to optimise decisions by considering the product [probability of event] x [value at risk], and decide on investigating those transactions in which this product is bigger.

CHARACTERISTICS SELECTION

Characteristics selection, also known as feature selection, variable selection, feature reduction, attribute selection or variable subset selection, is commonly used in machine learning and statistical techniques to select a subset of relevant characteristics for the building of more robust models Witten & Franku (2005).

Decision trees do characteristics selection as part of their training process when selecting only the most powerful characteristics in each subpopulation, leaving out all weak or highly correlated characteristics. Bayesian nets link different characteristics by cause and effect rules, leaving out non-correlated characteristics Charniak (1991). Logistic Regression does not use any intrinsic strategy for removing weak characteristics; however, in most implementations methods such as forward, backward and stepwise are always available. In our tests, we have applied a common approach in the bank industry that is to consider only those characteristics with information value greater than a given percentage threshold.

OBJECTIVE FUNCTIONS

When defining an objective function, in order to compare different models, we found in our experiments that two facts are especially important:

1. We have noticed that academia and industry speak in different languages. In the academic world, measures such as Kolmogorov Smirnov (KS) Chakravarti et al. (1967) or Receiver Operating Characteristic (ROC curve) Green & Swets (1966) are the most common; in industry, on the other hand, rates are more commonly used. In the fraud detection area for example it is common to find measures such as hit rate (confidence) and detection rate (cover). Hit rate and detection rate are two different dimensions and they

are not canonical. To optimise a problem with an objective having two outcomes is not a simple task Trautmann & Mehnen (2009). In our work in fraud detection we avoided this two-objective function by calculating one single outcome value: the total cost of fraud;

2. In an unbalanced environment it is common to find that not only the proportion between classes differs, but also the cost between classes. For example, in the fraud detection environment, the loss by fraud when a transaction is fraudulent is much bigger than the cost to call a customer to confirm whether he/she did or did not do the transaction.

BOTTOM LINE EXPECTED PREDICTION

The problem of finding the best model can be computationally expensive, as there are many parameters involved in such a search. For this reason, it is very common for model developers to get satisfied with suboptimal models. A question equally difficult to answer, in general, is how far we are from an optimum. We do not intend to respond to this question here; what we want to address is a list of ways to help the model developer to estimate a minimum acceptable performance before getting close to the end of the model development. In our fraud analysis we found two good options for estimating a bottom line for expected suboptimal cost: a first option could be the cost resulting from a Na¨ıve Bayes model. It is important to notice that Na¨ıve Bayes does not need any grouping, characteristics selection or parameter tuning; a second option could be to consider the cost from a first "quick and dirty" model developed using the method chosen by the model developer.

LIMITED RESOURCE SITUATION

Many real-world application present limited resource problems. This can make the decision of what is the best model different compared to a model without restrictions. In a hospital, for example, there may be a limited number of beds for patients; in a telephone costumer service facility, there may be a limited number of attendants; in the fraud detection world the number of people available to handle manual transactions is in general fixed; and the number of transactions each member of fraud detection can handle per day is also fixed due to practical reasons. In such applications, being aware of the capacity rate becomes very important. It is also extremely important for the model outcome to indicate the probability6 of the event rather than providing a simple yes/no response. By having the outcome as a probability, models can be compared using for example, cutoffs that keep the selecting rate equal to the capacity rate. In fraud detection, comparing models detection rate and hit rate fixing for example 1000 transaction to be investigated.

PARAMETRIC OPTIMISATION

Once we have the data and the optimisation criteria, the following questions have to be answered:

Which classification method is recommended for producing the best model for any given application?

Which parameter set should be used?

For instance, we can apply classification methods such as: Neural Networks (NN), Bayesian Networks (BN), Na¨ıve Bayes (NB), Artificial Immune Systems (AIS) and Decision Trees (DT), Support Vector Machines (SVM), Logistic Regression and others. In fact, there is not a final and unique answer to this first question. Support Vector Machines, for instance, is known to be very effective for data with a very large number of characteristics and is reported to perform well in categorisation problems in Information Retrieval. However, our experience with SVM on fraud data did not meet our expectations. For many parameter sets, the method did not even converge to a final model and this behaviour for unbalanced data is reported to not be uncommon.

In order to assess methods many factors can be used including the chosen optimisation criteria, scalability, time for classification and time spent in training, and sometimes more abstract criteria as time to understand how the method works. Most of the time, when a method is published, or when an implementation is done, the method depends on parameter choices that may influence the final results significantly. Default parameters, in general, are a good start. However, most of the time, they are far from producing the best model. This comprises with our experience with many methods in many different areas of Computer Science. This is particular true for classification problems with skewed data.

Quite often we see comparisons against known methods where the comparison is done by applying a special parameter variation strategy (sometimes a parameter optimisation) for the chosen method while not fairly conducing the same procedure for the other methods. In general, for the other methods, default parameters, or a parameter set published in some previous work is used. Therefore, it is not a surprise that the new proposed method wins. At a first glance, the usage of the default parameter set may seem to be fair and this bias is often reproduced in publications. However, using default sets can be biased by the original training set and, thus, not be fair.

Parameter optmisation takes time and is rarely conduced. For a fair comparison, we argue that one has to fine tune the parameters for all compared method. This can be done, for instance, via an exhaustive search of the parameter space if this search is affordable, or some kind of sampling like in Genetic Algorithm (GA) (see Figure 1). Notice, that the final parameter set cannot be claimed to be optimal in this case.

Unfortunately, this sampling procedure is not as easy as one may suppose. There is not a single best universal optimisation algorithm for all problems (No Free Lunch theorem - Wolpert and Macready 1997 Wolpert & Macready (1997)). Even the genetic algorithm scheme as shown in Figure 1 might require parameter adjustment. According to our experience, we verified that a simple mistake in the probability distribution computation may drive the results to completely different and/or misleading results. A good genetic algorithm requires expertise, knowledge about the problem that should be optimised by the GA, an intelligent design, and resources. The more, the better. These considerations also imply that comparisons involving methods with suboptimal parameter sets depend very much on how well each parameter space sampling was conduced.

ROBUSTNESS OF PARAMETERS

After the parameter optimisation has been conducted, it can be advantageous or desirable to have the optimised parameters independent from the training set, i.e. they can be applied to different datasets of the same problem. In this case we can call this parameter set robust.

When the parameter are not robust, the optimisation process is not as strong as expected since the obtained optimised parameter set has no or little generalisation power. In this case, in our experiments, we found that it is a good approach to sacrifice some prediction power in order to gain robustness in the parameter set. Note that a procedure using n-fold cross validation could lead to a parameter set that is more independent from a dataset. However, we choose to present a different approach which also generates robust parameter sets with more control of what is happening during the process. This procedure is based on repeated sampling from the development dataset into training and holdout samples. Then, we applied parameter optimisation and choose the set of parameters which is the best in average over all splits at the same time.

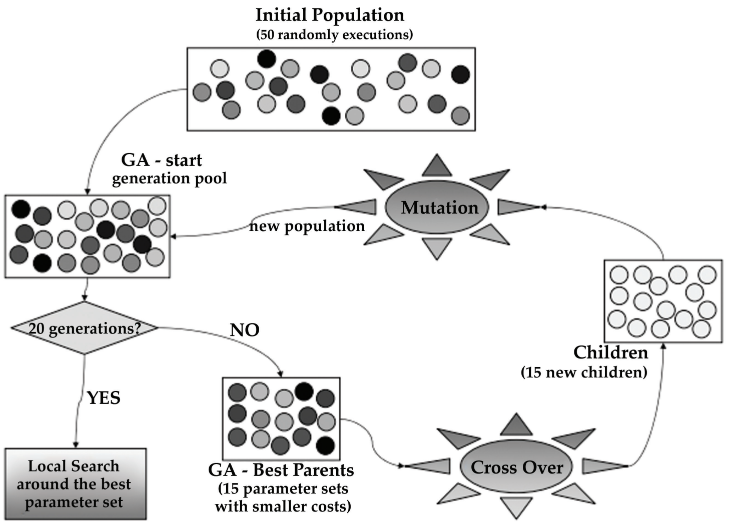


Figure 1: Genetic Algorithm for parameters optimisation. We start with an initial pool of e.g. 50 random individuals having a certain fitness, followed by e.g. 20 Genetic Algorithm (GA) generations. Each GA generation combines two randomly selected candidates among the best e.g. 15 from previous generation. This combination performs: crossover, mutation, random change or no action for each parameter independently. As the generation goes by, the chance of no action increases. In the end, one may perform a local search around the optimised founded by GA optimisation. Retrieved from Gadi et al. Gadi et al. (2008b).

In our work, in order to rewrite the optimisation function that should be used in a GA algorithm, we have used a visualization procedure with computed costs for many equally spaced parameter sets in the parameter space. After having defined a good optimisation function, due to time constraints, we did not proceed with another GA optimisation, but we reused our initial runs used in the visualization, with the following kind of multiresolution optimisation Kim & Zeigler (1996) (see Figure 2):

- we identified those parameters that have not changed, and we frozen these values for these respective parameters;
- with any other parameter, we screened the 20 best parameter sets for every split and identified a reasonable range;
- for all non-robust parameters, we chose an integer step s so the search space did not explode;

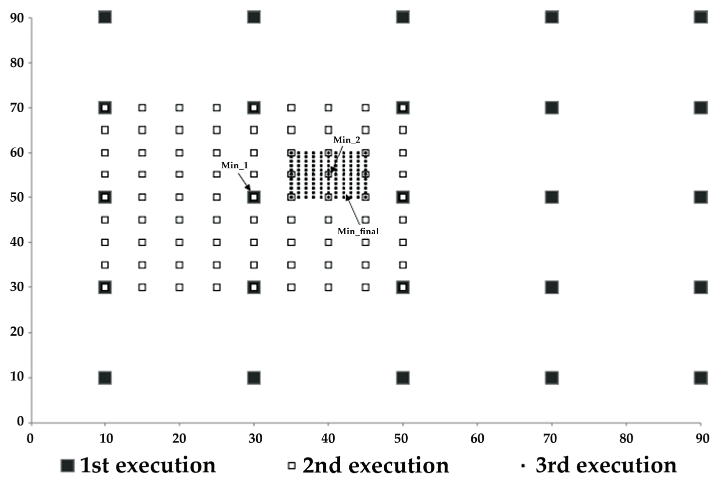


Figure 2: An example of the multiresolution optimisation that was applied in order to find robust parameters. In this example one can see two parameters in the search space and three steps of this multiresolution optimisation. For the parameter represented in horizontal line, the search space in first step ranges from 10 to 90 with step size 20 and the minimum was found for 30. In the second step, the scale ranges from 10 to 50 with step size 5 and the minimum was found for 40. In third step, it ranges from 35 to 45, with step size 1, which is equivalent to a local exhaustive search in this neighborhood. Retrieved from Gadi et al. Gadi et al. (2008a).

- we evaluated the costs for all possible combinations according to the search space defined above and found the parameter set P that brings the minimum average cost among all the different used splits;
- if the parameter set P was at the border of the search space, we shifted this search space by one step in the direction of this border and repeated last step until we found this minimum P in the inner area of the search space;

- we zoomed the screening in on the neighborhood of P, refined steps s, and repeated the process from then on until no refinement was possible.

MODEL STABILITY

In industry, generally the aim of modelling is to apply a model to a real situation and to generate profit, either by automating decision making where a model was not previously available or replacing old models by a new and improved one. For doing so, most model development processes rely on past information for their training. Therefore, it is very important to be able to assess whether or not a model is still fit for propose when it is in use, and to have a set of actions to expand the model's life span. In this section we explore advantages of using out-oftime samples, monitoring reports, stability by vintage, vintage selection and how to deal with different scales over time.

Out-of-time

An Out-Of-Time sample (OOT) is any sample of the same phenomena used in the model development that is not in the development window8 , historic vintages or observation point selected for development. In most cases in reality a simple split of the development sample into training and testing data cannot identify a real over-fitting of the model Sobehart et al. (2000). Therefore, the most appropriated approach to identify this change is either to select a vintage or observation point posterior to the development window or select this previously to the development window. The second approach gives the extra advantage of using the most up-to-date information for the development.

Monitoring reports

The previous action, OOT, should be best done before the actual model implementation; after that, it becomes important to evaluate whether the implemented model still delivers a good prediction. For this purpose, it is crucial to create a set of period based monitoring reports to track the model's performance and stability over time.

Stability by Vintage

Stability by vintage corresponds to breaking the development sample down by time within the development window and evaluate the model's performance in all of the different periods within the data. For example, if one has information collected from January 08 to December 08, a good stability by vintage analysis would be to evaluate the model's performance over each month of 2008. This tends to increase the chance of a model to be stable after its implementation.

Vintage Selection

Many phenomena found in nature, and even in human behaviour, repeat themselves year after year in a recurrent manner; this is known as seasonality. Choosing a development window from a very atypical month of the year can be very misleading; in credit cards, for example, choosing only December as the development time window can lead to overestimation of expected losses since this is the busiest time of the year. Two approaches intend to mitigate this problem. Both approaches are based on selecting the broadest development window possible. One common window size is 12 months, allowing the window to cover the whole year. Please notice, there is no need to fix the start of the window to any particular month. The first approach corresponds to simply develop the model with this pool of observation points; it is expected for the model to be an average model that will work throughout the year. A second approach is to introduce a characteristic indicating the "month of the year" the information was collected from, or any good combination of it, and then to develop the model. As a result, one would expect a model that adjusts better to each observation point in the development window.

Different Scale over Time

Another common problem applies to the situation where characteristic values fall outside the training sample boundaries or some unknown attributes occur. To reduce the impact of this problem, one can always leave the groups with the smallest and biggest boundaries as negative infinite and positive infinite, respectively, for example, changing [0,10];[11,20];[21,30] to]−∞,10];[11,20];[21,+∞[. Furthermore, undefined values could always be assigned to a default group. For example, if for a numeric characteristic a non-numeric value ocurrs it could be assigned to a default group.

FINAL REMARKS

This work provided a brief introduction to pratical problem solving for machine learning with skewed data sets. Classification methods are generally not designed to cope with skewed data, thus, various action have to be taken when dealing with imbalanced data sets. For a reader looking for more information about the field we can recommend a nice editorial by Chawla et al. ? and three conference proceedings Chawla et al. (2003); Dietterich et al. (2000); Japkowicz (2000). In addition, good algorithm examples can be found in Weka Witten & Franku (2005) and SAS Delwiche & Slaughter (2008).

Perhaps, most solutions that deal with skewed data do some sort of sampling (e.g: with undersampling, oversampling, cost sensitive training Elkan (2001), etc.). These contributions are effective Gadi et al. (2008a) and quite well known nowadays. This text provides recommendations for practitioners who are facing data mining problems due to skewed data.

Details on the experiments can be found at Gadi et al. (2008b) and Gadi et al. (2008a), which presents an application of Artificial Immune Systems on credit card fraud detection. Finally, another subject explored in this work was the importance of parametric optimization for chosing a good classification method for skewed data. We also suggested a proceedure for parametric optimization.

REFERENCES

1. Agterberg, F. P., Bonham-Carter, G. F., Cheng, Q. & Wright, D. F. (1993). Weights of evidence modeling and weighted logistic regression for mineral potential mapping, pp. 13–32.
2. Chakravarti, I. M., Laha, R. G. & Roy, J. (1967). Handbook of Methods of Applied Statistics, Vol. I, John Wiley and Sons, USE.
3. Charniak, E. (1991). Bayesians networks without tears, AI Magazine pp. 50 – 63.
4. Chawla, N. V., Japkowicz, N. & Kotcz, A. (2004). Special issue on learning from imbalanced data sets, SIGKDD Explorations 6(1): 1–6.
5. Chawla, N. V., Japkowicz, N. & Kotcz, A. (eds) (2003). Proceedings of the ICML'2003 Workshop on Learning from Imbalanced Data Sets.
6. Delwiche, L. & Slaughter, S. (2008). The Little SAS Book: A Primer, SAS Publishing.
7. Dietterich, T., Margineantu, D., Provost, F. & Turney, P. (eds) (2000). Proceedings of the ICML'2000 Workshop on Cost-Sensitive Learning.
8. Elkan, C. (2001). The foundations of cost-sensitive learning, IJCAI, pp. 973–978. URL: citeseer.ist.psu.edu/elkan01foundations.html
9. Gadi, M. F. A., Wang, X. & Lago, A. P. d. (2008a). Comparison with parametric optimization in credit card fraud detection, ICMLA '08: Proceedings of the 2008 Seventh International Conference on Machine Learning and Applications, IEEE Computer Society, Washington, DC, USA, pp. 279–285.
10. Gadi, M. F., Wang, X. & Lago, A. P. (2008b). Credit card fraud detection with artificial immune system, ICARIS '08: Proceedings of the 7th international conference on Artificial Immune Systems, Springer-Verlag, Berlin, Heidelberg, pp. 119–131.
11. Green, D. M. & Swets, J. A. (1966). Signal Detection Theory and Psychophysics, John Wiley and Sons. URL: http://www.amazon.co.uk/exec/obidos/ASIN/B000WSLQ76/citeulike00-21
12. Japkowicz, N. (ed.) (2000). Proceedings of the AAAI'2000 Workshop on Learning from Imbalanced Data Sets. AAAI Tech Report WS-00-05.
13. Kim, J. & Zeigler, B. P. (1996). A framework for multiresolution optimization in a parallel/distributed environment: simulation of hierarchical gas, J. Parallel Distrib. Comput. 32(1): 90–102.

14. Schneider, H. (1986). Truncated and censored samples from normal populations, Marcel Dekker, Inc., New York, NY, USA.
15. Sobehart, J., Keenan, S. & Stein, R. (2000). Validation methodologies for default risk models, pp. 51–56.
16. URL: http://www.moodyskmv.com/research/files/wp/p51p56.pdf
17. Trautmann, H. & Mehnen, J. (2009). Preference-based pareto optimization in certain and noisy environments, Engineering Optimization 41: 23–38.
18. Witten, I. H. & Franku, E. (2005). Data Mining: Practical Machine Learning Tools and Techniques (Second Edition), Elsevier.
19. Wolpert, D. H. & Macready, W. G. (1997). No free lunch theorems for optimization, Evolutionary Computation, IEEE Transactions on 1(1): 67–82. URL: http://dx.doi.org/10.1109/4235.585893

SECTION II
MACHINE LEARNING TECHNIQUES AND APPLICATIONS

CHAPTER
4

SURVEY OF MACHINE LEARNING ALGORITHMS FOR DISEASE DIAGNOSTIC

Meherwar Fatima[1], Maruf Pasha[2]

[1]Institute of CS & IT, The Women University Multan, Multan, Pakistan

[2]Department of Information Technology, Bahauddin Zakariya University, Multan, Pakistan

ABSTRACT

In medical imaging, Computer Aided Diagnosis (CAD) is a rapidly growing dynamic area of research. In recent years, significant attempts are made for the enhancement of computer aided diagnosis applications because errors in medical diagnostic systems can result in seriously misleading medical treatments. Machine learning is important in Computer Aided Diagnosis. After using an easy equation, objects such as organs may not be indicated accurately. So, pattern recognition fundamentally involves learning from examples. In the field of bio-medical, pattern recognition and machine

Citation: Fatima, M. and Pasha, M. (2017), "Survey of Machine Learning Algorithms for Disease Diagnostic". *Journal of Intelligent Learning Systems and Applications*, **9**, 1-16. doi: 10.4236/jilsa.2017.91001.

learning promise the improved accuracy of perception and diagnosis of disease. They also promote the objectivity of decision-making process. For the analysis of high-dimensional and multimodal bio-medical data, machine learning offers a worthy approach for making classy and automatic algorithms. This survey paper provides the comparative analysis of different machine learning algorithms for diagnosis of different diseases such as heart disease, diabetes disease, liver disease, dengue disease and hepatitis disease. It brings attention towards the suite of machine learning algorithms and tools that are used for the analysis of diseases and decision-making process accordingly.

Keywords: Machine Learning, Artificial Intelligence, Machine Learning Techniques

INTRODUCTION

Artificial Intelligence can enable the computer to think. Computer is made much more intelligent by AI. Machine learning is the subfield of AI study. Various researchers think that without learning, intelligence cannot be developed. There are many types of Machine Learning Techniques that are shown in Figure 1. Supervised, Unsupervised, Semi Supervised, Reinforcement, Evolutionary Learning and Deep Learning are the types of machine learning techniques.

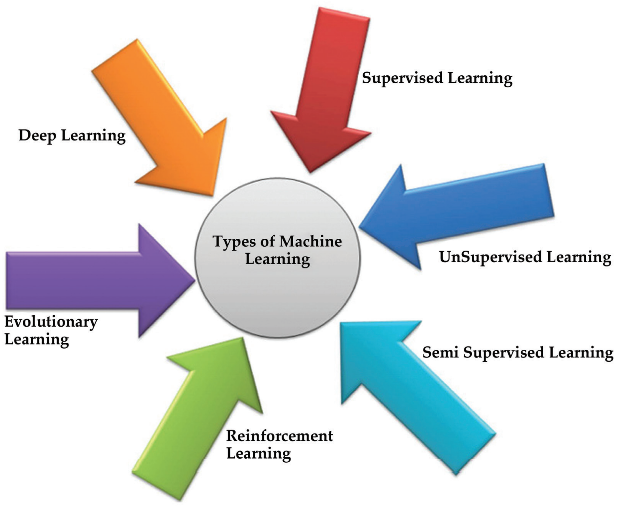


Figure 1. Types of machine learning techniques.

These techniques are used to classify the data set.

1) Supervised learning: Offered a training set of examples with suitable targets and on the basis of this training set, algorithms respond correctly to all feasible inputs. Learning from exemplars is another name of Supervised Learning. Classification and regression are the types of Supervised Learning.

Classification: It gives the prediction of Yes or No, for example, "Is this tumor cancerous?", "Does this cookie meet our quality standards?"

Regression: It gives the answer of "How much" and "How many".

2) Unsupervised learning: Correct responses or targets are not provided. Unsupervised learning technique tries to find out the similarities between the input data and based on these similarities, un-supervised learning technique classify the data. This is also known as density estimation. Unsupervised learning contains clustering [1] .

Clustering: it makes clusters on the basis of similarity.

3) Semi supervised learning: Semi supervised learning technique is a class of supervised learning techniques. This learning also used unlabeled data for training purpose (generally a minimum amount of labeled-data with a huge amount of unlabeled-data). Semi-supervised learning lies between unsupervised-learning (unlabeled-data) and supervised learning (labeled-data).

4) Reinforcement learning: This learning is encouraged by behaviorist psychology. Algorithm is informed when the answer is wrong, but does not inform that how to correct it. It has to explore and test various possibilities until it finds the right answer. It is also known as learning with a critic. It does not recommend improvements. Reinforcement learning is different from supervised learning in the sense that accurate input and output sets are not offered, nor sub- optimal actions clearly précised. Moreover, it focuses on on-line performance.

5) Evolutionary Learning: This biological evolution learning can be considered as a learning process: biological organisms are adapted to make progress in their survival rates and chance of having off springs. By using the idea of fitness, to check how accurate the solution is, we can use this model in a computer [1] .

6) Deep learning: This branch of machine learning is based on set of algorithms. In data, these learning algorithms model high-level abstraction. It uses deep graph with various processing layer, made up of many linear and nonlinear transformation.

Pattern recognition process and data classification are valuable for a long time. Humans have very strong skill for sensing the environment. They take action against what they perceive from environment [2] . Big data turns into Chunks due to multidisciplinary combined effort of machine learning, databases and statistics. Today, in medical sciences disease diagnostic test is a serious task. It is very important to understand the exact diagnosis of patients by clinical examination and assessment. For effective diagnosis and cost effective management, decision support systems that are based upon computer may play a vital role. Health care field generates big data about clinical assessment, report regarding patient, cure, follow-ups, medication etc. It is complex to arrange in a suitable way. Quality of the data organization has been affected due to inappropriate management of the data. Enhancement in the amount of data needs some proper means to extract and process data effectively and efficiently [3] . One of the many machine-learning applications is employed to build such classifier that can divide the data on the basis of their attributes. Data set is divided into two or more than two classes. Such classifiers are used for medical data analysis and disease detection.

Initially, algorithms of ML were designed and employed to observe medical data sets. Today, for efficient analysis of data, ML recommended various tools. Especially in the last few years, digital revolution has offered comparatively low- cost and obtainable means for collection and storage of data. Machines for data collection and examination are placed in new and modern hospitals to make them capable for collection and sharing data in big information systems. Technologies of ML are very effective for the analysis of medical data and great work is done regarding diagnostic problems. Correct diagnostic data are presented as a medical record or reports in modern hospitals or their particular data section. To run an algorithm, correct diagnostic patient record is entered in a computer as an input. Results can be automatically obtained from the previous solved cases. Physicians take assistance from this derived classifier while diagnosing novel patient at high speed and enhanced accuracy. These classifiers can be used to train non-specialists or students to diagnose the problem [4] .

In past, ML has offered self-driving cars, speech detection, efficient web search, and improved perception of the human generation. Today machine learning is present everywhere so that without knowing it, one can possibly use it many times a day. A lot of researchers consider it as the excellent way in moving towards human level. The machine learning techniques discovers electronic health record that generally contains high dimensional patterns and multiple data sets. Pattern recognition is the theme of MLT that offers support to predict and make decisions for diagnosis and to plan treatment. Machine learning algorithms are capable to manage huge number of data, to combine data from dissimilar resources, and to integrate the background information in the study [3] .

DIAGNOSIS OF DISEASES BY USING DIFFERENT MACHINE LEARNING ALGORITHMS

Many researchers have worked on different machine learning algorithms for disease diagnosis. Researchers have been accepted that machine-learning algorithms work well in diagnosis of different diseases. Figurative approach of diseases diagnosed by Machine Learning Techniques is shown in Figure 2. In this survey paper diseases diagnosed by MLT are heart, diabetes, liver, dengue and hepatitis.

Heart Disease

Otoom et al. [5] presented a system for the purpose of analysis and monitoring. Coronary artery disease is detected and monitored by this proposed system. Cleveland heart data set is taken from UCI. This data set consists of 303 cases and 76 attributes/features. 13 features are used out of 76 features. Two tests with three algorithms Bayes Net, Support vector machine, and Functional Trees FT are performed for detection purpose. WEKA tool is used for detection. After experimenting Holdout test, 88.3% accuracy is attained by using SVM technique.

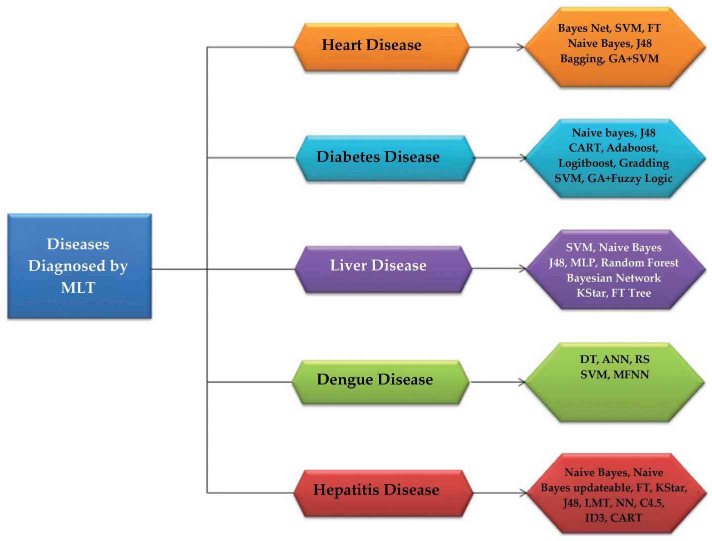

Diseases Diagnosed by MLT
Heart Disease
Bayes Net, SVM, FT Naive Bayes, J48 Bagging, GA+SVM
Diabetes Disease
Naive bayes, J48 CART, Adaboost, Logitboost, Gradding SVM, GA+Fuzzy Logic
Liver Disease
SVM, Naive Bayes J48, MLP, Random Forest Bayesian Network KStar, FT Tree
Dengue Disease
DT, ANN, RS SVM, MFNN
Hepatitis Disease
Naive Bayes, Naive Bayes updateable, FT, KStar, J48, LMT, NN, C4.5, ID3, CART

Figure 2: Diseases diagnosed by MLT.

In Cross Validation test, Both SVM and Bayes net provide the accuracy of 83.8%. 81.5% accuracy is attained after using FT. 7 best features are picked up by using Best First selection algorithm. For validation Cross Validation test are used. By applying the test on 7 best selected features, Bayes Net attained 84.5% of correctness, SVM provides 85.1% accuracy and FT classify 84.5% correctly.

Vembandasamy et al. [6] performed a work, to diagnose heart disease by using Naive Bayes algorithm. Bayes theorem is used in Naive Bayes. Therefore, Naive Bayes have powerful independence assumption. The employed data-set are obtained from one of the leading diabetic research institute in Chennai. Data set consists of 500 patients. Weka is used as a tool and executes classification by using 70% of Percentage Split. Naive Bayes offers 86.419% of accuracy.

Use of data mining approaches has been suggested by Chaurasia and Pal [7] for heart disease detection. WEKA data mining tool is used that contains a set of machine learning algorithms for mining purpose. Naive

Bayes, J48 and bagging are used for this perspective. UCI machine learning laboratory provide heart disease data set that consists of 76 attributes. Only 11 attributes are employed for prediction. Naive bayes provides 82.31% accuracy. J48 gives 84.35% of correctness. 85.03% of accuracy is achieved by Bagging. Bagging offers better classification rate on this data set.

Parthiban and Srivatsa [8] put their effort for diagnosis of heart disease in diabetic patients by using the methods of machine learning. Algorithms of Naive Bayes and SVM are applied by using WEKA. Data set of 500 patients is used that are collected from Research Institute of Chennai. Patients that have the disease are 142 and disease is missing in 358 patients. By using Naive Bayes Algorithm 74% of accuracy is obtained. SVM provide the highest accuracy of 94.60.

Tan et al. [9] proposed hybrid technique in which two machine-learning algorithms named Genetic Algorithm (G.A) and Support Vector Machine (SVM) are joined effectively by using wrapper approach. LIBSVM and WEKA data mining tool are used in this analysis. Five data sets (Iris, Diabetes disease, disease of breast Cancer, Heart and Hepatitis disease) are picked up from UC Irvine machine learning repository for this experiment. After applying GA and SVM hybrid approach, 84.07% accuracy is attained for heart disease. For data set of diabetes 78.26% accuracy is achieved. Accuracy for Breast cancer is 76.20%. Correctness of 86.12% is resulting for hepatitis disease. Graphical representation of Accuracy according to time for detection of heart disease is shown in Figure 3.

Analysis:

In existing literature, SVM offers highest accuracy of 94.60% in 2012 as in Table 1. In many application areas, SVM shows good performance result. Attribute or features used by Parthiban and Srivatsa in 2012 are correctly responded by SVM. In 2015, Otoom et al. used SVM variant called SMO. It also uses FS technique to find best features. SVM responds to these features and offers the accuracy of 85.1% but it is comparatively low as in 2012. Training and testing set of both data sets are different, as well as, data types are different.

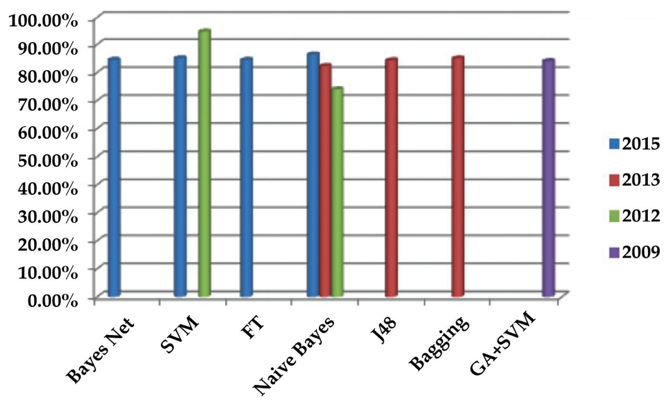


Figure 3: Machine learning algorithm's accuracy to detect heart disease.

Table 1: Comprehensive view of machine learning techniques for heart disease diagnosis

Machine Learning Techniques	Author	Year	Disease	Resources of Data Set	Tool	Accuracy
Bayes Net	Otoom et al.	2015	CAD (Coronary artery disease)	UCI	WEKA	84.5%
SVM						85.1%
FT						84.5%
Naive Bayes	Vembandasamy et al.	2015	Heart Disease	Diabetic Research Institute in Chennai	WEKA	86.419%
Naive Bayes	Chaurasia and Pal	2013	Heart Disease	UCI	WEKA	82.31%
J48						84.35%
Bagging						85.03%
SVM	Parthiban and Srivatsa	2012	Heart disease	Research institute in Chennai	WEKA	94.60%
Naive Bayes						74%
Hybrid Technique (GA + SVM)	Tan et al.	2009	Heart disease	UCI	LIBSVM and WEKA	84.07%

Advantages and Disadvantages of SVM:

Advantages: Construct correct classifiers and fewer over fitting, robust to noise.

Disadvantages: It is a binary classifier. For the classification of multi-class, it can use pair wise classification. Its Computational cost is high, so it runs slow [10] .

Diabetes Disease

Iyer et al. [11] has performed a work to predict diabetes disease by using decision tree and Naive Bayes. Diseases occur when production of insulin is insufficient or there is improper use of insulin. Data set used in this work is Pima Indian diabetes data set. Various tests were performed using WEKA data mining tool. In this data-set percentage split (70:30) predict better than cross validation. J48 shows 74.8698% and 76.9565% accuracy by using Cross Validation and Percentage Split Respectively. Naive Bayes presents 79.5652% correctness by using PS. Algorithms shows highest accuracy by utilizing percentage split test.

Meta learning algorithms for diabetes disease diagnosis has been discussed by Sen and Dash [12] . The employed data set is Pima Indians diabetes that is received from UCI Machine Learning laboratory. WEKA is used for analysis. CART, Adaboost, Logiboost and grading learning algorithms are used to predict that patient has diabetes or not. Experimental results are compared on the behalf of correct or incorrect classification. CART offers 78.646% accuracy. The Adaboost obtains 77.864% exactness. Logiboost offers the correctness of 77.479%. Grading has correct classification rate of 66.406%. CART offers highest accuracy of 78.646% and misclassification Rate of 21.354%, which is smaller as compared to other techniques.

An experimental work to predict diabetes disease is done by the Kumari and Chitra [13] . Machine learning technique that is used by the scientist in this experiment is SVM. RBF kernel is used in SVM for the purpose of classification. Pima Indian diabetes data set is provided by machine learning laboratory at University of California, Irvine. MATLAB 2010a are used to conduct experiment. SVM offers 78% accuracy.

Sarwar and Sharma [14] have suggested the work on Naive Bayes to predict diabetes Type-2. Diabetes disease has 3 types. First type is Type-1 diabetes, Type-2 diabetes is the second type and third type is gestational diabetes. Type-2 diabetes comes from the growth of Insulin resistance. Data set consists of 415 cases and for purpose of variety; data are gathered from dissimilar sectors of society in India. MATLAB with SQL server is used for development of model. 95% correct prediction is achieved by Naive Bayes.

Ephzibah [15] has constructed a model for diabetes diagnosis. Proposed model joins the GA and fuzzy logic. It is used for the selection of best subset of features and also for the enhancement of classification accuracy. For experiment, dataset is picked up from UCI Machine learning laboratory that has 8 attributes and 769 cases. MATLAB is used for implementation. By using genetic algorithm only three best features/attributes are selected. These three attributes are used by fuzzy logic classifier and provide 87% accuracy. Around 50% cost is less than the original cost. Table 2provides the Comprehensive view of Machine learning Techniques for diabetes disease diagnosis.

Analysis:

Naive Bayes based system is helpful for diagnosis of Diabetes disease. Naive Bayes offers highest accuracy of 95% in 2012. The results show that this system can do good prediction with minimum error and also this technique is important to diagnose diabetes disease. But in 2015, accuracy offered by Naive Bayes is low. It presents 79.5652% or 79.57% accuracy. This proposed model for detection of Diabetes disease would require more training data for creation and testing. Figure 4shows the Accuracy graph of Algorithms for the diagnosis of Diabetes disease according to time.

Advantages and Disadvantages of Naive Bayes:

Advantages: It enhances the classification performance by eliminating the unrelated features. Its performance is good. It takes less computational time.

Table 2: Comprehensive view of machine learning techniques for diabetes disease diagnosis

Machine Learning Techniques	Author	Year	Disease	Resource of Data Set	Tool	Accuracy
Naive Bayes	Iyer et al.	2015	Diabetes Disease	Pima Indian Diabetes dataset	WEKA	79.5652%
J48						76.9565%
CART	Sen and Dash	2014	Diabetes Disease	Pima Indian Diabetes dataset from UCI	WEKA	78.646%
Adaboost						77.864%
Logiboost						77.479%
Grading						66.406%
SVM	Kumari and Chitra	2013	Diabetes Disease	UCI	MATLAB 2010a	78%
Naive Bayes	Sarwar and Sharma	2012	Diabetes type-2	Different Sectors of Society in India	MATLAB with SQL Server	95%
GA + Fuzzy Logic	Ephzibah	2011	Diabetes disease	UCI	MATLAB	87%

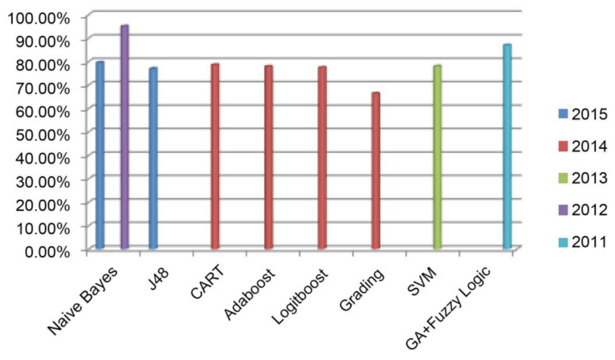


Figure 4: Accuracy of machine learning algorithms to detect diabetes disease.

Disadvantages: This algorithm needs large amount of data to attain good outcomes. It is lazy as they store entire the training examples [16] .

Liver Disease

Vijayarani and Dhayanand [17] predict the liver disease by using Support vector machine and Naive bayes Classification algorithms. ILPD data set is obtained from UCI. Data set comprises of 560 instances and 10 attributes. Comparison is made on the basis of accuracy and time execution. Naive bayes shows 61.28% correctness in 1670.00 ms. 79.66% accuracy is attained in 3210.00 ms by SVM. For implementation, MATLAB is used. SVM shows highest accuracy as compared to the Naive bayes for liver disease prediction. In terms of time execution, Naives bayes takes less time as compared to the SVM.

A study on intelligent techniques to classify the liver patients is performed by the Gulia et al. [18] . Used data set is picked up from UCI. WEKA data mining tool and five intelligent techniques J48, MLP, Random Forest, SVM and Bayesian Network classifiers are used in this experiment. In first step, all algorithms are applied on the original data set and get the percentage of correctness. In second step, feature selection method is applied on whole data-set to get the significant subset of liver patients and all these algorithms are used to test the subset of whole data-set. In third step they take comparison of outcomes before and after feature selection. After FS, algorithms provide highest accuracy as J48 presents 70.669% accuracy, 70.8405% exactness is achieved by the MLP algorithm, SVM provides 71.3551% accuracy, 71.8696% accuracy is offered by Random forest and Bayes Net shows 69.1252% accuracy.

Rajeswari and Reena [19] used the data mining algorithms of Naive Bayes, K star and FT tree to analyze the liver disease. Data set is taken from UCI that comprises of 345 instances and 7 attributes. 10 cross validation test are applied by using WEKA tool. Naive Bayes provide 96.52% Correctness in 0 sec. 97.10% accuracy is achieved by using FT tree in 0.2 sec. K star algorithm classify the instances about 83.47% accurately in 0 sec. On the basis of outcomes, highest classification accuracy is offered by FT tree on liver disease dataset as compared to other data mining algorithms. Table 3 presents the comprehensive view of algorithms for the detection of liver disease.

Analysis:

To diagnose liver disease, FT Tree Algorithm provides the highest result as compare to the other algorithms. When FT tree algorithm is applied on the dataset of liver disease, time taken for result or building the model is fast as compared to other algorithms. According to its attribute, it shows the improved performance. This algorithm fully classified the attributes and offers 97.10% correctness. From the results, this Algorithm plays an important role in determining enhanced classification accuracy of data set. Accuracy graph of algorithms are shown in Figure 5.

Advantages and Disadvantages of FT:

Advantage: Easy to interpret and understand; Fast prediction.

Disadvantage: Calculations are complex mainly if values are uncertain or if several outcomes are linked.

Dengue Disease

Tarmizi et al. [20] performed a work for Malaysia Dengue Outbreak Detection by using the Models of Data Mining.

Table 3: Comprehensive view of machine learning techniques for liver disease diagnosis

Machine Learning Techniques	Author	Year	Disease	Resource of Data Set	Tool	Accuracy
SVM	Vijayarani and Dhayanand	2015	Liver Disease	ILPD from UCI	MATLAB	79.66%
Naive Bayes						61.28%

J48	Gulia et al.	2014	Liver Disease	UCI	WEKA	70.669%
MLP						70.8405%
Random Forest						71.8696%
SVM						71.3551%
Bayesian Network						69.1252%
Naive Bayes	Rajeswari and Reena	2010	Liver Disease	UCI	WEKA	96.52%
K Star						83.47%
FT tree						97.10%

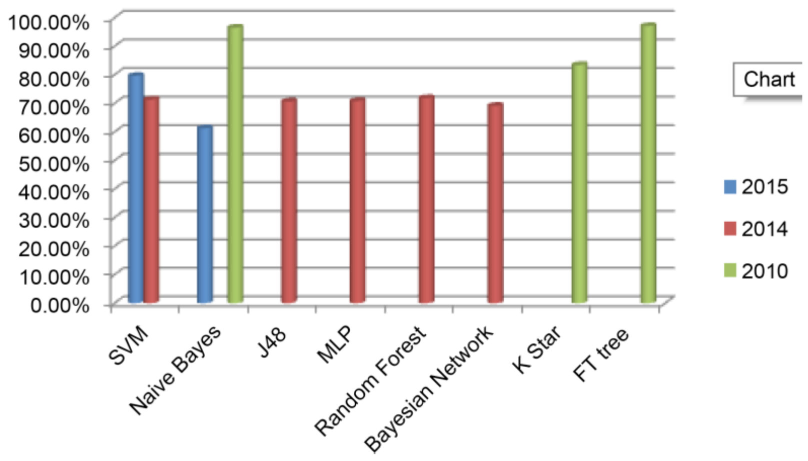


Figure 5: Accuracy of machine learning algorithms to detect liver disease.

Dengue is becoming a severe contagious disease. It creates trouble in those countries where weather is humid for example Thailand, Indonesia and Malaysia. Decision Tree (DT), Artificial Neural Network (ANN), and Rough Set Theory (RS) are the classification algorithms that are used in this study to predict dengue disease. Data set are taken from Public Health Department of Selangor State. WEKA data mining tool with two tests (10 Cross-fold Validation and Percentage split) is used. By using 10-Cross fold validation DT offers 99.95% accuracy, ANN presents 99.98% of Correctness and RS shows 100% accuracy. After using PS, Both Decision tree and Artificial Neural Network gives 99.92% of correctness. RS achieves 99.72% accuracy.

Fathima and Manimeglai [21] performed a work to predict Arbovirus-Dengue disease. Data mining algorithm that are used by these researchers are Support Vector Machine. Data set for analysis is obtained from King

Institute of Preventive Medicine and surveys of many hospitals and laboratories of Chennai and Tirunelveli from India. It contains 29 attributes and 5000 samples. Data is examined by R project version 2.12.2. Accuracy that is achieved by SVM is 0.9042.

Ibrahim et al. [22] suggested a system in which Artificial neural network is used for forecasting the defervescence day of fever in patients of dengue disease. Only clinical signs and symptoms are used by the proposed system for detection. The data are gathered from 252 hospitalized patients, in which 4 patients are having DF (Dengue fever) and 248 patients are having DHF (dengue hemorrhagic fever). MATLAB's neural network toolbox is used. Algorithm of Multilayer feed-forward neural network (MFNN) is used in this experiment. Day of defervescence of fever is accurately predicted by MFNN in DF and DHF with 90% correctness.

Figure 6 shows the accuracy graph of all algorithms for the diagnosis of Dengue disease.

Analysis:

Different Machine learning techniques are used to diagnose dengue disease. Dengue disease is one of the serious contagious diseases. As in Table 4, for detection of dengue disease, RS theory shows the highest result as compared to the other algorithms.

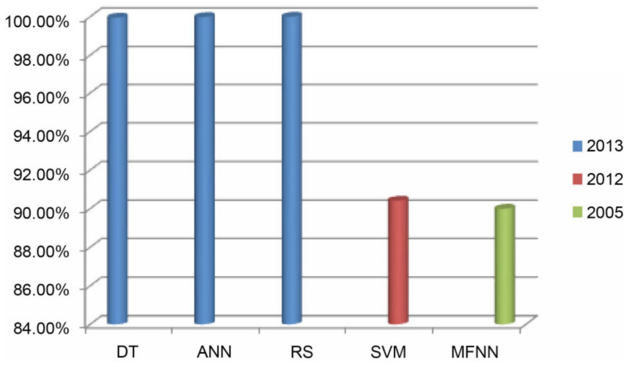


Figure 6: Accuracy of machine learning algorithms for dengue disease.

Table 4: Analysis of machine learning techniques for dengue disease detection

Machine Learning Techniques	Author	Year	Disease	Resource of Data Set	Tool	Accuracy
DT	Tarmizi et al.	2013	Dengue Disease	Public Health Department of Selangor State	WEKA	99.95%
ANN						99.98%
RS						100%
SVM	Fathima and Manimeglai	2012	Arbovirus-Dengue disease	King Institute of Preventive Medicine and surveys of many hospitals and laboratories of Chennai and Tirunelveli from India	R project Version 2.12.2	90.42%
MFNN	Ibrahim et al.	2005	Dengue disease	From 252 hospitalized patients	MATLAB neural network Tool box	90%

In 2005 and 2012, researchers used different algorithms but did not attain highest result and improvements. In 2013, accuracy is improved by using RS. It is capable to manage uncertainty, noise and missing data. For the purpose of classification, Developed RS classifier is based on the Rough set theory. Selection of attribute empowers the classifier to surpass the other models. RS is a promising rule based method that offers meaningful information. RS is also best from neural network in term of time. NN takes much time to build model. DT is complex as well as costly algorithm. RS does not need any initial and additional information about data but Decision tree needs information.

Advantages and Disadvantages of RS:

Advantages: It is very easy to understand and provides direct understanding of attained result. It evaluates data significance. It is appropriate for both qualitative and quantitative data. It discovers the hidden patterns. It also finds minimal set of data. It can find relationship that cannot be identified by statistical methods.

Disadvantages: It has not so many limitations still it is not widely used.

Hepatitis Disease

Ba-Alwi and Hintaya [23] suggested a comparative analysis. Data mining algorithms that are used for hepatitis disease diagnosis are Naive Bayes, Naive Bayes updatable, FT Tree, K Star, J48, LMT, and NN. Hepatitis disease data set was taken from UCI Machine Learning repository. Classification results are measured in terms of accuracy and time. Comparative Analysis is taken by using neural connections and WEKA: data mining tool. Results that are taken by using neural connection are low than the algorithms used in WEKA. In this Analysis of Hepatitis disease diagnosis, second technique that is used is rough set theory, by using WEKA. Performance of Rough set procedure is better than NN specially in case of medical data analysis. Naive Bayes gives the accuracy of 96.52% in 0 sec. 84% Accuracy is attained by the Naive Bayes Updateable algorithm in 0 sec. In 0.2 sec FT Tree presents the accuracy of 87.10%. K star offers 83.47% Correctness. Time taken for K star algorithm is 0 sec. Correctness of 83% is achieved by J48 and time that J48 takes to classify is 0.03 sec. LMT provides 83.6% accuracy 0.6 sec. Neural network shows 70.41% of correctness. Naive Bayes is best classification algorithm used in the rough set technique. It offers high accuracy in minimum time.

Karlik [24] shows a comparative analysis of Naive Bayes and back propagation classifiers to diagnose hepatitis disease. Key advantage of using these classifiers is that they require small amount of data for categorization. Types of hepatitis are “A, B, C, D and E”. These are generated by different viruses of hepatitis. Rapid Miner open source software is used in this analysis. Hepatitis data set is taken from UCI. Data set include 20 features and 155 instances. 15 attributes are used in this experiment. Naive Bayes classifier gives 97% accuracy. Three-layered feed- forward NN are used and trained with Back propagation algorithm 155 instances are used for training. Correctness of 98% is attained.

Sathyadevi [25] employed C4.5, ID3 and CART algorithms for diagnosing the disease of hepatitis. This study uses the UCI hepatitis patient data set. WEKA, tool is used in this analysis. CART has offered great performance handling of missing values. So, CART algorithm shows a highest classification accuracy of 83.2%. ID3 Algorithm offers 64.8% of accuracy. 71.4% is attained by C4.5 algorithm. Binary decision tree (DT) that is generated by CART algorithm has only two or no child. DT that is formed by the C4.5 and ID3 can have two or more children. CART algorithm performs well in terms of Accuracy and time complexity.

Analysis:

Many algorithms have been used for diagnosis of different diseases. Table 5 gives the comprehensive view. For the detection of Hepatitis disease, Feed forward neural network with back propagation shows highest accuracy of 98%. Because in this model, three layered feed forward neural network is trained with error back propagation algorithm. Back propagation training with the rule of delta learning is an iterative gradient algorithm planned to lessen the RMSE "root mean square error" between the real output of a multilayered feed-forward neural networks and a desired output. Every layer is connected to preceding layer and having no other connection. Second best result is offered by Naive Bayes. But in terms of time to build model, Naive Bayes runs fast as compare to neural network. Figurative approach for the detection of hepatitis is shown in Figure 7.

Advantages and Disadvantages of NN:

Advantages: Adaptive Learning, Self-Organization, Real Time Operation Fault Tolerance via Redundant Information Coding.

Disadvantages: Less over fitting needs great computational effort. Sample Size must be large. It's time consuming. Engineering Judgment does not develop the relations between input and output variables so that the model behaves like a black box [26] .

DISCUSSIONS AND ANALYSIS OF MACHINE LEARNING TECHNIQUES

For diagnosis of Heart, Diabetes, Liver, Dengue and Hepatitis diseases, several machine-learning algorithms perform very well. From existing literature, it is observed that Naive Bayes Algorithm and SVM are widely used algorithms for detection of diseases.

Table 5: Comprehensive view of machine learning techniques for hepatitis disease

Machine Learning Techniques	Author	Year	Disease	Resource of Data Set	Tool	Accuracy
Naive Bayes	Ba-Alwi and Hintaya,	2013	Hepatitis Disease	UCI	WEKA	96.52%
Naive Bayes updateable						84%
FT						87.10%
K Star						83.47%
J48						83%
LMT						83.6%
NN						70.41%

Naive Bayes	Karlik	2011	Hepatitis Disease	UCI	Rapid Miner	97%
Feed forward NN with Back propagation						98%
C4.5	Sathyadevi	2011	Hepatitis Disease	UCI	WEKA	71.4%
ID3						64.8%
CART						83.2%

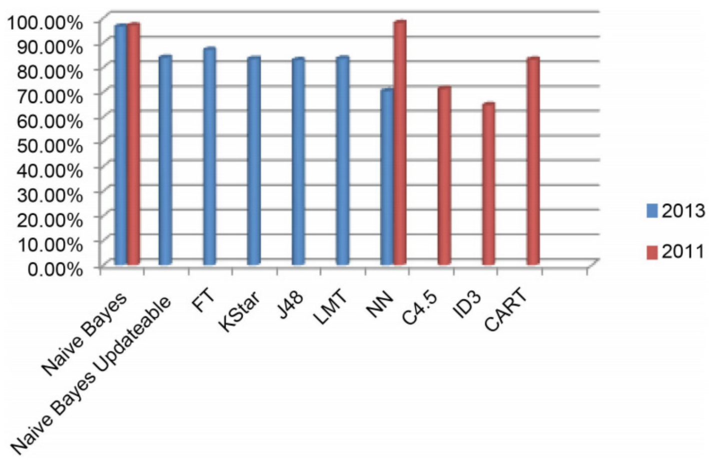


Figure 7: Machine learning algorithm's accuracy to detect hepatitis disease.

Both algorithms offer the better accuracy as compare to other algorithms. Artificial Neural network is also very useful for prediction. It also shows the maximum output but it takes more time as compared to other algorithms. Trees algorithm are also used but they did not attain wide acceptance due to its complexity. They also shows enhanced accuracy when it responded correctly to the attributes of data set. RS theory is not widely used but it presents maximum output.

CONCLUSION

Statistical models for estimation that are not capable to produce good performance results have flooded the assessment area. Statistical models are unsuccessful to hold categorical data, deal with missing values and large data points. All these reasons arise the importance of MLT. ML plays a vital role in many applications, e.g. image detection, data mining, natural language processing, and disease diagnostics. In all these domains, ML offers possible solutions. This paper provides the survey of different machine learning techniques for diagnosis of different diseases such as heart disease, diabetes disease, liver disease, dengue and hepatitis disease. Many algorithms have shown good results because they identify the attribute accurately. From previous study, it is observed that for the detection of heart disease, SVM provides improved accuracy of 94.60%. Diabetes disease is accurately diagnosed by Naive Bayes. It offers the highest classification accuracy of 95%. FT provides 97.10% of correctness for the liver disease diagnosis. For dengue disease detection, 100% accuracy is achieved by RS theory. The feed forward neural network correctly classifies hepatitis disease as it provides 98% accuracy. Survey highlights the advantages and disadvantages of these algorithms. Improvement graphs of machine learning algorithms for prediction of diseases are presented in detail. From analysis, it can be clearly observed that these algorithms provide enhanced accuracy on different diseases. This survey paper also provides a suite of tools that are developed in community of AI. These tools are very useful for the analysis of such problems and also provide opportunity for the improved decision making process.

REFERENCES

1. Marshland, S. (2009) Machine Learning an Algorithmic Perspective. CRC Press, New Zealand, 6-7.
2. Sharma, P. and Kaur, M. (2013) Classification in Pattern Recognition: A Review. International Journal of Advanced Research in Computer Science and Software Engineering, 3, 298.
3. Rambhajani, M., Deepanker, W. and Pathak, N. (2015) A Survey on Implementation of Machine Learning Techniques for Dermatology Diseases Classification. International Journal of Advances in Engineering & Technology, 8, 194-195.
4. Kononenko, I. (2001) Machine Learning for Medical Diagnosis: History, State of the Art and Perspective. Journal of Artificial Intelligence in Medicine, 1, 89-109.
5. Otoom, A.F., Abdallah, E.E., Kilani, Y., Kefaye, A. and Ashour, M. (2015) Effective Diagnosis and Monitoring of Heart Disease. International Journal of Software Engineering and Its Applications. 9, 143-156.
6. Vembandasamy, K., Sasipriya, R. and Deepa, E. (2015) Heart Diseases Detection Using Naive Bayes Algorithm. IJISET-International Journal of Innovative Science, Engineering & Technology, 2, 441-444.
7. Chaurasia, V. and Pal, S. (2013) Data Mining Approach to Detect Heart Disease. International Journal of Advanced Computer Science and Information Technology (IJACSIT), 2, 56-66.
8. Parthiban, G. and Srivatsa, S.K. (2012) Applying Machine Learning Methods in Diagnosing Heart Disease for Diabetic Patients. International Journal of Applied Information Systems (IJAIS), 3, 25-30.
9. Tan, K.C., Teoh, E.J., Yu, Q. and Goh, K.C. (2009) A Hybrid Evolutionary Algorithm for Attribute Selection in Data Mining. Journal of Expert System with Applications, 36, 8616-8630. https://doi.org/10.1016/j.eswa.2008.10.013
10. Karamizadeh, S., Abdullah, S.M., Halimi, M., Shayan, J. and Rajabi, M.J. (2014) Advantage and Drawback of Support Vector Machine Functionality. 2014 IEEE International Conference on Computer, Communication and Control Technology (I4CT), Langkawi, 2-4 September 2014, 64-65. https://doi.org/10.1109/i4ct.2014.6914146
11. Iyer, A., Jeyalatha, S. and Sumbaly, R. (2015) Diagnosis of Diabetes

Using Classification Mining Techniques. International Journal of Data Mining & Knowledge Management Process (IJDKP), 5, 1-14. https://doi.org/10.5121/ijdkp.2015.5101

12. Sen, S.K. and Dash, S. (2014) Application of Meta Learning Algorithms for the Prediction of Diabetes Disease. International Journal of Advance Research in Computer Science and Management Studies, 2, 396-401.
13. Kumari, V.A. and Chitra, R. (2013) Classification of Diabetes Disease Using Support Vector Machine. International Journal of Engineering Research and Applications (IJERA), 3, 1797-1801.
14. Sarwar, A. and Sharma, V. (2012) Intelligent Naive Bayes Approach to Diagnose Diabetes Type-2. Special Issue of International Journal of Computer Applications (0975-8887) on Issues and Challenges in Networking, Intelligence and Computing Technologies-ICNICT 2012, 3, 14-16.
15. Ephzibah, E.P. (2011) Cost Effective Approach on Feature Selection using Genetic Algorithms and Fuzzy Logic for Diabetes Diagnosis. International Journal on Soft Computing (IJSC), 2, 1-10. https://doi.org/10.5121/ijsc.2011.2101
16. Archana, S. and DR Elangovan, K. (2014) Survey of Classification Techniques in Data Mining. International Journal of Computer Science and Mobile Applications, 2, 65-71
17. Vijayarani, S. and Dhayanand, S. (2015) Liver Disease Prediction using SVM and Naive Bayes Algorithms. International Journal of Science, Engineering and Technology Research (IJSETR), 4, 816-820.
18. Gulia, A., Vohra, R. and Rani, P. (2014) Liver Patient Classification Using Intelligent Techniques. (IJCSIT) International Journal of Computer Science and Information Technologies, 5, 5110-5115.
19. Rajeswari, P. and Reena,G.S. (2010) Analysis of Liver Disorder Using Data Mining Algorithm. Global Journal of Computer Science and Technology, 10, 48-52.
20. Tarmizi, N.D.A., Jamaluddin, F., Abu Bakar, A., Othman, Z.A., Zainudin, S. and Hamdan, A.R. (2013) Malaysia Dengue Outbreak Detection Using Data Mining Models. Journal of Next Generation Information Technology (JNIT), 4, 96-107.
21. Fathima, A.S. and Manimeglai, D. (2012) Predictive Analysis for the Arbovirus-Dengue using SVM Classification. International Journal of Engineering and Technology, 2, 521-527.

22. Ibrahim, F., Taib, M.N., Abas, W.A.B.W., Guan, C.C. and Sulaiman, S. (2005) A Novel Dengue Fever (DF) and Dengue Haemorrhagic Fever (DHF) Analysis Using Artificial Neural Network (ANN). Computer Methods and Programs in Biomedicine, 79, 273-281. https://doi.org/10.1016/j.cmpb.2005.04.002
23. Ba-Alwi, F.M. and Hintaya, H.M. (2013) Comparative Study for Analysis the Prognostic in Hepatitis Data: Data Mining Approach. International Journal of Scientific & Engineering Research, 4, 680-685.
24. Karlik, B. (2011) Hepatitis Disease Diagnosis Using Back Propagation and the Naive Bayes Classifiers. Journal of Science and Technology, 1, 49-62.
25. Sathyadevi, G. (2011) Application of CART Algorithm in Hepatitis Disease Diagnosis. IEEE International Conference on Recent Trends in Information Technology (ICRTIT), MIT, Anna University, Chennai, 3-5 June 2011, 1283-1287.
26. Singh, Y., Bhatia, P.K., and Sangwan, O. (2007) A Review of Studies on Machine Learning Techniques. International Journal of Computer Science and Security, 1, 70-84.

CHAPTER
5

BANKRUPTCY PREDICTION USING MACHINE LEARNING

Nanxi Wang

Shanghai Starriver Bilingual School, Shanghai, China

ABSTRACT

With improved machine learning models, studies on bankruptcy prediction show improved accuracy. This paper proposes three relatively newly-developed methods for predicting bankruptcy based on real-life data. The result shows among the methods (support vector machine, neural network with dropout, autoencoder), neural network with added layers with dropout has the highest accuracy. And a comparison with the former methods (logistic regression, genetic algorithm, inductive learning) shows higher accuracy.

Citation: Wang, N. (2017), "Bankruptcy Prediction Using Machine Learning". *Journal of Mathematical Finance*, **7**, 908-918. doi: 10.4236/jmf.2017.74049.

Keywords: Support Vector Machine, Autoencoder, Neural Network, Bankruptcy, Machine Learning

INTRODUCTION

Machine learning is a subfield of computer science. It allows computers to build analytical models of data and find hidden insights automatically, without being unequivocally coded. It has been applied to a variety of aspects in modern society, ranging from DNA sequences classification, credit card fraud detection, robot locomotion, to natural language processing. It can be used to solve many types of tasks such as classification. Bankruptcy prediction is a typical example of classification problems.

Machine learning was born from pattern recognition. Earlier works of the same topic (machine learning in bankruptcy) use models including logistic regression, genetic algorithm, and inductive learning.

Logistic regression is a statistical method allowing researchers to build predictive function based on a sample. This model is best used for understanding how several independent variables influence a single outcome variable [1] . Though useful in some ways, logistic regression is also limited.

Genetic algorithm is based on natural selection and evolution. It can be used to extract rules in propositional and first-order logic, and to choose the appropriate sets of if-then rules for complicated classification problems [2] .

Inductive learning's main category is decision tree algorithm. It identifies training data or earlier knowledge patterns and then extracts generalized rules which are then used in problem solving [2] .

To see if the accuracy of bankruptcy prediction can be further improved, we propose three latest models—support vector machine (SVM), neural network, and autoencoder.

Support vector machine is a supervised learning method which is especially effective in cases of high dimensions, and is memory efficient because it uses a subset of training points in the decision function. Also, it specifies kernel functions according to the decision function [3] . Its nice math property guarantees a simple convex optimization problem to converge to a single global problem.

Neural networks, unlike conventional computers, are expressive models that learn by examples. They contain multiple hidden layers, thus are capable of learning very complicated relationships between inputs and outputs. And they operate significantly faster than conventional techniques. However, due to limited training data, overfitting will affect the ultimate accuracy. To prevent this, a technique called dropout—temporarily and randomly removes units (hidden and visible)—to the neural network [4] .

Autoencoder, also known as Diabolo network, is an unsupervised learning algorithm that sets the target values to be equal to the inputs. By doing this, it suppresses the computation of representing a few functions, which improves accuracy. Also, the amount of training data required to learn these functions is reduced [5] .

This paper is structured as follows. Section 2 describes the motivation for this idea. Section 3 describes relevant previous work. Section 4 formally describes the three models. In Section 5 we present our experimental results where we do a parallel comparison within the three models we choose and a longitudinal comparison with the three older models. Section 6 is the conclusion. Section 7 is the reference.

MOTIVATION

The three models we choose (SVM, neural network, autoencoder) are relatively newly-developed but have already been applied to many fields.

SVM has been used successfully in many real-world problems such as text categorization, object tracking, and bioinformatics (Protein classification, Cancer classification). Text categorization is especially helpful in daily life—web searching and email filtering provide huge convenience and work efficiency.

Neural networks learn by examples instead of algorithms, thus, they have been widely applied to problems where it is hard or impossible to apply algorithmic methods [6] . For instance, finger print recognition is an exciting application. People can now use their unique fingerprints as keys to unlock their phones and payment accounts, free from the troubling, long passwords.

Autoencoders are especially successful in solving difficult tasks like natural language processing (NLP). They have been used to solve the previous seemingly intractable problems in NLP, including word embeddings, machine translation, document clustering, sentiment analysis, and paraphrase detection.

However, the usage of the three models in economics or finance is comparatively hard to find. So, we aim to find out if they still work well in economical field by running them with real-life data in a predicting bankruptcy task.

Another motivation is finding out if the accuracy of this particular problem (bankruptcy prediction) can be improved after reading previous works—The discovery of experts' decision rules from qualitative bankruptcy data using genetic algorithms [2] , and Predicting Bankruptcy with Robust Logistic Regression [1] —which uses older models. Thus, a comparison of the models and results is included in this paper.

RELATED WORK

Machine learning enables computers to find insights from data automatically. The idea of using machine learning to predict bankruptcy has previously been used in the context of Predicting Bankruptcy with Robust Logistic Regression by Richard P. Hauser and David Booth [1] . This paper uses robust logistic regression which finds the maximum trimmed correlation between the samples remained after removing the overly large samples and the estimated model using logistic regression [1] . This model has its limitation. The value of this technique relies heavily on researchers' abilities to include the correct independent variables. In other words, if researchers fail to identify all the relevant independent variables, logistic regression will have little predictive value [7] . Its overall accuracy is 75.69% in the training set and 69.44% in testing set.

Another work, the discovery of experts' decision rules from qualitative bankruptcy data using genetic algorithms, in 2003 by Myoung-Jong Kim and Ingoo Han uses the same dataset as we do. They apply older models—inductive learning algorithms (decision tree), genetic algorithms, and neural networks without dropout. Since the length of genomes in GA is fixed, a given problem cannot easily be encoded. And GA gives no guarantee of finding the global maxima. The problem of inductive learning is with the one-step-ahead node splitting without backtracking, which may generate a suboptimal tree. Also, decision trees can be unstable because small variations in the data might result in a completely different tree being generated [3] . And the absence of dropout in the neural network model increases the possibility of overfitting which affects accuracy. The overall accuracies are 89.7%, 94.0%, and 90.3% respectively.

The models we choose either contain a newly developed technique, like dropout, or completely new models that have hardly been utilized in bankruptcy prediction.

MODEL DESCRIPTION

This section describes the proposed three models.

Support Vector Machine

Specifically, we use support vector classify (SVC), a subcategory of SVM, in this task. It constructs a hyper-plane, as shown in Figure 1, in a high dimensional space which is used for classification. Generally, a good separation represented by the solid line in Figure 1 means the distance(the space between the dotted lines) to the nearest training data points (the red and blue dots) of any class (represented by the color red and blue) is the largest. This is also known as functional margin [3] .

With training vectors in two classes and a vector,

$$x_i \in \mathbb{R}^p, i = 1, \cdots, n, \ y \in \{1, -1\}^n$$

respectively, SVM aims at solving the problem:

$$\min_{\omega, b, \zeta} \frac{1}{2} \omega^{\mathrm{T}} \omega + C \sum_{i=1}^{n} \zeta_i$$

subject to

$$y_i \left(\omega^{\mathrm{T}} \phi(x_i) + b \right) \geq 1 - \zeta_i$$

Its dual is

$$\min_{\alpha} \frac{1}{2} \alpha^{\mathrm{T}} Q \alpha - e^{\mathrm{T}} \alpha$$

subject to

$$y^{\mathrm{T}} \alpha = 0, \ 0 \leq \alpha_i \leq C, i = 1, \cdots, n$$

where e is a common vector, C>0 is upper bound, Q is n by n positive semidefinite matrix, $Q_{ij} \equiv y_i y_j k(x_i \cdot x_j)$, and $K(x_i, x_j) = \phi(x_i)^{\mathrm{T}} \phi(x_j)$ is the kernel.

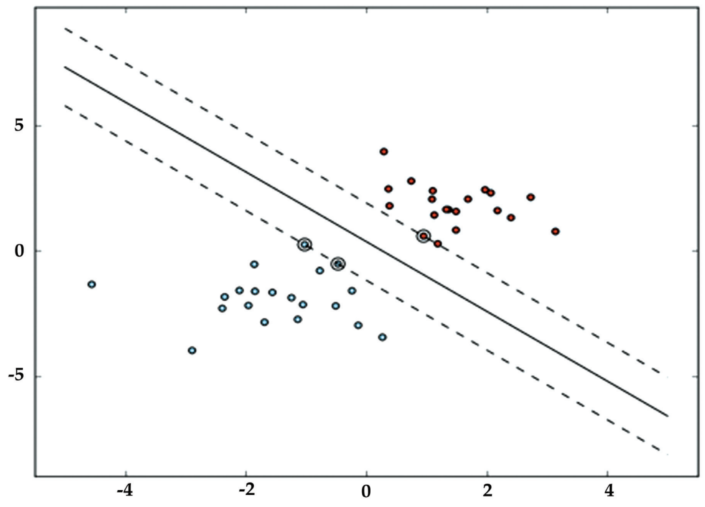


Figure 1: SVM model [3] .

Here the function implicitly maps the training vectors into a higher dimensional space.

The decision function is:

$$\operatorname{sgn}\left(\sum_{i=1}^{n} y_i \alpha_i K\left(x_i, x\right) + \rho\right) \quad [3]$$

Neural Network with Dropout

Neural networks' inputs are modelled as layers of neurons. Its structure is shown in the following figure.

As shown in Figure 1, the formal neuron uses n inputs $x_1,x_2,\cdots,x_n$ to classify the signals coming from dendrites, and are then synoptically weighted correspondingly with $w_1,w_2,\cdots,w_n$ that measure their permeabilities. Then, the excitation level of the neuron is calculated as the weighted sum of input values:

$$\xi = \sum_{i=1}^{n} w_i x_i$$

f in Figure 2 represents activation function.

When the value of excitation level x reaches the threshold h, the output y (state) of the neuron is induced. This simulates the electric impulse generated by axon [8] .

Dropout is a technique that further improves neural network's accuracy. In Figure 3, let L be the number of hidden layers, l∈{1,⋯,L}l∈{1,⋯,L} the hidden layers of the neural network, z(l)z(l) and y(l)y(l) the vectors of inputs and outputs of layer ll , respectively. W(l)W(l) and b(l)b(l) are the weights and biases at layer ll . For l∈{0,⋯,L−1}l∈{0,⋯,L−1} and any hidden unit i, the network then can be described as:

$$z^{(l+1)} = w^{(l+1)} y^{l} + b^{(l+1)} ,^{\text{iii}}$$

$$y^{(l+1)} = f\left(z^{(l+1)}\right) ,^{\text{ii}}$$

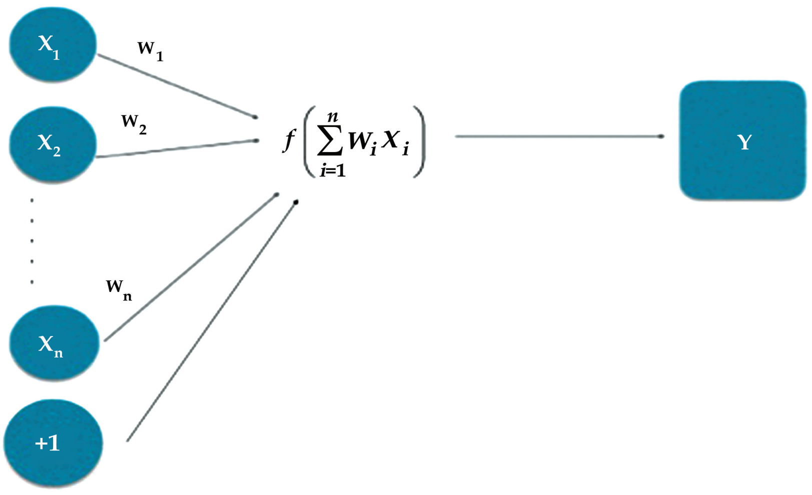


Figure 2: Neural network model.

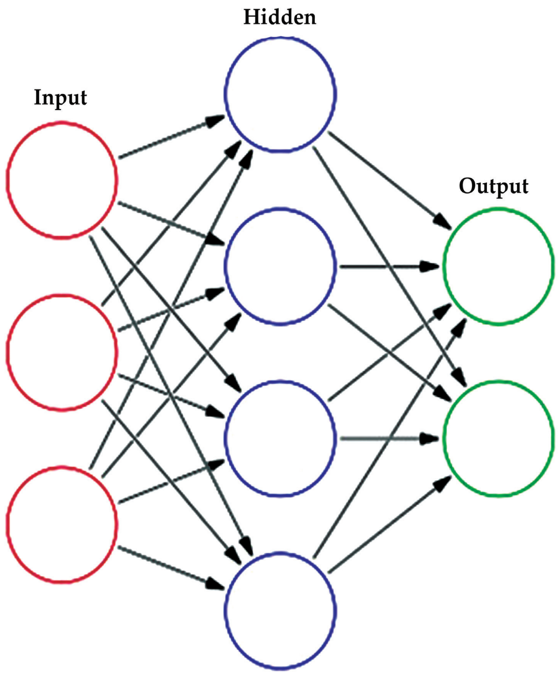


Figure 3: Artificial neural network.

where f is any activation function.

With dropout, the feed-forward operation becomes:

$r^{(l)}$-Bernoulli(p), j

$$y^{(l)} = r^{(l)} y^{(l)} ,$$

$$z^{(l+1)} = w^{(l+1)} y^{l} + b^{(l+1)} ,^{iii} \qquad [4] .$$

Autoencoder

Consider an n/p/n autoencoder.

In Figure 4, let F and G denote sets, n and p be positive integers where $0 < p < n$, and B be a class of functions from F^n to G^p.

Define $X=\{x_1,\cdots,x_m\}$ as a set of training vectors in F^n. When there are external targets, let$Y=\{y_1,\cdots,y_m\}$ denote the corresponding set of target vectors in F^n. And Δ is a distortion function (e.g. Lp norm, Hamming distance) defined over F^n.

For any A ∈ A and B∈B, the input vector x ∈ F^n becomes output vector A ∘ B(x) ∈ F^n through the autoencoder. The goal is to find A ∈ A and B ∈ B that minimize the overall distortion function:

$$\min\ E(A,B) = \min E(x_t) = \min \Delta A \circ B(x_t), x_t \qquad [10].$$

Decision Tree

Given training vectors $x_i \in R^n,\ i = 1,\cdots,l$ and a label vector $y \in R^l$, a decision tree groups the sample according to the same labels.

Let Q represents the data at node m. The tree partitions the data θ=(j,t_m)

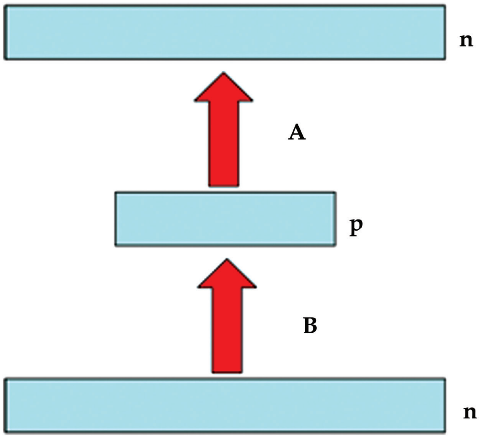


Figure 4: An n/p/n Autoencoder Architecture [Pierre Baldi, 2012].

(feature j and threshold t_m) into $Q_{\text{left}}(\theta)$ and $Q_{\text{right}}(\theta)$ subsets:

$$Q_{\text{left}}(\theta) = (x,y) \mid x_j \le t_m$$
$$Q_{\text{right}}(\theta) = Q \setminus Q_{\text{left}}(\theta)$$

The impurity function H() is used to calculate the impurity at m, the choice of which depends on the task being solved (classification or regression)

$$G(Q,\theta) = \frac{n_{\text{left}}}{N_m} H(Q_{\text{left}}(\theta)) + \frac{n_{\text{right}}}{N_m} H(Q_{\text{right}}(\theta))$$

Choose the parameters that minimises the impurity

$$\theta^* = \arg\min_\theta G(Q,\theta)$$

Then recur for subsets $Q_{\text{left}}\left(\theta^*\right)$ and $Q_{\text{right}}\left(\theta^*\right)$ until reaching the maximum possible depth, $N_m < \min_{\text{samples}}$ or $N_m = 1$ [3] .

EXPERIMENTAL RESULT

The data we used shown in Table 1, called Qualitative Bankruptcy database, is created by Martin. A, Uthayakumar. j, and Nadarajan. m in February 2014 [10] . The attributes include industrial risk, management risk, financial flexibility, credibility, competitiveness, and operating risk.

Parallel Comparison

SVM (Linear Kernel)

As shown in Table 2, the accuracy increases when truncate increases in a SVM model.

Neural Network (Activation = Softmax, Num_Classes = 2, Optimiser = Adam, Loss = Categorical _Crossentropy, Metrics = Accuracy)

As shown in Table 3, when other things in the model hold the same, dropout rate of 0.5 yields the highest accuracy.

Table 1: Dataset Description

Data set	Dimensionality	Instances	Training Set	Test Set	Validation
Bankruptcy	6 times1	250	80%	10%	10%

Table 2: Accuracy of Neural Network Model with Truncate 50 or 100

variation	accuracy
truncate = 50	0.9899
truncate = 100	0.9933

Table 3: Accuracy of Neural Network Model with and without Dropout

variation	accuracy
without dropout	0.9867 with loss 0.0462

with dropout (dropout rate = 0.1)	0.9867 with loss 0.0292
with dropout (dropout rate = 0.3)	0.9933 with loss 0.0300
with dropout (dropout rate = 0.4)	0.9933 with loss 0.0401
with dropout (dropout rate = 0.5)	0.9933 with loss 0.0278
with dropout (dropout rate = 0.7)	0.9933 with loss 0.0428
with dropout (dropout rate = 0.8)	0.9867 with loss 0.0318

As shown in Table 4 and Table 5, we can conclude that adding layers increases accuracy. Figure 5and Figure 6 depict Table 5.

Autoencoder (Encoding_Dim = 2, Activation = "Relu", Optimizer = "Adam", Lose = "Mse")

As shown in Table 6, autoencoder with decision tree yields higher accuracy.

Longitudinal Comparison

As shown in Table 7, neural network with truncate = 100 with added layers with dropout has the highest accuracy. And all the new models have higher accuracy than the old ones.

CONCLUSIONS

Support vector machine, neural network with dropout, and autoencoder are three relatively new models applied in bankruptcy prediction problems. Their accuracies outperform those of the three older models (robust logistic regression, inductive learning algorithms, genetic algorithms). The improved aspects include the control for overfitting, the improved probability of finding the global maxima, and the ability to handle large feature spaces. This paper compared and concluded the progress of machine leaning models regarding bankruptcy prediction, and checked to see the performance of relatively new models in the context of bankruptcy prediction that have rarely been applied in that field.

However, the three models also have drawbacks. SVM does not directly give probability estimates, but uses an expensive five-fold cross-validation instead.

Table 4: Accuracy of Neural Network Model with Two, Three, and Four Layer

variation	accuracy
two layer with dropout (dropout rate = 0.5)	0.9933 with loss 0.0278
three layer (added layer with dense 200) with dropout (dropout rate = 0.5)	0.9933 with loss 0.0221
four layer (added layer with dense 16) with dropout (dropout rate = 0.5)	1.0000 with loss 0.0004

Table 5: Accuracy of Neural Network Model with Truncate 50 or 100 and With Four Layers

variation	accuracy
truncate = 50 with four layers (added layer dense 16,200) with dropout rate 0.5	0.9950 with loss 0.0389
truncate = 100 with four layers (added layer dense 16,200) with dropout rate 0.5	1.0000 with loss 0.0004

Table 6: Accuracy of Neural Network Model with SVM or With Decision Tree

variation	accuracy
with SVM	0.9867
with decision tree	0.9933

Table 7: Accuracy of Neural Network Model with Different models

model	accuracy
Robust logistic regression	0.6944
inductive learning algorithms (decision tree)	0.897
genetic algorithms	0.94

neural networks without dropout	0.903
SVM truncate = 100	0.9933
Truncate = 100 with four layers (added layer dense 16,200) with dropout rate 0.5	1.0000 with loss 0.0004
autoencoder (with decision tree)	0.9933

Also, if the data sample is not big enough, especially when outnumbered by the number of features, SVM is likely to give bad performance [4] . With dropout, the time to train the neural network will be 2 to 3 times longer than training a standard neural network. An autoencoder captures as much information as possible, not necessarily the relevant information. And this can be a problem when the most relevant information only makes up a small percent of the input.

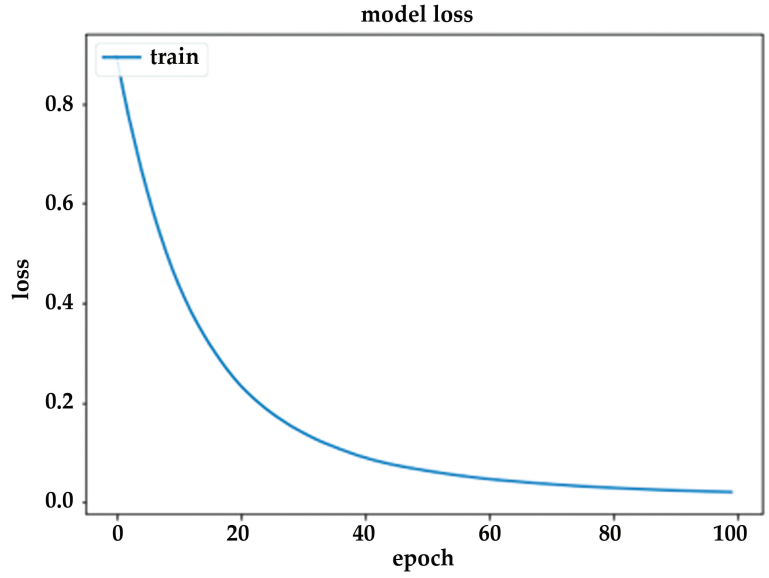


Figure 5: Neural network-loss.

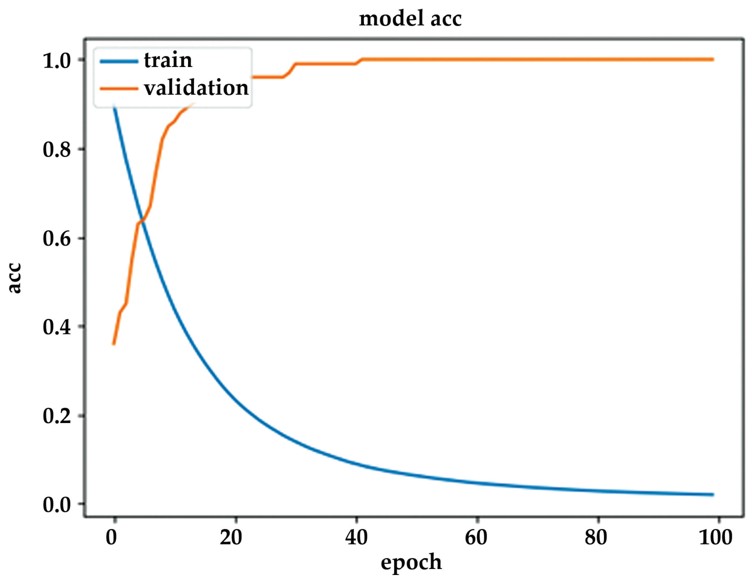


Figure 6: Neural network-accuracy.

The solutions to overcome these drawbacks are yet to be found.

REFERENCES

1. Hauser, R.P. and Booth, D. (2011) Predicting Bankruptcy with Robust Logistic Regression. Journal of Data Science, 9, 565-584.
2. Kim, M.-J. and Han, I. (2003) The Discovery of Experts' Decision Ruels from Qualitative Bankruptcy Data Using Genetic Algorithms. Expert Systems with Application, 25, 637-646,
3. Pedregosa, et al. (2011) Scikit-Learn: Machine Learning in Python. Journal of Machine Learning Research, 12, 2825-2830.
4. Sirvastava, N., et al. (2014) Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 15, 1929-1958.
5. Dev, D. (2017) Deep Learning with Hadoop. Packet Publishing, Birmingham, 52.
6. Nielsen, F. (2001) Neural Networks—Algorithms and Applications. https://www.mendeley.com/research-papers/neural-networks-algorithms-applications-5/
7. Robinson, N. (n.d.) The Disadvantages of Logistic Regression. http://classroom.synonym.com/disadvantages-logistic-regression-8574447.html
8. Sima, J. (1998) Introduction to Neural Networks. Technical Report No. 755.
9. Baldi, P. (2012) Autoencoders, Unsupervised Learning, and Deep Architectures. Journal of Machine Learning Research, 27, 37-50.
10. Martin, A., Uthayakumar, J. and Nadarajan, M. (2014) Qualitative Bankruptcy Data Set, UCI. https://archive.ics.uci.edu/ml/datasets/qualitative_bankruptcy

CHAPTER
6

PREDICTION OF SOLAR IRRADIATION USING QUANTUM SUPPORT VECTOR MACHINE LEARNING ALGORITHM

Makhamisa Senekane[1], Benedict Molibeli Taele[2]

[1]Faculty of Computing, Botho University-Maseru Campus, Maseru, Lesotho

[2]Department of Physics and Electronics, National University of Lesotho, Roma, Lesotho

ABSTRACT

Classical machine learning, which is at the intersection of artificial intelligence and statistics, investigates and formulates algorithms which can be used to discover patterns in the given data and also make some forecasts based on the given data. Classical machine learning has its quantum part, which is known as quantum machine learning (QML). QML, which is a

Citation: Senekane, M. and Taele, B. (2016), "Prediction of Solar Irradiation Using Quantum Support Vector Machine Learning Algorithm". *Smart Grid and Renewable Energy*, **7**, 293-301. doi: 10.4236/sgre.2016.712022.

field of quantum computing, uses some of the quantum mechanical principles and concepts which include superposition, entanglement and quantum adiabatic theorem to assess the data and make some forecasts based on the data. At the present moment, research in QML has taken two main approaches. The first approach involves implementing the computationally expensive subroutines of classical machine learning algorithms on a quantum computer. The second approach concerns using classical machine learning algorithms on a quantum information, to speed up performance of the algorithms. The work presented in this manuscript proposes a quantum support vector algorithm that can be used to forecast solar irradiation. The novelty of this work is in using quantum mechanical principles for application in machine learning. Python programming language was used to simulate the performance of the proposed algorithm on a classical computer. Simulation results that were obtained show the usefulness of this algorithm for predicting solar irradiation.

Keywords: Quantum, Quantum Machine Learning, Machine Learning, Support Vector Machine, Quantum Support Vector Machine, Energy, Solar Irradiation

INTRODUCTION

Machine learning is a subfield of artificial intelligence. It is a set of techniques that are used to analyze and find patterns in input data to make predictions/inferences [1] - [10] . It has applications in areas such as image recognition, natural language processing, robotics, spam filtering, drug discovery, medical diagnosis, financial analysis, bioinformatics, marketing and even politics [10] [11] [12] .

There are various classical machine learning algorithms, and these include Bayesian networks, artificial neural networks, deep learning, clustering and Support Vector Machine (SVM) to name but a few. The main focus of this paper is on the quantum version of SVM algorithm, which was introduced by Vapnik in the 1990s [13] . Machine learning algorithms can be divided into three major categories, namely supervised learning, unsupervised learning and reinforcement learning, depending on the type of data to be used for predictive analytics [1] [3] [10] [13] .

The field of Quantum Information Processing (QIP) exploits quantum mechanical concepts such as superposition, entanglement and tunneling for computation and communication tasks [14] . Recently, there has been a concerted effort to explore the benefits of using QIP for machine learning applications. This results in the field of Quantum Machine Learning (QML). It has also been demonstrated that QML techniques provide a performance speedup compared to their classical counterparts [11] [15] . This speedup is the major motivation for exploring QML algorithms.

There are two basic approaches to QML [9] . The first approach uses the classical data as input, and transforms it into quantum data so that it could be processed on a quantum computer. In essence, this approach implements classical machine learning algorithms on a quantum computer. The second approach involves making use of quantum mechanical principles in order to design machine learning algorithms for classical computers. In the work reported in this paper, we used the first approach to model solar power using quantum SVM.

The remainder of this paper is structured as follows. The next section provides background information on machine learning, QIP and QML. This is followed by Section 3, which discusses the design and implementation of the sun power prediction model reported in this Manuscript. Section 4 provides the results and discusses the results obtained. Finally, the last section concludes this paper.

BACKGROUND INFORMATION

Machine learning, which is used interchangeably with predictive analytics, is a sub-field of artificial intelligence which is concerned with building algorithms that make use of input data to make predictions [1] [2] [3] [4] . There are three main categories of machine learning, and they are [1] [10] :

-Supervised learning: makes use of both training data and data label to make predictions about future points. Examples of supervised learning algorithms are logistic regression, artificial neural networks and support vector machines.

- Unsupervised learning: makes use of training data only to make a model that maps inputs to output. As opposed to supervised learning, unsupervised learning does not make use of data label. Examples of unsupervised learning are clustering and anomaly detection algorithms.
- Reinforcement learning: uses reinforcement in the form of reward or punishment. If the algorithm succeeds in making correct predictions, it is rewarded. However, if it fails, it is punished.

Reinforcement learning is used mainly in robotics and computer games.

Support Vector Machines

Support vector machine learning is the most commonly used "off-the-shelf" supervised learning algorithm [1] . SVM solves problems in both classification and regression. It uses the principle of maximum margin classifier to separate data. For a d- dimensional data, SVM uses a d ? 1 hyperplane for data separation. For instance, if data are supplied on a plane (two dimensions), SVM would use a line (one dimension) for classification. The principle of maximum margin classification ensures that there is a maximum separation between positive results ($y = 1$) and negative results ($y = -1$). The margin in this case is the distance between the decision boundary and the support vectors, where support vectors are data points closest to the decision boun- dary.

One of the key advantages of support vector machines is that unlike other supervised learning algorithms, its loss function is a global optimization problem, hence it is not prone to local optima [4] . Additionally, SVM is robust against over-fitting, hence it is suitable for making generalizations even with a small dataset. Lastly, by using a technique known as kernel trick, SVM can separate data which is not linearly separable in its input space. This technique enables SVM to transform input data into higher-dimen- sional space, where a separating linear hyperplane can be found.

Quantum Information Processing

In stark contrast to classical computers, which use a binary digit (bit) as a unit of information, quantum computers use a quantum bit (qubit) as a unit of information. Mathematically, a qubit is given as [14] [16]

$$\left|\psi\right\rangle = \alpha\left|0\right\rangle + \beta\left|1\right\rangle \tag{1}$$

where α and β are probability amplitudes. These amplitudes satisfy the condition

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2}$$

It is worth noting that a qubit, which is a unit of information for a two-state system, can be generalized to any arbitrary d-state. Such a generalized unit of information is known as a quantum digit (qudit) [16] . Just like a classical computer, which use gates for computation, quantum computers also use quantum gates to perform operations on qudits. Essentially, a quantum gate operation on a quantum state $|\psi\rangle$ (which is represented as a column vector) is a linear operation. Therefore, mathematically speaking, quantum information processing makes use of vectors, matrices and tensors, hence it involves linear transformations.

Quantum Machine Learning

Machine learning generally represents data in vector and matrix form. This is also the case with QIP, hence why QIP concepts find applications in machine learning. This results in the new field of research called quantum machine learning. Quantum machine learning can take two forms: where classical machine learning algorithms are transformed into their quantum counterparts; to be implemented on a quantum information processor, or taking some of the computationally expensive classical machine learning sub-routines and implementing them on the quantum computer.

Model Evaluation and Validation

Different measures are used to evaluate and validate models. These measures include mean squared error (MSE), Root mean squared error (RMSE), mean absolute error (MAE), and R^2 error.

Mean Squared Error

Mean squared error is one of the measures of the goodness of fit. It measures the closeness of a data line to the data points. For n as the number of predictions, $\tilde{y}$ as the vector of predicted values, and Y as the vector of observations, MSE is given as

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(\tilde{y}_i - Y_i\right)^2 \tag{3}$$

Root Mean Squared Error

Root mean squared error, which is also a measure of goodness of fit, is the average Euclidean distance of the line from the data points. It is given as

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\tilde{y}_i - Y_i\right)^2} \tag{4}$$

where n is the number of predictions, $\tilde{y}$ is the vector of predicted values, and Y is the vector of observations.

Mean Absolute Error

Mean absolute error measures the closeness of predicted results to the observations. It is given as

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}\left|\tilde{y}_i - Y_i\right| \tag{5}$$

R² Error

R² error is also known as coefficient of determination. It is the measure of degree of variance. It is given as

$$R^2 = s - \frac{\text{SStot}}{\text{SSres}} \tag{6}$$

where, for a mean of observations $\bar{y}$, SStot is given as

$$\text{SStot} = \sum_{i=1}^{n}\left(Y_i - \bar{y}_i\right)^2 \tag{7}$$

and

$$\text{SSres} = \sum_{i=1}^{n}\left(\tilde{y}_i - Y_i\right)^2 \tag{8}$$

IMPLEMENTATION

In this work, quantum support vector machine was implemented using a recorded data from Digital Technology Group (DTG) Weather Station in Cambridge University[1]. The dataset consists of forty nine instances, which are the training examples. These instances represent the measurements that were recorded at DTG, with a time interval of thirty minutes. Additionally, this dataset consists of three features, namely temperature, humidity and wind speed.

The recorded classical information is converted to quantum state such that for a training example $\ddot{x}$ and number of training examples N:

$$|\ddot{x}\rangle = \frac{1}{|\ddot{x}|}\sum_{j=1}^{n}(\ddot{x})j|j\rangle \tag{9}$$

This is then followed by optimizing the quantum support vector hyperplane parameters, as articulated in [17] . The optimization is done by reducing this optimization problem into a system of linear equations, and then using a quantum algorithm for solving a system of linear equations, which uses matrix inversion. This quantum algorithm is known to have an exponential speedup over its classical counterpart.

The quantum support vector machine was implemented using Python programming language.

Python machine learning package used for this task was Scikit-learn version 0.18.0 [5] . The graphical user interface (GUI) part of the implementation was realized using Orange data mining software package, release number 3.3.8[2]. This GUI helped visualize the input dataset and the plots for the results obtained from this implementation. It also supports other python packages such as scikit-learn.

The results were then recorded and errors calculated. The following errors were calculated, for different training sizes:

_mean square error (MSE),

_root mean square error (RMSE),

_mean absolute error (MAE),

_coefficient of determination, R^2.

RESULTS AND DISCUSSION

The dataset was broken down into different portions, with some part being used for training data, and the other part being used for cross-validation. Table 1 shows different calculated errors for different training data sizes. From the table, it can be observed that the best results are obtained when the training size is 70% of the dataset. Therefore, the training size of 70% was chosen for this implementation.

The next step was to analyze the correlation of the three features used (temperature, humidity and wind speed). Figure 1 and Figure 2 show the scatter plots of these correlations. Since the graphs in the figures are not linear, it implies that the features were not correlated, hence they were independent. Finally, Sieve diagrams were plotted, and are shown in Figure 3 and Figure 4. These results underline the robustness of the proposed algorithm.

CONCLUSIONS

We have reported an algorithm for solar power prediction using quantum support vector machine learning algorithm. The algorithm is a quantum counterpart of a classical support vector machine, which is known to have a unique solution, and hence it converges to a global optimum.

Table 1: Calculated errors for different dataset training sizes

Training Size (%)	MSE	RMSE	MAE	R^2
60	3.629	1.905	1.435	0.978
66	3.098	1.760	1.283	0.826
70	2.643	1.626	1.171	0.852
75	2.992	1.730	1.259	0.835
80	3.014	1.736	1.257	0.835

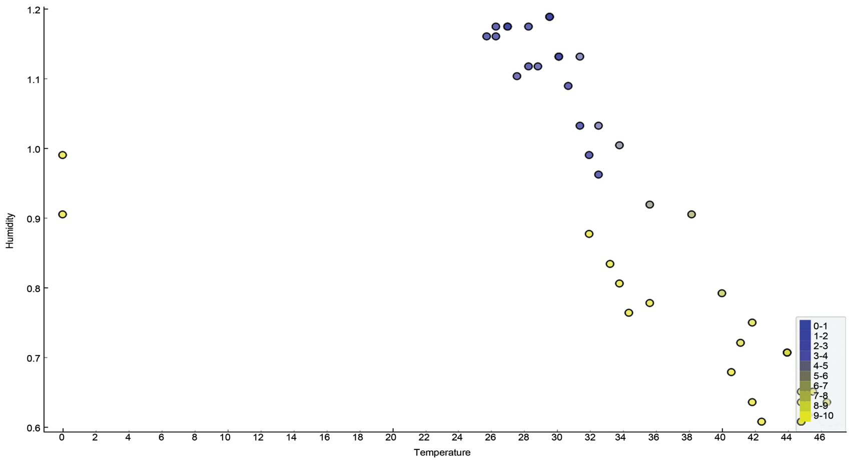


Figure 1: This figure shows the relationship between temperature (in degrees Celsius) and humidity. The non-linearity of the data points implies that the two features are not correlated.

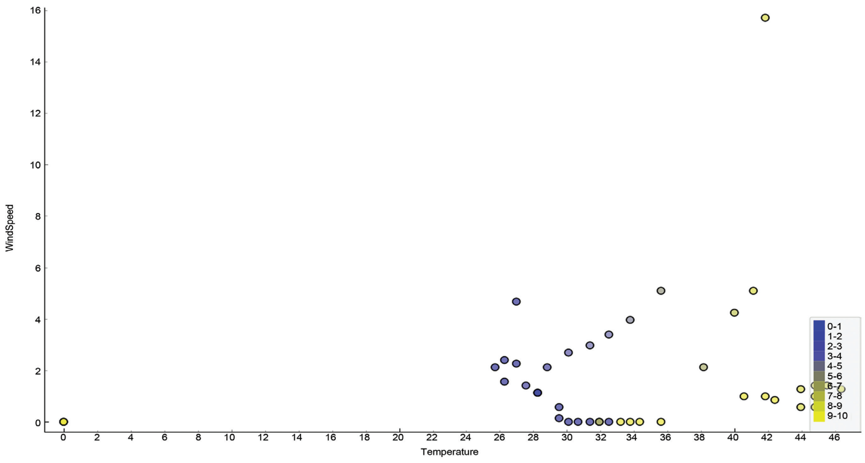


Figure 2: This figure shows the relationship between temperature (in degrees Celsius) and wind speed (in knots). Since the data point portray non-linearity, it can be observed that these two features are independent.

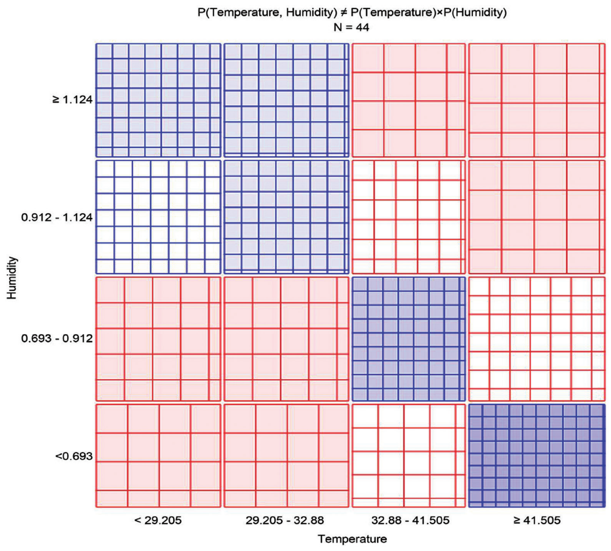


Figure 3: A sieve diagram for temperature and humidity attributes.

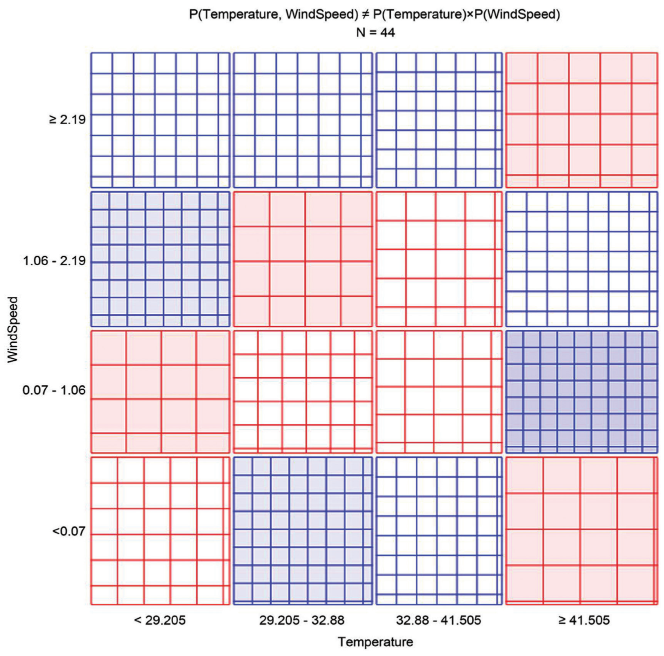


Figure 4: A sieve diagram for temperature and wind speed attributes.

This is in contrast to other machine learning algorithms such as neural networks, which can converge to local optima, since they may not have unique solutions.

In the work reported in this paper, the quantum support vector algorithm was simulated using Python programming language. A dataset with forty nine instances and three features (temperature, humidity and windspeed) was used for this simulation. The results obtained from the simulation underline the utility of the proposed quantum support vector algorithm for solar power prediction. However, it should be noted that in the implementation, a generic optimization algorithm was used for implementing quantum SVM. Future work should explore the feasibility.

REFERENCES

1. Russell, S.J., Norvig, P., Canny, J.F., Malik, J.M. and Edwards, D.D. (2010) Artificial Intelligence: A Modern Approach. Prentice Hall, New York.
2. Rogers, S. and Girolami, M. (2015) A First Course in Machine Learning. CRC Press, London.
3. Sugiyama, M. (2015) Introduction to Statistical Machine Learning. Morgan Kaufmann, Amsterdam.
4. Bishop, C.M., et al. (2006) Pattern Recognition and Machine Learning. Springer, New York.
5. Garreta, R. and Moncecchi, G. (2013) Learning Scikit-Learn: Machine Learning in Python. Packt Publishing Ltd., Birmingham.
6. Raschka, S. (2015) Python Machine Learning. Packt Publishing Ltd., Birmingham.
7. Ivezic, Z., Connolly, A., Vanderplas, J. and Gray, A. (2014) Statistics, Data Mining and Machine Learning in Astronomy. Princeton University Press, Princeton, New Jersey.
8. Lantz, B. (2013) Machine learning with R. Packt Publishing Ltd., Birmingham.
9. Wittek, P. (2014) Quantum Machine Learning: What Quantum Computing Means to Data Mining. Academic Press, Cambridge, Massachusetts.
10. Schuld, M., Sinayskiy, I. and Petruccione, F. (2015) An Introduction to Quantum Machine Learning. Contemporary Physics, 56, 172-185.
11. Cai, X.D., Wu, D., Su, Z.E., Chen, M.C., Wang, X.L., Li, L., Liu, N.L., Lu, C.Y. and Pan, J.W. (2015) Entanglement-Based Machine Learning on a Quantum Computer. Physical Review Letters, 114, 110504. https://doi.org/10.1103/PhysRevLett.114.110504
12. Siegel, E. (2013) Predictive Analytics: The Power to Predict Who Will Click, Buy, Lie, or Die. John Wiley & Sons, Hoboken, New Jersey.
13. Marsland, S. (2015) Machine Learning: An Algorithmic Perspective. CRC Press, Boca Raton, Florida.
14. Nielsen, M.A. and Chuang, I.L. (2010) Quantum Computation and Quantum Information. Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/CBO9780511976667

15. Lloyd, S., Mohseni, M. and Rebentrost, P. (2013) Quantum Algorithms for Supervised and Unsupervised Machine Learning. arXiv:1307.0411
16. Wilde, M.M. (2013) Quantum Information Theory. Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/CBO9781139525343
17. Li, Z., Liu, X., Xu, N. and Du, J. (2015) Experimental Realization of a Quantum Support Vector Machine. Physical Review Letters, 114, 140504. https://doi.org/10.1103/PhysRevLett.114.140504

CHAPTER

7

PREDICTING ACADEMIC ACHIEVEMENT OF HIGH-SCHOOL STUDENTS USING MACHINE LEARNING

Hudson F. Golino[1], Cristiano Mauro Assis Gomes[2], Diego Andrade[2]

[1]Núcleo de Pós-Graduação, Pesquisa e Extensão, Faculdade Independente do Nordeste, Vitória da Conquista, Brazil

[2]Department of Psychology, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

ABSTRACT

The present paper presents a relatively new non-linear method to predict academic achievement of high school students, integrating the fields of psychometrics and machine learning. A sample composed by 135 high-school students (10th grade, 50.34% boys), aged between 14 and 19 years old (M = 15.44, DP = 1.09), answered to three psychological instruments:

Citation: Golino, H. , Gomes, C. & Andrade, D. (2014). "Predicting Academic Achievement of High-School Students Using Machine Learning". *Psychology, 5,* 2046-2057. doi: 10.4236/psych.2014.518207.

the Inductive Reasoning Developmental Test (TDRI), the Metacognitive Control Test (TCM) and the Brazilian Learning Approaches Scale (BLAS-Deep Approach). The first two tests have a self-appraisal scale attached, so we have five independent variables. The students' responses to each test/scale were analyzed using the Rasch model. A subset of the original sample was created in order to separate the students in two balanced classes, high achievement (n = 41) and low achievement (n = 47), using grades from nine school subjects. In order to predict the class membership a machine learning non-linear model named Random Forest was used. The subset with the two classes was randomly split into two sets (training and testing) for cross validation. The result of the Random Forest showed a general accuracy of 75%, a specificity of 73.69% and a sensitivity of 68% in the training set. In the testing set, the general accuracy was 68.18%, with a specificity of 63.63% and with a sensitivity of 72.72%. The most important variable in the prediction was the TDRI. Finally, implications of the present study to the field of educational psychology were discussed.

Keywords: Machine Learning, Assessment, Prediction, Intelligence, Learning Approaches, Metacognition

INTRODUCTION

Machine learning is a relatively new science field composed by a broad class of computational and statistical methods to make predictions, inferences, and to discover new relations in data (Flach, 2012; Hastie, Tibshirani, & Friedman, 2009) . There are two main areas within the machine learning field. The unsupervised learning focuses in the discovery and detection of new relationships, patterns and trends in data. The supervised learning area, by the other side, focuses in the prediction of an outcome using a given set of predictors. If the outcome is categorical, then the task to be accomplished is named classification, if it is numeric then the task is called regression.

There are several types of algorithms to perform classification and regression (Hastie et al., 2009) . Among these algorithms, the tree based models are supervised learning techniques of special interest to the psychology and to the education research field. It can be used to discover which variable, or combination of variables, better predicts a given outcome, e.g. high or low academic achievement. It can identify the cutoff points for each variable that maximally predict the outcome, and can also be applied to study the non-linear interaction effects of the independent variables and its relation to the quality of the prediction (Golino & Gomes, 2014) . Within psychology, there are a growing number of applications of the tree-based models in different areas, from ADHA diagnosis(Eloyan et al., 2012; Skogli et al., 2013) to perceived stress (Scott, Jackson, & Bergeman, 2011) , suicidal behavior (Baca-Garcia et al., 2007; Kuroki & Tilley, 2012) , adaptive depression assessment (Gibbons et al., 2013) , emotions (Tian et al., 2014; van der Wal & Kowalczyk, 2013)and education (Blanch & Aluja, 2013; Cortez & Silva, 2008; Golino & Gomes, 2014; Hardman, Paucar-Caceres, & Fielding, 2013) .

The main benefit of using the tree-based models in psychology is that they do not make any assumption regarding normality, linearity of the relation between variables, homoscedasticity, collinearity or independency (Geurts, Irrthum, & Wehenkel, 2009) . The tree-based models also do not demand a high sample-to-predictor ratio and are more suitable to interaction effects (especially non-linearity) than the classical techniques, such as linear and logistic regression, ANOVA, MANOVA, structural equation modelling and so on. Finally, the tree- based models, especially the ensemble techniques, can lead to high prediction accuracy, since they are known as the state-of-the-art methods in terms of prediction accuracy (Flach, 2012; Geurts et al., 2009) . The current paper focuses on the methodological aspects of the classification tree (Breiman, Friedman, Olshen, & Stone, 1984) and its most famous ensemble technique, Random Forest (Breiman, 2001a) . To illustrate the use of tree-based models in educational psychology, the Random Forest algorithm will be used to predict levels of academic achievement of high school students (low vs. high). Finally, we will discuss the limits and possibilities of this new predictive method to the field of educational psychology.

Recursive Partitioning and Ensemble Techniques

A classification tree partitions the feature space into several distinct mutually exclusive regions (non-overlap- ping). Each region is fitted with a specific model that designates one of the classes to that particular space. The class is assigned to the region of the feature space by identifying the majority class in that region. In order to arrive in a solution that best separates the entire feature space into more pure nodes (regions), recursive binary partition is used. A node is considered pure when 100% of the cases are of the same class, for example, low academic achievement. A node with 90% of low achievement and 10% of high achievement students is more “pure” then a node with 50% of each. Recursive binary partitions work as follows. The feature space is split into two regions using a specific cutoff from the variable of the feature space (predictor) that leads to the most purity configuration. Then, each region of the tree is modeled accordingly to the majority class. One or two original nodes are also split into more nodes, using some of the given predictors that provide the best fit possible. This splitting process continues until the feature space achieves the most purity configuration possible, with Rm regions or nodes classified with a distinct Ck class. If more than one predictor is given, then the selection of each variable used to split the nodes will be given by the variable that splits the feature space into the most purity configuration. In a classification tree, the first split indicates the most important variable, or feature, in the prediction. Let's take a look in Figure 1 to see how a classification tree looks like.

Figure 1 shows the classification tree presented by Golino and Gomes (2014) with three predictors of the academic achievement (high and low) of medicine students: The Metacognitive Control Test (TCM), Deep Learning Approach (DeepAp) and the Self-Appraisal of the Inductive Reasoning Developmental Test (SA_ TDRI). The most important variable in the prediction was TCM, since it was the predictor located at the first split of the classification tree.

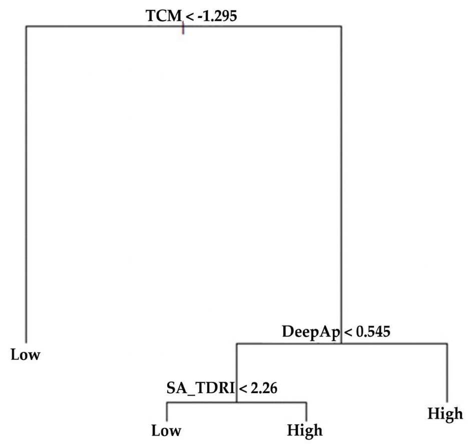


Figure 1: A classification tree from Golino and Gomes (2014) .

The first split indicates the variable that separates the feature space into two purest nodes. In the case shown in Figure 1, 52.50% of the sample used to grow the tree had a TCM score smaller than −1.295, and were classified as having a low academic achievement. The remaining 47.5% had a TCM score greater than −1.295, and were classified in the low or in the high achievement class accordingly their scores on the DeepAp and on the SA_TDRI. Those with a TCM score greater than −1.295 and a DeepAp score greater than .545 were classified as belonging to the high achievement class. The same occurred to those with a TCM score greater than −1.295, a DeepAp score lower than .545 and a SA_TDRI score greater than 2.26. Finally, the participants with a TCM score greater than −1.295, a DeepAp score lower than .545 but with a SA_TDRI score smaller than 2.26 were classified as belonging to the low achievement group. This classification tree presented a total accuracy of 72.50%, with a sensitivity of 57.89% and a specificity of 85.71%(Golino & Gomes, 2014) .

Geurts, Irrthum and Wehenkel (2009) argue that learning trees are among the most popular algorithms of machine learning due to its interpretability, flexibility and ease of use. Interpretability referrers to its easiness of understanding. It means that the model constructed to map the feature space (predictors) into the output space (dependent variable) is easy to understand, since it is a roadmap of if-then rules. The description of Figure 1 above shows exactly that. James, Witten, Hastie and Tibshirani (2013) points that the tree models are easier to explain to people than linear regression, since it mirrors more the human decision-making then other predictive models. Flexibility means that the tree techniques are applicable to a wide range of problems, handles different kind of variables (including nominal, ordinal, interval and ratio scales), are non-parametric techniques and does not make any assumption regarding normality, linearity or independency(Geurts et al., 2009) . Furthermore, it is sensible to the impact of additional variables to the model, being especially relevant to the study of incremental validity. It also assesses which variable or combination of them, better predicts a given outcome, as well as calculates which cutoff values are maximally predictive of it. Finally, the ease of use means that the tree based techniques are computationally simple, yet powerful.

In spite of the qualities of the learning trees, it suffers from two related limitations. The first one is known as the overfitting issue. Since the feature space is linked to the output space by recursive binary partitions, the tree models can learn too much from data, modeling it in such a way that may turn out a sample dependent model. Being sample dependent, in the sense that the partitioning is too suitable to the data set in hand, it will tend to behave poorly in new data sets.Golino and Gomes (2014) showed that in spite of having a total accuracy of 72.50% in the training sample, the tree presented in Figure 1 behaved poorly in a testing set, with a total accuracy of 64.86%. The difference between the two data sets is due to the overfit of the tree to the training set.

The second issue is exactly a consequence of the overfitting, and is known as the variance issue. The predictive error in a training set, a set of features and outputs used to grown a classification tree for the first time, may be very different from the predictive error in a new test set. In the presence of overfitting, the errors will present a large variance from the training set to the test set used, as shown by the results of Golino and Gomes (2014) . Additionally, the classification tree does not have the same predictive accuracy as other classical machine learning approaches (James et al., 2013) . In order to prevent overfitting, the variance issue and also to increase the prediction accuracy of the classification trees, a strategy named ensemble trees can be used.

The ensemble trees are simply the junction of several models to perform the classification task based on the prediction made by every single tree. The most famous ensemble tree algorithm is the Random Forest (Breiman, 2001a) , that is used to increase the prediction accuracy, decrease the variance between data sets and to avoid overfitting.

The procedure takes a random subsample of the original data set (with replacement) and of the feature space to grow the trees. The number of the selected features (variables) is smaller than the number of total elements of the feature space. Each tree assigns a single class to the each region of the feature space for every observation. Then, each class of each region of every tree grown is recorded and the majority vote is taken (Hastie et al., 2009; James et al., 2013) . The majority vote is simply the most commonly occurring class over all trees. As the Random Forest does not use the entire observations (only a subsample of it, usually 2/3), the remaining observations (known as out-of-bag, or OOB) is used to verify the accuracy of the prediction. The out-of-bag error can be computed as a "valid estimate of the test error for the bagged model, since the response for each observation is predicted using only the trees that were not fit using that observation" (James et al., 2013: p. 323) .

As pointed by Breiman (2001a) , the number of selected variables is held constant during the entire procedure for growing the forest, and usually is set to square-root of the total number of variables. Since the Random Forest subsamples the original sample and the predictors, it is considered an improvement over other ensemble trees, as the bootstrap aggregating technique (Breiman, 2001b), or simply bagging. Bagging is similar to Random Forest, except for the fact that does not subsample the predictors. Thus, bagging creates correlated trees (Hastie et al., 2009) , which may affect the quality of the prediction. The Random Forest algorithm decorrelates the trees grown, and as a consequence it also decorrelates the errors made by each tree, yielding a more accurate prediction.

Why decorrelating the trees is so important? Following the example created by James et al. (2013), imagine that we have a very strong predictor in our feature space, together with other moderately strong predictors. In the bagging procedure, the strong predictor will be in the top split of most of the trees, since it is the variable that better separates the classes available in our data. By consequence, the bagged trees will be very similar to each other, making the predictions and the errors highly correlated. This may not lead to a decrease in the variance if compared to a single tree. The Random Forest procedure, on the other hand, forces each split to consider only a subset of the features, opening chances for the other variables to do their job. The strong predictor will be left out of the bag in a number of situations, making the trees very different from each other. Therefore, the resulting trees will present less variance in the classification error and in the OOB error, leading to a more reliable prediction. In sum, the Random Forest is an ensemble of trees that improves the prediction accuracy, decreases variance and avoids overfitting by using only a subsample of the observations and a subsample of predictors. It has two main tuning parameters. The first is the size of the subsample of features used in each split (mtry), which is mandatory to be smaller than the total number of features, and is usually set as the square root of the total number of predictors. The second tuning parameter is the number of trees to grow (n_{tree}).

The present paper investigates the prediction of academic achievement of high-school students (high achieve- ment vs. low achievement) using two psychological tests and one educational scale: the Inductive Reasoning Developmental Test (TDRI), the Metacognitive Control Test (TCM) and the Brazilian Learning Approaches Scale (BLAS-Deep approach). The first two tests have a self-appraisal scale attached, so we have five independent variables. In the next section will be presented the participants, instruments used and the data analysis procedures.

METHOD

Participants

The sample is composed by 135 high-school students (10th grade, 50.34% boys), aged between 14 and 19 years old (M = 15.44, DP = 1.09), from a public high-school from [omitted as required by the review process]. The sample was selected by convenience, and represents approximately 90% of the students of the 10th grade. The students received a letter inviting them to be part of the study. Those who agreed in participating signed a inform consent, and confirmed they would be present in the schedule days to answer all the instruments.

Measures and Procedures

The Inductive Reasoning Developmental Test (TDRI) and Its Self-Appraisal Scale (SA_TDRI)

The Inductive Reasoning Developmental Test (TDRI) was developed by Gomes and Golino (2009)and by Golino and Gomes (2012) to assess developmental stages of reasoning based on Common's Hierarchical Complexity Model (Commons, 2008; Commons & Pekker, 2008; Commons & Richards, 1984) and on Fischer's Dynamic Skill Theory (Fischer, 1980; Fischer & Yan, 2002) . This is a pencil-and-paper test composed by 56 items, with a time limit of 100 minutes. Each item presents five letters or set of letters (see Figure 2), being four with the same rule and one with a different rule. The task is to identify which letter or set of letters have the different rule.

Golino and Gomes (2012) evaluated the structural validity of the TDRI using responses from 1459 Brazilian people (52.5% women) aged between 5 to 86 years (M = 15.75, SD = 12.21). The results showed a good fit to the Rasch model (INFIT mean = .96; SD = .17) with a high separation reliability for items (1.00) and a moderately high for people (.82). The item's difficulty distribution formed a seven cluster structure with gaps between them, presenting statistically significant differences in the 95% C.I. level (t-test). The CFA showed an adequate data fit for a model with seven first-order factors and one general factor [χ^2 (61) = 8832.594, p = .000, CFI = .96, RMSEA = .059]. The latent class analysis showed that the best model is the one with seven latent classes (AIC: 263.380; BIC: 303.887; Loglik: −111.690). The TDRI test has a self-appraisal scale attached to each one of the 56 items. In this scale, the participants are asked to appraise their achievement on the TDRI items, by reporting if he/she passed or failed the item. The scoring procedure of the TDRI self-appraisal scale works as follows. The participant receive a score of 1 in two situations: 1) if the participant passed the ith item and reported that he/she passed the item, and 2) if the participant failed the ith item and reported that he/she failed the item. On the other hand, the participant receives a score of 0 if his appraisal does not match his performance on the ith item: 1) he/ she passed the item, but reported that failed it, and 2) he/she failed the item, but reported that passed it.

The Metacognitive Control Test (TCM) and Its Self-Appraisal Scale (SA_TCM)

The Metacognitive Control Test (TCM) was developed by Golino and Gomes (2013) to assess the ability of people to control intuitive answers to logical-mathematical tasks. The test is based on Shane Frederick's Cognitive Reflection Test (Frederick, 2005) , and is composed by 15 items. The structural validity of the test was assessed by Golino and Gomes (2013) using responses from 908 Brazilian people (54.8% women) aged between 9 to 86 years (M = 27.70, SD = 11.90). The results showed a good fit to the Rasch model (INFIT mean = 1.00; SD = .13) with a high separation reliability for items (.99) and a moderately high for people (.81). The TCM also has a self-appraisal scale attached to each one of its 15 items. The TCM self-appraisal scale is scored exactly as the TDRI self-appraisal scale: an incorrect appraisal receives a score of 0, and a correct appraisal receives a score of 1.

The Brazilian Learning Approaches Scale (EABAP)

The Brazilian Learning Approaches Scale (EABAP) is a self-report questionnaire composed by 17 items, developed by Gomes and colleagues (Gomes, 2010; Gomes, Golino, Pinheiro, Miranda, & Soares, 2011) . Nine items were elaborated to measure deep learning approaches, and eight items measure surface learning approaches. Each item has a statement that refers to a student's behavior while learning. The student considers how much of the behavior described is present in his life, using a Likert-like scale ranging from (1) not at all, to (5) entirely present. BLAS presents reliability, factorial structure validity, predictive validity and incremental validity as good marker of learning approaches. These psychometrical proprieties are described respectively in Gomes et al. (2011), Gomes (2010) , and Gomes and Golino (2012) . In the present study only the deep learning approach items (DeepAp) were used. We will analyze only the nine deep approach items using the partial credit Rasch model.

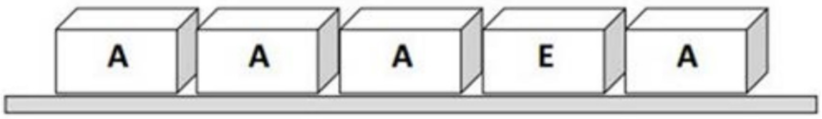


Figure 2: Example of TDRI's item 1 (from the first developmental stage assessed).

Data Analysis

Estimating the Students' Ability in Each Test/Scale

The student's ability estimates on the inductive reasoning developmental test, on the metacognitive control test, on the Brazilian learning approaches scale, and on the self-appraisal scales were computed using the original data set of each test/scale, through the software Winsteps (Linacre, 2012) . This procedure was followed in order to achieve reliable estimates, since only 135 students answered the tests. The mixture of the original data set from each test to the high-school students' answers did not significantly change the reliability or fit to the models used. A summary of the separation reliability and fit of the items, the separation reliability of the sample (after adding the data from the high-school students) and the statistical model used is provided in Table 1.

Defining the Achievement Classes (High vs. Low)

The final grade in the following nine school subjects was provided by the school at the end of the academic year: arts, philosophy, physics, history, informatics, math, chemistry, sociology and Brazilian Portuguese. The final grades ranged from 0 to 10, and the students were considered approved in the academic year in each school subject only if he/she had a grade equal to or above seven. Students with grades lower than seven in a particular school subject are submitted to an additional assessment. Finally, those with an average grade of seven or more are considered able to proceed to the next school grade (11th grade). Otherwise, the students need to re-do the current grade (10th grade). From the total sample, only 65.18% (n = 88) were considered able to proceed to the next school year and 34.81% (n = 47) were requested to re-do the 10th grade. These two groups could be used to compose the high and the low achievement classes. However, since the tree-based models require balanced classes (i.e., classes with approximately the same number of cases) we needed to subset the high achievement class (those who proceeded to the next school grade) in order to obtain a subsample closer to the low achievement class size (those who would need to re-do the 10th grade). Therefore, we computed the mean final grade over all nine grades for every student, and verified the mean of each group of students. Those who passed to the next school grade had a mean final grade of 7.69 (SD = .48), while those who would need to re-do the 10th grade had a mean final grade of 6.06 (SD = 1.20). We select every student with a mean final grade equals to or higher than 7.69 (n = 41) and called them the "high achievement" group. The 47 students that would need to re- do the 10th grade formed the "low achievement" group. Finally, we had 88 students divided in two balanced classes.

Machine Learning Procedures

The sample was randomly split in two sets with equal sizes, training and testing, for cross-validation. The training set is used to grow the trees, to verify the quality of the prediction in an exploratory fashion, and to adjust the tuning parameters. Each model created using the training set is applied in the testing set to verify how it performs on a new data set.

Since the single trees usually lead to overiftting and to high variance between datasets, we used only the Random Forest algorithm through the random Forest package (Liaw & Wiener, 2012) of the R software (R Development Core Team, 2011). As pointed in the introduction, the Random Forest has two main tuning parameters: the number of trees (n_{tree}) and the number of variables used (mtry). We set mtry as two, because is the integer closest to the square root of the total number of predictors (5), and n_{tree} as 10,000. In order to verify the quality of the prediction both in the training (modeling phase) and in the testing set (cross-validation phase), the total accuracy, the sensitivity and specificity were used.

Table 1: Item reliability, item fit, person reliability, person fit and model used by instrument

Test	Item reli-ability	Item INFIT (mean, SD)	Person reliabil-ity	Person INFIT (mean, SD)	Model
Inductive reasoning develop-mental test (TDRI)	1.00	.98, .17	.85	.98, .91	Dichotomous Rasch Model
TDRI's self-appraisal scale (SA_TDRI)	.98	.98, .11	.79	.97, .31	Dichotomous Rasch Model
Metacognitive control test (TCM)	.99	1.00, .13	.80	.99, .31	Dichotomous Rasch Model
TCM's self-appraisal scale (SA_TCM)	.98	1.02, .26	.74	.98, .20	Dichotomous Rasch Model
Brazilian learning approaches scale— Deep learning items (DeepAp)	.99	1.00, .08	.80	1.01, .69	Partial Credit Rasch Model
Inductive reasoning develop-mental test (TDRI)	1.00	.98, .17	.85	.98, .91	Dichotomous Rasch Model

Total accuracy is the proportion of observations correctly classified:

$$Acc = \frac{1}{n|T_E|} \sum_{x \in T_E} I(y_i = C_k)$$

where $n|T_E|$ is the number of observations in the testing set. In spite of being an important indicator of the general prediction's quality, the total accuracy is not an informative measure of the errors in each class. For example, a general accuracy of 80% can represent an error-free prediction for the C1 class, and an error of 40% for the C2 class. In the educational scenario, it is preferable to have lower error in the prediction of the low achievement class, since students at risk of academic failure compose this class. So, the sensitivity will be preferred over general accuracy and specificity. The sensitivity is the rate of observations correctly classified in a target class, e.g. C1 = low achievement, over the number of observations that belong to that class:

$$Sens = \frac{\sum_{x \in T_E} I(y_i = C_1)}{\sum_{x \in T_E} I(C_1)}$$

Specificity, on the other hand, is the rate of correctly classified observations of the non-target class, e.g. C2 = high achievement, over the number of observations that belong to that class:

$$Spec = \frac{\sum_{x \in T_E} I(y_i = C_2)}{\sum_{x \in T_E} I(C_2)}$$

Finally, the model construct in the training set will be applied in the testing set for cross-validation. Since the Random Forest is a black box technique—i.e. there is only a prediction based on majority vote and no "typical tree" to look at the partitions—to determine which variable is important in the prediction one importance measure will be used: the mean decrease of accuracy. It indicates how much in average the accuracy decreases on the out-of-bag samples when a given variable is excluded from the model (James et al., 2013).

Descriptive Analysis Procedures

After estimating the student's ability in each test or scale the Shapiro-Wilk test of normality will be conducted in order to discover which variables presented a normal distribution. To verify if there is any statistically significant difference between the students' groups (high achievement vs. low achievement) the two-sample T test will be conducted in the normally distributed variables and the Wilcoxon Sum-Rank test in the non-normal variables, both at the .05 significance level. In order to estimate the effect sizes of the differences, the R's compute.es package (Del Re, 2013) is used. This package computes the effect sizes, along with their variances, confidence intervals, p-values and the common language effect size (CLES) indicator using the p-values of the significance testing. McGraw and Wong (1992) developed the CLES indicator as a more intuitive tool than the other effect size indicators. It converts an effect into a probability that a score taken at random from one distribution will be greater than a score taken at random from another distribution(McGraw & Wong, 1992) . In other words, it expresses how much (in %) the score from one population is greater than the score of the other population if both are randomly selected (Del Re, 2013) .

RESULTS

Descriptive

The Brazilian Learning Approaches Scale (Deep Learning) presented a normal distribution (W = .99, p-value = .64), while all the other four variables presented a p-value smaller than .001. There was a statistically significant difference at the 99% level between the high and the low achievement groups in the median Rasch score of the Inductive Reasoning Developmental ($\tilde{X}_{High} = 2.14$, $\sigma^2 = 5.80$, $\tilde{X}_{Low} = -1.47$, $\sigma^2_{Low} = 15.52$, W = 1359, p < .01), in the median Rasch score of the Metacognitive Control Test ($\tilde{X}_{High} = -1.03$, $\sigma^2 = 7.29$, $\tilde{X}_{Low} = -3.40$, $\sigma^2_{Low} = 4.37$, W = 928, p < .01), in the median Rasch score of the TDRI's self-appraisal scale ($\tilde{X}_{High} = 2.03$, $\sigma^2 = 3.01$, $\tilde{X}_{Low} = 1.16$, $\sigma^2_{Low} = 4.66$, W = 1152, p < .001), in the median Rasch score of the TCM's self-appraisal scale ($\tilde{X}_{High} = 1.07$, $\sigma^2 = 4.18$, $\tilde{X}_{Low} = -1.08$, $\sigma^2_{Low} = 2.45$, W = 954, p < .01) and in the mean Rasch score of the Brazilian learning approaches scale-deep approach ($\bar{x}_{High} = 1.13$, $\sigma^2 = .80$, $\bar{x}_{Low} = .50$, $\sigma^2_{Low} = .61$, t(37) = 3.32, p < .01). The effect sizes, its 95% confidence intervals, variance, significance and common language effect sizes are described in Table 2.

According to Cohen (1988) , the effect size is considered small when it is between .20 and .49, moderate between .50 and .79 and large when values are over .80. Only the difference in the Rasch score of the inductive reasoning developmental test presented a large effect size (d = .88, p < .05).

As pointed before, the common language effect size indicates how often a score sampled from one distribution is greater than the score sampled from the other distribution if both are randomly selected (McGraw & Wong, 1992) . Then, considering the common language effect size, the probability that a TDRI score taken at random from the high achievement group is greater than a TDRI score taken at random of the low achievement group is 73.41%. It means that out of 100 TDRI scores from the high achievement group, 73.41 will be greater than the TDRI scores of the low achievement group. The Rasch scores of the other tests have moderate effect sizes. Their common language effect size varied from 64.92% to 70.10%, meaning that the probability of a score taken at random at the high achievement group be greater than a score taken at random in the low achievement group is at least 64.92% and at most 70.10%. Figure 3 shows the mean score for each test and its 95% confidence interval by both classes (low and high).

Machine Learning Results

The result of the Random Forest model with 10,000 trees showed an out-of-bag error rate of .29, a total accuracy of 75.00%, a sensitivity of 68.00% and a specificity of 73.69%. The mean decrease accuracy showed the inductive reasoning developmental stage (TDRI) as the most important variable in the prediction, since when it is left out of the prediction the accuracy decreases 66.22% in average. The second most important variable is the deep learning approach, which is associated with a mean decrease accuracy of 28.45% when is not included in the predictive model. In third place is the metacognitive control test (19.68%); in the fourth position is the TDRI self-appraisal scale (19.50%), followed by the TCM self-appraisal scale (5.78%). Figure 4 shows the high achievement prediction error (green line), the out-of-bag error (red line) and the low achievement prediction error (black line) per tree. The errors become more stable with approximately more than 1700 trees.

The predictive model constructed in the training set was applied in the testing set for cross-validation. It presented a total accuracy of 68.18%, a sensitivity of 72.72% and a specificity of 63.63%. There was a difference of 6.82% in the total accuracy, of 2.28% in the sensitivity, and of 10.06% in the specificity.

DISCUSSION

The present paper briefly introduced the concept of recursive partitioning used in the tree-based models of machine learning. The tree-based models are very useful to study the role of psychological and educational constructs in the prediction of academic achievement. Unlike the most classical approaches, such as linear and logistic regression, as well as the structural equation modeling, the tree-based models do not make assumptions about the normality of data, the linearity of the relation between the variables, neither requires homoscedasticity, collinearity or independence (Geurts, Irrthum, & Wehenkel, 2009) . A high predictor-to-sample ratio can be used without harm to the quality of the prediction, and missingness is well handled by the prediction algorithms.

Table 2: Tests, effect sizes and common language effect size (CLES)

Test	Effect size of the difference (d)	95% C.I. (d)	σ^2 (d)	p-value (d)	CLES
Inductive reasoning developmental test (TDRI)	.88	.43, 1.34	.05	.00	73.41%
Metacognitive control test (TCM)	.59	.11, 1.06	.06	.02	66.05%

TDRI' self-appraisal scale (SA_TDRI)	.54	.10, .99	.05	.02	64.92%
TCM' self-appraisal scale (SA_TCM)	.65	.17, 1.12	.06	.01	67.62%
EABAP (DeepAp)	.75	.27, 1.22	.06	.00	70.10%

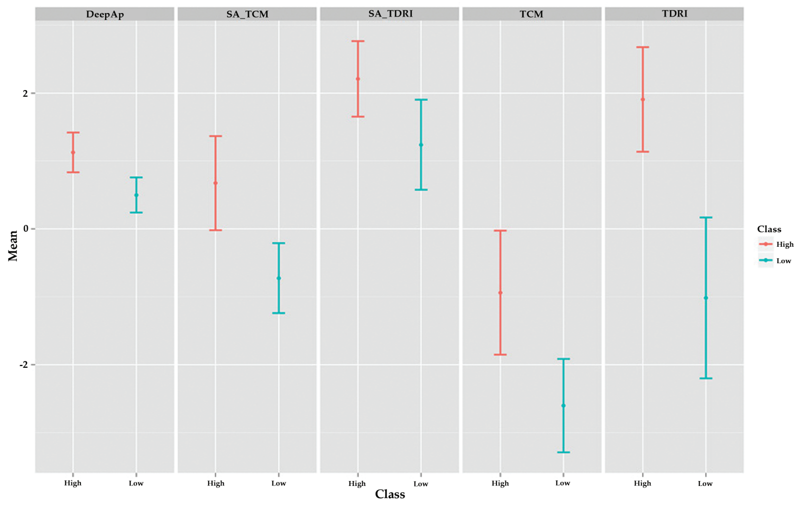


Figure 3: Score means and its 95% confidence intervals for each test, by class (high vs. low academic achievement).

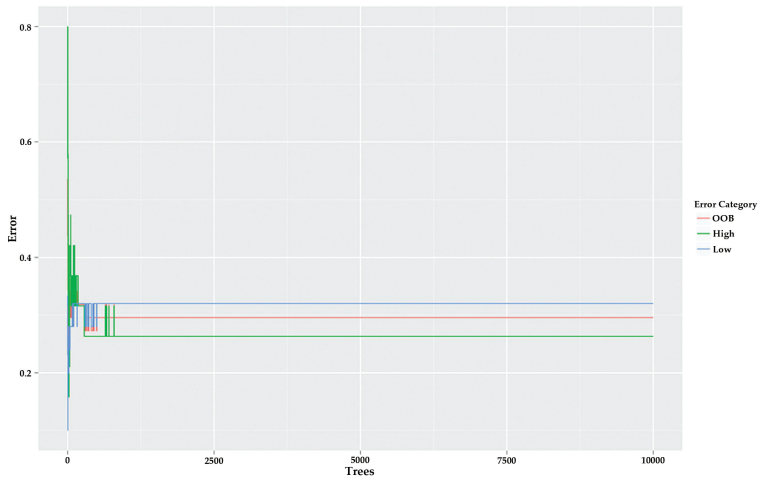


Figure 4: Random Forest's out-of-bag error (red), high achievement prediction error (green) and low achievement predic- tion error (blue).

he tree-based models are also more suitable to non-linear interaction effects than the classical techniques. When several trees are ensemble to perform a prediction it generally leads to a high accuracy (Flach, 2012; Geurts et al., 2009) , decreasing the chance of overfitting and diminishing the variance between datasets. The focus of the current paper was the application of this relatively new predictive method in the educational psychology field.

Psychology is taking advantage of the tree-based models in a broad set of applications (Baca-Garcia et al., 2007; Eloyan et al., 2012; Gibbons et al., 2013; Kuroki & Tilley, 2012; Scott, Jackson, & Bergeman, 2011; Skogli et al., 2013; Tian et al., 2014; van der Wal & Kowalczyk, 2013) . Within education, Blanch and Aluja (2013) , Cortes and Silva (2008) and Golino and Gomes (2014) applied the tree-based models to predict the academic achievement of students from the secondary and tertiary levels using a set of psychological and socio-demographic variables as predictors. The discussion of their methods and results are beyond the scope of the current paper, since we focused on the methodological aspects of machine learning, and how it can be applied in the educational psychology field.

In the present paper we showed the Rasch scores of the tests and scales used significantly differentiated the high achievement from the low achievement 10th grade students. Inductive reasoning presented a large effect size, while the deep learning approach, metacognitive control and self-appraisals presented moderate effect sizes. The random forest prediction lead to a total accuracy of 75%, a sensitivity of 68% and a specificity of 73.69% in the training set. The testing set result was a little bit worse, with a total accuracy of 68.18%, a sensitivity of 72.72% and a specificity of 63.63%. The most important variable in the prediction was the inductive reasoning that was associated with a mean decrease accuracy of 66.22% when left out of the prediction bag. The deep learning approach was the second most important variable (mean decrease accuracy of 28.45%), followed by metacognitive control (19.68%), TDRI self-appraisal (19.50%) and TCM self-appraisal (5.78%). This result reinforces previous findings that showed incremental validity of the learning approaches in the explanation of academic performance beyond intelligence, using traditional techniques (Chamorro-Premuzic & Furnham; 2008; Furnham Monsen, & Ahmetoglu, 2009; Gomes & Golino, 2012). It also reinforces the incremental validity of metacognition, over intelligence, in the explanation of academic achievement (van der Stel & Veenman, 2008; Veenman & Beishuizen, 2004).

CONCLUSION

The application of machine learning models in the prediction of academic achievement/performance, especially the tree-based models, represents an innovative complement to the traditional techniques such as linear and logistic regression, as well as structural equation modelling (Blanch & Aluja, 2013) . More than the advantages pointed earlier, the tree-based models can help us to understand the non-linear interactions between psycho- educational variables in the prediction of academic outcomes. These machine learning models not only represent an advance in terms of prediction accuracy, but also represent an advance in terms of inference. Future studies could benefit from employing a larger and broader sample, involving students from different schools. It would also be interesting to investigate, in the future, the impact of varying the tuning parameters of the random forest model in the accuracy, sensitivity, specificity and variability of the prediction.

ACKNOWLEDGEMENTS

Hudson F. Golino, Cristiano Mauro Assis Gomes, Diego Andrade (FAPEMIG) to the authors. The authors also receive grants provided by the Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira (INEP) and by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) of the Brazil’s Ministry of Science, Technology and Innovation.

REFERENCES

1. Baca-Garcia , E., Perez-Rodriguez, M., Saiz-Gonzalez, D., Basurte-Villamor, I., Saiz-Ruiz, J., Leiva-Murillo, J. M., & de Leon, J. (2007). Variables Associated with Familial Suicide Attempts in a Sample of Suicide Attempters. Progress in Neuro-Psychopharmacology & Biological Psychiatry, 31, 1312-1316. http://dx.doi.org/10.1016/j.pnpbp.2007.05.019
2. Blanch, A., & Aluja, A. (2013). A Regression Tree of the Aptitudes, Personality, and Academic Performance Relationship. Personality and Individual Differences, 54, 703-708. http://dx.doi.org/10.1016/j.paid.2012.11.032
3. Breiman , L. (2001a). Random Forests. Machine Learning, 1, 5-32. http://dx.doi.org/10.1023/A:1010933404324
4. Breiman, L. (2001b). Bagging Predictors. Machine Learning, 24, 123-140. http://dx.doi.org/10.1007/BF00058655
5. Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). Classification and Regression Trees. New York: Chapman & Hall.
6. Cohen, J. (1988). Statistical Power Analysis for the Behavioral Sciences (2nd ed.), Hillsdale, NJ: Lawrence Erlbaum Associates.
7. Commons, M. L., & Richards, F. A. (1984). Applying the General Stage Model. In M. L. Commons, F. A. Richards, & C. Armon (Eds.), Beyond Formal Operations. Late Adolescent and Adult Cognitive Development: Late Adolescent and Adult Cognitive Development (Vol. 1, pp. 141-157). New York: Praeger.
8. Commons, M. L. (2008). Introduction to the Model of Hierarchical Complexity and Its Relationship to Postformal Action. World Futures, 64, 305-320. http://dx.doi.org/10.1080/02604020802301105
9. Commons, M. L., & Pekker, A. (2008). Presenting the Formal Theory of Hierarchical Complexity. World Futures, 64, 375- 382. http://dx.doi.org/10.1080/02604020802301204
10. Cortez, P., & Silva, A. M. G. (2008). Using Data Mining to Predict Secondary School Student Performance. In A. Brito, & J. Teixeira (Eds.), Proceedings of 5th Annual Future Business Technology Conference, Porto, 5-12.
11. Del Re, A. C. (2013). compute.es: Compute Effect Sizes. R Package Version 0.2-2. http://cran.r-project.org/web/packages/compute.es

12. Eloyan, A., Muschelli, J., Nebel, M., Liu, H., Han, F., Zhao, T., Caffo, B. et al. (2012). Automated Diagnoses of Attention Deficit Hyperactive Disorder Using Magnetic Resonance Imaging. Frontiers in Systems Neuroscience, 6, 61. http://dx.doi.org/10.3389/fnsys.2012.00061
13. Fischer, K. W. (1980). A Theory of Cognitive Development: The Control and Construction of Hierarchies of Skills. Psychological Review, 87, 477-531. http://dx.doi.org/10.1037/0033-295X.87.6.477
14. Fischer, K. W., & Yan, Z. (2002). The Development of Dynamic Skill Theory. In R. Lickliter, & D. Lewkowicz (Eds.), Conceptions of Development: Lessons from the Laboratory. Hove: Psychology Press.
15. Flach, P. (2012). Machine Learning: The Art and Science of Algorithms That Make Sense of Data. Cambridge: Cambridge University Press. http://dx.doi.org/10.1017/CBO9780511973000
16. Frederick, S. (2005). Cognitive Reflection and Decision Making. Journal of Economic Perspectives, 19, 25-42. http://dx.doi.org/10.1257/089533005775196732
17. Geurts, P., Irrthum, A., & Wehenkel, L. (2009). Supervised Learning with Decision Tree-Based Methods in Computational and Systems Biology. Molecular BioSystems, 5, 1593-1605. http://dx.doi.org/10.1039/b907946g
18. Gibbons, R. D., Hooker, G., Finkelman, M. D., Weiss, D. J., Pilkonis, P. A., Frank, E., Moore, T., & Kupfer, D. J. (2013). The Computerized Adaptive Diagnostic Test for Major Depressive Disorder (CAD-MDD): A Screening Tool for Depression. Journal of Clinical Psychiatry, 74, 669-674. http://dx.doi.org/10.4088/JCP.12m08338
19. Golino, H. F., & Gomes, C. M. A. (2012). The Structural Validity of the Inductive Reasoning Developmental Test for the Measurement of Developmental Stages. In K. Stålne (Chair), Adult Development: Past, Present and New Agendas of Research, Symposium Conducted at the Meeting of the European Society for Research on Adult Development, Coimbra, 7-8 July 2012.
20. Golino, H. F., & Gomes, C. M. A. (2013). Controlando pensamentos intuitivos: O que o pão de queijo e o café podem dizer sobre a forma como pensamos. In C. M. A. Gomes (Chair), Neuroeconomia e Neuromarketing, Symposium conducted at the VII Simpósio de Neurociências da Universidade Federal de Minas Gerais, Belo Horizonte.

21. Golino, H. F., & Gomes, C. M. A. (2014). Four Machine Learning Methods to Predict Academic Achievement of College Students: A Comparison Study. Revista E-PSI, 4, 68-101.

22. Gomes, C. M. A., & Golino, H. F. (2009). Estudo exploratório sobre o Teste de Desenvolvimento do Raciocinio Indutivo (TDRI). In D. Colinvaux (Ed.), Anais do VII Congresso Brasileiro de Psicologia do Desenvolvimento: Desenvolvimento e Direitos Humananos (pp. 77-79). Rio de Janeiro: UERJ. http://www.abpd.psc.br/files/congressosAnteriores/AnaisVIICBPD.pdf

23. Gomes, C. M. A. (2010). Perfis de estudantes e a relação entre abordagens de aprendizagem e rendimento Escolar. Psico, 41, 503-509.

24. Gomes, C. M. A., & Golino, H. F. (2012). Validade incremental da Escala de Abordagens de Aprendizagem. Psicologia: Reflexão e Crítica, 25, 623-633. http://dx.doi.org/10.1590/S0102-79722012000400001

25. Gomes, C. M. A., Golino, H. F., Pinheiro, C. A. R., Miranda, G. R., & Soares, J. M. T. (2011). Validação da Escala de Abordagens de Aprendizagem (EABAP) em uma amostra brasileira. Psicologia: Reflexão e Crítica, 24, 19-27. http://dx.doi.org/10.1590/S0102-79722011000100004

26. Hardman , J., Paucar-Caceres, A., & Fielding, A. (2013). Predicting Students' Progression in Higher Education by Using the Random Forest Algorithm. Systems Research and Behavioral Science, 30, 194-203. http://dx.doi.org/10.1002/sres.2130

27. Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference and Prediction (2nd ed.). New York: Springer. http://dx.doi.org/10.1007/978-0-387-84858-7

28. James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An Introduction to Statistical Learning with Applications in R. New York: Springer. http://dx.doi.org/10.1007/978-1-4614-7138-7

29. Kuroki, Y., & Tilley, J. L. (2012). Recursive Partitioning Analysis of Lifetime Suicidal Behaviors in Asian Americans. Asian American Journal of Psychology, 3, 17-28. http://dx.doi.org/10.1037/a0026586

30. Liaw, A., & Wiener, M. (2012). Random Forest: Breiman and Cutler's Random Forests for Classification and Regression. R Package Version 4.6-7. http://cran.r-project.org/web/packages/randomForest/

31. Linacre, J. M. (2012). Winsteps® Rasch Measurement Computer Program. Beaverton, OR: Winsteps.com.

32. McGraw, K. O., & Wong, S. P. (1992). A Common Language Effect Size Statistic. Psychological Bulletin, 111, 361-365. http://dx.doi.org/10.1037/0033-2909.111.2.361

33. Scott, S. B., Jackson, B. R., & Bergeman, C. S. (2011). What Contributes to Perceived Stress in Later Life? A Recursive Partitioning Approach. Psychology and Aging, 26, 830-843. http://dx.doi.org/10.1037/a0023180

34. Skogli, E., Teicher, M. H., Andersen, P., Hovik, K., & Øie, M. (2013). ADHD in Girls and Boys—Gender Differences in Co-Existing Symptoms and Executive Function Measures. BMC Psychiatry, 13, 298. http://dx.doi.org/10.1186/1471-244X-13-298

35. Tian, F., Gao, P., Li, L., Zhang, W., Liang, H., Qian, Y., & Zhao, R. (2014). Recognizing and Regulating e-Learners' Emotions Based on Interactive Chinese Texts in e-Learning Systems. Knowledge-Based Systems, 55, 148-164. http://dx.doi.org/10.1016/j.knosys.2013.10.019

36. van der Wal, C., & Kowalczyk, W. (2013). Detecting Changing Emotions in Human Speech by Machine and Humans. Applied Intelligence, 39, 675-691. http://dx.doi.org/10.1007/s10489-013-0449-1

SECTION III
PYTHON LANGUAGE DETAILS

CHAPTER
8

A PYTHON 2.7 PROGRAMMING TUTORIAL

John W. Shipman

INTRODUCTION

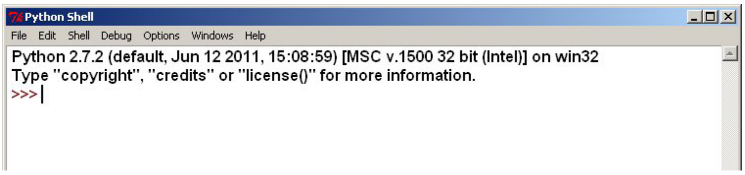
This document contains some tutorials for the Python programming language, as of Python version 2.7. These tutorials accompany the free Python classes taught by the New Mexico Tech Computer Center. Another good tutorial is at the Python website.

Starting Python in Conversational Mode

This tutorial makes heavy use of Python's conversational mode. When you start Python in this way, you will see an initial greeting message, followed by the prompt ">>>".

- On a TCC workstation in Windows, from the Start menu, select All Programs → Python 2.7 → IDLE (Python GUI). You will see something like this:

Citation: John W. Shipman, "A Python 2.7 Programming Tutorial", New Mexico Tech, 2007, online tutorial, accesses on April 25, 2018.



```
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

- For Linux or MacOS, from a shell prompt (such as "$" for the bash shell), type:
- python

You will see something like this:

```
$ python
Python 2.7.1 (r271:86832, Apr 12 2011, 16:15:16)
[GCC 4.6.0 20110331 (Red Hat 4.6.0-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

When you see the ">>>" prompt, you can type a Python expression, and Python will show you the result of that expression. This makes Python useful as a desk calculator. (That's why we sometimes refer to conversational mode as "calculator mode".) For example:

```
>>> 1+1
2
>>>
```

Each section of this tutorial introduces a group of related Python features.

PYTHON'S NUMERIC TYPES

Pretty much all programs need to do numeric calculations. Python has several ways of representing numbers, and an assortment of operators to operate on numbers.

Basic Numeric Operations

To do numeric calculations in Python, you can write expressions that look more or less like algebraic expressions in many other common languages. The "+" operator is addition; "-" is subtraction; use "*" to multiply; and use "/" to divide. Here are some examples:

```
>>> 99 + 1
100
>>> 1 - 99
-98
>>> 7 * 5
35
>>> 81 / 9
9
```

The examples in this document will often use a lot of extra space between the parts of the expression, just to make things easier to read. However, these spaces are not required:

```
>>> 99+1
100
>>> 1-99
-98
```

When an expression contains more than one operation, Python defines the usual order of operations, so that higher-precedence operations like multiplication and division are done before addition and subtraction. In this example, even though the multiplication comes after the addition, it is done first.

```
>>> 2 + 3 * 4
14
```

If you want to override the usual precedence of Python operators, use parentheses:

```
>>> (2+3)*4
20
```

Here's a result you may not expect:

```
>>> 1 / 5
0
```

You might expect a result of 0.2, not zero. However, Python has different kinds of numbers. Any number without a decimal point is considered an *integer*, a whole number. If any of the numbers involved contain a decimal point, the computation is done using *floating point* type:

```
>>> 1.0 / 4.0
0.25
>>> 1.0 / 3.0
0.33333333333333331
```

That second example above may also surprise you. Why is the last digit a one? In Python (and in pretty much all other contemporary programming languages), many real numbers cannot be represented exactly. The representation of 1.0/3.0 has a slight error in the seventeenth decimal place. This behavior may be slightly annoying, but in conversational mode, Python doesn't know how much precision you want, so you get a ridiculous amount of precision, and this shows up the fact that some values are approximations.

You can use Python's print statement to display values without quite so much precision:

```
>>> print 1.0/3.0
0.333333333333
```

It's okay to mix integer and floating point numbers in the same expression. Any integer values are coerced to their floating point equivalents.

```
>>> print 1.0/5
0.2
>>> print 1/5.0
0.2
```

Later we will learn about Python's string format method , which allows you to specify exactly how much precision to use when displaying numbers. For now, let's move on to some more of the operators on numbers.

The "%" operator between two numbers gives you the *modulo*. That is, the expression "*x* % *y*" returns the remainder when *x* is divided by *y*.

```
>>> 13 % 5
3
>>> 5.72 % 0.5
```

```
0.21999999999999975
>>> print 5.72 % 0.5
0.22
```

Exponentiation is expressed as "*x* ** *y*", meaning *x* to the *y* power.

```
>>> 2 ** 8
256
>>> 2 ** 30
1073741824
>>> 2.0 ** 0.5
1.4142135623730951
>>> 10.0 ** 5.2
158489.31924611141
>>> 2.0 ** 100
1.2676506002282294e+30
```

That last number, 1.2676506002282294e+30, is an example of *exponential* or *scientific* notation. This number is read as "1.26765... times ten to the 30th power". Similarly, a number like 1.66e-24 is read as "1.66 times ten to the minus 24th power".

So far we have seen examples of the integer type, which is called int in Python, and the floating-point type, called the float type in Python. Python guarantees that int type supports values between -2,147,483,648 and 2,147,483,647 (inclusive).

There is another type called long, that can represent much larger integer values. Python automatically switches to this type whenever an expression has values outside the range of int values. You will see letter "L" appear at the end of such values, but they act just like regular integers.

```
>>> 2 ** 50
1125899906842624L
>>> 2 ** 100
1267650600228229401496703205376L
>>> 2 ** 1000
107150860718626732094842504906000181056140481170553360744375
038837035105112
493612249319837881569585812759467291755314682518714528569231
```

```
404359845775746
985748039345677748242309854210746050623711418779541821530464
749835819412673
987675591655439460770629145711964776865421676604298316526243
868372056680693
76L
```

The Assignment Statement

So far we have worked only with numeric constants and operators. You can attach a name to a value, and that value will stay around for the rest of your conversational Python session.

Python names must start with a letter or the underbar (_) character; the rest of the name may consist of letters, underbars, or digits. Names are case-sensitive: the name Count is a different name than count.

For example, suppose you wanted to answer the question, "how many days is a million seconds?" We can start by attaching the name sec to a value of a million:

```
>>> sec = 1e6
>>> sec
1000000.0
```

A statement of this type is called an *assignment statement*. To compute the number of minutes in a million seconds, we divide by 60. To convert minutes to hours, we divide by 60 again. To convert hours to days, divide by 24, and that is the final answer.

```
>>> minutes = sec / 60.0
>>> minutes
16666.666666666668
>>> hours=minutes/60
>>> hours
277.77777777777777
>>> days=hours/24.
>>> days
11.574074074074074
>>> print days, hours, minutes, sec
```

```
11.5740740741 277.777777778 16666.6666667 1000000.0
```

You can attach more than one name to a value. Use a series of names, separated by equal signs, like this.

```
>>> total = remaining = 50
>>> print total, remaining
50 50
```

The general form of an assignment statement looks like this:

$name_1 = name_2 = ... = expression$

Here are the rules for evaluating an assignment statement:

- Each $name_i$ is some Python variable name. Variable names must start with either a letter or the underbar (_) character, and the remaining characters must be letters, digits, or underbar characters. Examples: skateKey; _x47; sum_of_all_fears.
- The *expression* is any Python expression.
- When the statement is evaluated, first the *expression* is evaluated so that it is a single value. For example, if the *expression* is "(2+3)*4", the resulting single value is the integer 20.

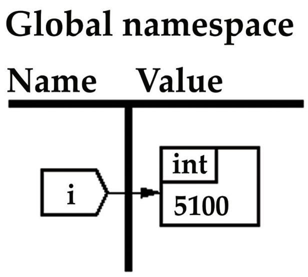
Then all the names $name_i$ are *bound* to that value.

What does it mean for a name to be bound to a value? When you are using Python in conversational mode, the names and value you define are stored in an area called the *global namespace*. This area is like a two-column table, with names on the left and values on the right.

Here is an example. Suppose you start with a brand new Python session, and type this line:

```
>>> i = 5100
```

Here is what the global namespace looks like after the execution of this assignment statement.

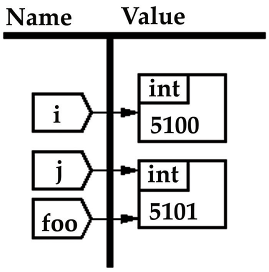


In this diagram, the value appearing on the right shows its type, int (integer), and the value, 5100.

In Python, values have types, but names are *not* associated with any type. A name can be bound to a value of any type at any time. So, a Python name is like a luggage tag: it identifies a value, and lets you retrieve it later.

Here is another assignment statement, and a diagram showing how the global namespace appears after the statement is executed.

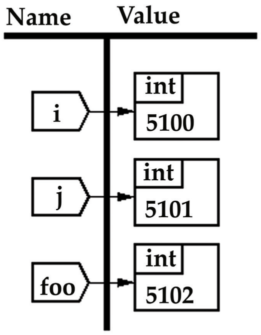
```
>>> j = foo = i + 1
```



The expression "i + 1" is equivalent to "5100 + 1", since variable i is bound to the integer 5100. This expression reduces to the integer value 5101, and then the names j and foo are both bound to that value. You might think of this situation as being like one piece of baggage with two tags tied to it.

Let's examine the global namespace after the execution of this assignment statement:

```
>>> foo = foo + 1
```



Because foo starts out bound to the integer value 5101, the expression "foo + 1" simplifies to the value 5102. Obviously, foo = foo + 1 doesn't make sense in algebra! However, it is a common way for programmers to add one to a value.

Note that name j is still bound to its old value, 5101.

More Mathematical Operations

Python has a number of built-in functions. To call a function in Python, use this general form:

$f(arg_1, arg_2, \ldots)$

That is, use the function name, followed by an open parenthesis "(", followed by zero or more *arguments* separated by commas, followed by a closing parenthesis ")".

For example, the round function takes one numeric argument, and returns the nearest whole number (as a float number). Examples:

```
>>> round ( 4.1 )
4.0
>>> round(4.9)
5.0
>>> round(4.5)
5.0
```

The result of that last case is somewhat arbitrary, since 4.5 is equidistant from 4.0 and 5.0. However, as in most other modern programming languages, the value chosen is the one further from zero. More examples:

```
>>> round (-4.1)
-4.0
>>> round (-4.9)
-5.0
>>> round (-4.5)
-5.0
```

For historical reasons, trigonometric and transcendental functions are not built-in to Python. If you want to do calculations of those kinds, you will need to tell Python that you want to use the math package. Type this line:

```
>>> from math import *
```

Once you have done this, you will be able to use a number of mathematical functions. For example, sqrt(x) computes the square root of x:

```
>>> sqrt(4.0)
2.0
>>> sqrt(81)
```

```
9.0
>>> sqrt(100000)
316.22776601683796
```

Importing the math module also adds two predefined variables, pi (as in π) and e, the base of natural logarithms:

```
>>> print pi, e
3.14159265359 2.71828182846
```

Here's an example of a function that takes more than argument. The function atan2(*dy* , *dx*) returns the arctangent of a line whose slope is *dy/dx*.

```
>>> atan2 ( 1.0, 0.0 )
1.5707963267948966
>>> atan2(0.0, 1.0)
0.0
>>> atan2(1.0, 1.0)
0.78539816339744828
>>> print pi/4
0.785398163397
```

For a complete list of all the facilities in the math module, see the *Python quick reference*. Here are some more examples; log is the natural logarithm, and log10 is the common logarithm:

```
>>> log(e)
1.0
>>> log10(e)
0.43429448190325182
>>> exp ( 1.0 )
2.7182818284590451
>>> sin ( pi / 2 )
1.0
>>> cos(pi/2)
6.1230317691118863e-17
```

Mathematically, cos(π/2) should be zero. However, like pretty much all other modern programming languages, transcendental functions like this use approximations. 6.12×10^{-17} is, after all, pretty close to zero.

Two math functions that you may find useful in certain situations:

- floor(*x*) returns the largest whole number that is less than or equal to *x*.
- ceil(*x*) returns the smallest whole number that is greater than or equal to *x*.

```
>>> floor(4.9)
4.0
>>> floor(4.1)
4.0
>>> floor(-4.1)
-5.0
>>> floor(-4.9)
-5.0
>>> ceil(4.9)
5.0
>>> ceil(4.1)
5.0
>>> ceil(-4.1)
-4.0
>>> ceil(-4.9)
-4.0
```

Note that the floor function always moves toward -∞ (minus infinity), and ceil always moves toward +∞.

CHARACTER STRING BASICS

Python has extensive features for handling strings of characters. There are two types:

- A str value is a string of zero or more 8-bit characters. The common characters you see on North American keyboards all use 8-bit characters. The official name for this character set is ASCII, for American Standard Code for Information Interchange.

This character set has one surprising property: all capital letters are considered less than all lowercase letters, so the string “Z” sorts before string “a”.

- A unicode value is a string of zero or more 32-bit Unicode characters. The Unicode character set covers just about every written language and almost every special character ever invented.

We’ll mainly talk about working with str values, but most unicode operations are similar or identical, except that Unicode literals are preceded with the letter u: for example, «abc” is type str, but u”abc” is type unicode.

String Literals

In Python, you can enclose string constants in either single-quote (‘...’) or double-quote (“...”) characters.

```
>>> cloneName = ‘Clem’
>>> cloneName
‘Clem’
>>> print cloneName
Clem
>>> fairName = “Future Fair”
>>> print fairName
Future Fair
>>> fairName
‘Future Fair’
```

When you display a string value in conversational mode, Python will usually use single-quote characters. Internally, the values are the same regardless of which kind of quotes you use. Note also that the print statement shows only the content of a string, without any quotes around it.

To convert an integer (int type) value *i* to its string equivalent, use the function “str(*i*)”:

```
>>> str(-497)
‘-497’
>>> str(000)
‘0’
```

The inverse operation, converting a string *s* back into an integer, is written as “int(*s*)”:

```
>>>
>>> int(“-497”)
-497
>>> int(“-0”)
0
>>> int ( “012this ain’t no number” )
    Traceback (most recent call last):
  File “<stdin>”, line 1, in ?
    ValueError: invalid literal for int(): 012this ain’t no number
```

The last example above shows what happens when you try to convert a string that isn’t a valid number.

To convert a string *s* containing a number in base *B*, use the form “int(*s*, *B*)”:

```
>>> int ( ‘0F’, 16 )
15
>>> int ( “10101”, 2 )
21
>>> int ( “0177776”, 8 )
65534
```

To obtain the 8-bit integer code contained in a one-character string *s*, use the function “ord(*s*)”. The inverse function, to convert an integer *i* to the character that has code *i*, use “chr(*i*)”. The numeric values of each character are defined by the ASCIIcharacter set.

```
>>> chr( 97 )
‘a’
>>> ord(“a”)
97
>>> chr(65)
‘A’
>>> ord(‘A’)
65
```

In addition to the printable characters with codes in the range from 32 to 127 inclusive, a Python string can contain any of the other unprintable, special characters as well. For example, the *null character*, whose official name is NUL, is the character whose code is zero. One way to write such a character is to use this form:

‘\x*NN*’

where *NN* is the character›s code in hexadecimal (base 16) notation.

```
>>> chr(0)
‘\x00’
>>> ord(‘\x00’)
0
```

Another special character you may need to deal with is the *newline* character, whose official name is LF (for “line feed”). Use the special *escape sequence* “\n” to produced this character.

```
>>> s = “Two-line\nstring.”
>>> s
‘Two-line\nstring.’
>>> print s
Two-line
string.
```

As you can see, when a newline character is displayed in conversational mode, it appears as “\n”, but when you print it, the character that follows it will appear on the next line. The code for this character is 10:

```
>>> ord(‘\n’)
10
>>> chr(10)
‘\n’
```

Python has several other of these escape sequences. The term “escape sequence” refers to a convention where a special character, the “escape character”, changes the meaning of the characters after it. Python’s escape character is backslash (\).

Input	**Code**	**Name**	**Meaning**
\b	8	BS	backspace

\t	9	HT	tab
\”	34	“	Double quote
\’	39	‘	Single quote
\\	92	\	Backslash

There is another handy way to get a string that contains newline characters: enclose the string within *three* pairs of quotes, either single or double quotes.

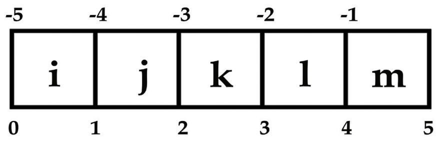
```
>>> multi = “””This string
...   contains three
...   lines.”””
>>> multi
‘This string\n  contains three\n  lines.’
>>> print multi
This string
  contains three
  lines.
>>> s2 = ‘’’
... xyz
... ‘’’
>>> s2
‘\nxyz\n’
>>> print s2

xyz

>>>
```

Notice that in Python’s conversational mode, when you press Enter at the end of a line, and Python knows that your line is not finished, it displays a “...” prompt instead of the usual “>>>” prompt.

Indexing Strings

To extract one or more characters from a string value, you have to know how positions in a string are numbered.

Here, for example, is a diagram showing all the positions of the string 'ijklm'.



In the diagram above, the numbers show the positions *between* characters. Position 0 is the position before the first character; position 1 is the position between the first and characters; and so on.

You may also refer to positions relative to the *end* of a string. Position -1 refers to the position before the last character; -2 is the position before the next-to-last character; and so on.

To extract from a string *s* the character that occurs just *after* position *n*, use an expression of this form:

s[*n*]

Examples:

```
>>> stuff = 'ijklm'
>>> stuff[0]
'i'
>>> stuff[1]
'j'
>>> stuff[4]
'm'
>>> stuff [ -1 ]
'm'
>>> stuff [-3]
'k'
>>> stuff[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    IndexError: string index out of range
```

The last example shows what happens when you specify a position after all the characters in the string.

You can also extract multiple characters from a string; see Section 4.3, "Slicing sequences".

String Methods

Many of the operations on strings are expressed as *methods*. A method is sort of like a function that lives only inside values of a certain type. To call a method, use this syntax:

expr.method(*arg*$_1$, *arg*$_2$, ...)

where each *arg*$_i$ is an argument to the method, just like an argument to a function.

For example, any string value has a method called center that produces a new string with the old value centered, using extra spaces to pad the value out to a given length. This method takes as an argument the desired new length. Here's an example:

```
>>> star = "*"
>>> star.center(5)
'  *  '
```

The following sections describe some of the more common and useful string methods.

.center(): Center Some Text

Given some string value *s*, to produce a new string containing *s* centered in a string of length *n*, use this method call:

s.center(*n*)

This method takes one argument *n*, the size of the result. Examples:

```
>>> k = "Ni"
>>> k.center(5)
'  Ni '
>>> "<*>".center(12)
'    <*>     '
```

Note that in the first example we are asking Python to center the string "Ni" in a field of length 5. Clearly we can't center a 2-character string in 5 characters, so Python arbitrarily adds the leftover space character before the old value.

.ljust() and .rjust(): Pad to Length on the Left or Right

Another useful string method left-justifies a value in a field of a given length. The general form:

s.ljust(*n*)

For any string expression *s*, this method returns a new string containing the characters from *s* with enough spaces added after it to make a new string of length *n*.

```
>>> "Ni".ljust(4)
'Ni  '
>>> "Too long to fit".ljust(4)
'Too long to fit'
```

Note that the .ljust() method will never return a shorter string. If the length isn't enough, it just returns the original value.

There is a similar method that right-justifies a string value:

s.rjust(*n*)

This method returns a string with enough spaces added before the value to make a string of length *n*. As with the .ljust() method, it will never return a string shorter than the original.

```
>>> "Ni".rjust(4)
'  Ni'
>>> m = "floccinaucinihilipilification"
>>> m.rjust(4)
'floccinaucinihilipilification'
```

.strip(), .lstrip(), and .rstrip(): Remove Leading and/or Trailing Whitespace

Sometimes you want to remove whitespace (spaces, tabs, and newlines) from a string. For a string *s*, use these methods to remove leading and trailing whitespace:

- *s*.strip() returns *s* with any leading or trailing whitespace characters removed.
- *s*.lstrip() removes only leading whitespace.
- *s*.rstrip() removes only trailing whitespace.

```
>>> spaceCase = ' \n \t Moon   \t\t '
>>> spaceCase
' \n \t Moon   \t\t '
```

```
>>> spaceCase.strip()
'Moon'
>>> spaceCase.lstrip()
'Moon   \t\t '
>>> spaceCase.rstrip()
' \n \t Moon'
```

.count(): How many occurrences?

The method *s*.count(*t*) searches string *s* to see how many times string *t* occurs in it.

```
>>> chiq = "banana"
>>> chiq
'banana'
>>> chiq.count("a")
3
>>> chiq.count("b")
1
>>> chiq.count("x")
0
>>> chiq.count("an")
2
>>> chiq.count("ana")
1
```

Note that this method does not count overlapping occurrences. Although the string "ana" occurs twice in string "banana", the occurrences overlap, so "banana".count("ana") returns only 1.

.find() and .rfind(): Locate a String within a Longer String

Use this method to search for a string *t* within a string *s*:

s.find(*t*)

If *t* matches any part of *s*, the method returns the position where the first match begins; otherwise, it returns -1.

```
>>> chiq
'banana'
>>> chiq.find ( "b" )
0
>>> chiq.find ( "a" )
1
>>> chiq.find ( "x" )
-1
>>> chiq.find ( "nan" )
2
```

If you need to find the *last* occurrence of a substring, use the similar method *s*.rfind(*t*), which returns the position where the last match starts, or -1 if there is no match.

```
>>> chiq.rfind("a")
5
>>> chiq[5]
'a'
>>> chiq.rfind("n")
4
>>> chiq.rfind("b")
0
>>> chiq.rfind("Waldo")
-1
```

.startswith() and .endswith()

You can check to see if a string *s* starts with a string *t* using a method call like this:

s.startswith(*t*)

This method returns True if *s* starts with a string that matches *t*; otherwise it returns False.

```
>>> chiq
'banana'
>>> chiq.startswith("b")
True
```

```
>>> chiq.startswith(“ban”)
True
>>> chiq.startswith ( ‘Waldo’ )
False
```

There is a similar method *s*.endswith(*t*) that tests whether string *s* ends with *t*:

```
>>> chiq.endswith(“Waldo”)
False
>>> chiq.endswith(“a”)
True
>>> chiq.endswith(“nana”)
True
```

The special values True and False are discussed later in Section 6.1, “Conditions and the bool type”.

.lower() and .upper(): Change the case of letters

The methods *s*.lower() and *s*.upper() are used to convert uppercase characters to lowercase, and vice versa, respectively.

```
>>> poet = ‘E. E. Cummings’
>>> poet.upper()
‘E. E. CUMMINGS’
>>> poet.lower()
‘e. e. cummings’
```

Predicates for testing for kinds of characters

Use the string methods in this section to test whether a string contains certain kinds of characters. Each of these methods is a *predicate*, that is, it asks a question and returns a value of True or False.

- *s*.isalpha() tests whether all the characters of *s* are letters.
- *s*.isupper() tests whether all the letters of *s* are uppercase. (It ignores any non-letter characters.)
- *s*.islower() tests whether all the letters of *s* are lowercase letters. (This method also ignores non-letter characters.)

- *s*.isdigit() tests whether all the characters of *s* are digits.

```
>>> mixed = 'abcDEFghi'
>>> mixed.isalpha()
True
>>> mixed.isupper()
False
>>> mixed.islower()
False
>>> "ABCDGOLDFISH".isupper()
True
>>> "lmno goldfish".islower()
True
>>> "abc $%&*(".islower()
True
>>> "abC $%&*(".islower()
False
>>> paradise = "87801"
>>> paradise.isalpha()
False
>>> paradise.isdigit()
True
>>> "abc123".isdigit()
False
```

.split(): Break fields out of a string

The .split() method is used to break a string up into pieces wherever a certain string called the *delimiter* is found; it returns a list of strings containing the text between the delimiters. For example, suppose you have a string that contains a series of numbers separated by whitespace. A call to the .split() method on that string, with no arguments, returns a list of the parts of the string that are surrounded by whitespace.

```
>>> line = "  1.4   8.6  -23.49   "
>>> line.split()
```

```
[‘1.4’, ‘8.6’, ‘-23.49’]
```

You can also specify a delimiter as the argument of the .split() method. Examples:

```
>>> s = ‘farcical/aquatic/ceremony’
>>> s.split(‘/’)
[‘farcical’, ‘aquatic’, ‘ceremony’]
>>> “//a/b/”.split(‘/’)
[‘’, ‘’, ‘a’, ‘b’, ‘’]
>>> “Stilton; Wensleydale; Cheddar;Edam”.split(“; “)
[‘Stilton’, ‘Wensleydale’, ‘Cheddar;Edam’]
```

You may also provide a second argument that limits the number of pieces to be split from the string.

```
>>> t = ‘a/b/c/d/e’
>>> t.split(‘/’)
[‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
>>> t.split(‘/’, 1)
[‘a’, ‘b/c/d/e’]
>>> t.split(‘/’, 3)
[‘a’, ‘b’, ‘c’, ‘d/e’]
```

The String Format Method

One of the commonest string operations is to combine fixed text and variable values into a single string. For example, maybe you have a variable named nBananas that contains the number of bananas, and you want to format a string something like «We have 27 bananas today». Here›s how you do it:

```
>>> nBananas = 54
>>> “We have {0} bananas today”.format(nBananas)
‘We have 54 bananas today’
```

Here is the general form of the string format operation:

S.format(p_0, p_1, ..., k_0=e_0, k_1=e_1, ...)

In this form:

- *S* is a format string that specifies the fixed parts of the desired text and also tells where the variable parts are to go and how they are to look.
- The .format() method takes zero or more positional arguments p_i followed by and zero or more keyword arguments k_i=e_i, where each k_i is any Python name and each e_i is any Python expression.
- The format string contains a mixture of ordinary text and *format codes*. Each of the format codes is enclosed in braces {...}. A format code containing a number refers to the corresponding positional argument, and a format code containing a name refers to the corresponding keyword argument.

Examples:

```
>>> “We have {0} bananas.”.format(27)
‘We have 27 bananas.’
>>> “We have {0} cases of {1} today.”.format(42, ‘peaches’)
‘We have 42 cases of peaches today.’
>>> “You’ll have {count} new {thing}s by {date}”.format(
...    count=27, date=”St. Swithin’s Day”, thing=”cooker”)
“You’ll have 27 new cookers by St. Swithin’s Day”
```

You can control the formatting of an item by using a format code of the form “{*N*:*type*}”, where *N* is the number or name of the argument to the .format() method, and *type* specifies the details of the formatting.

The *type* may be a single type code like s for string, d for integer, or f for float.

```
>>> “{0:d}”.format(27)
‘27’
>>> “{0:f}”.format(27)
‘27.000000’
>>> “{animal:s}”.format(animal=”sheep”)
‘sheep’
```

You may also include a field size just before the type code. With float values, you can also specify a precision after the field size by using a “.” followed by the desired number of digits after the decimal.

```
>>> "({bat:8s})".format(bat='fruit')
'(fruit   )'
>>> "{0:8f}".format(1.0/7.0)
'0.142857'
>>> "{n:20.11f}".format(n=1.0/7.0)
'       0.14285714286'
>>> "{silly:50.40f}".format(silly=5.33333)
'        5.3333300000000001261923898709937930107117'
```

Notice in the last example above that it is possible for you to produce any number of spurious digits beyond the precision used to specify the number originally! Beware, because those extra digits are utter garbage.

When you specify a precision, the value is rounded to the nearest value with that precision.

```
>>> "{0:.1f}".format(0.999)
'1.0'
>>> "{0:.1f}".format(0.99)
'1.0'
>>> "{0:.1f}".format(0.9)
'0.9'
>>> "{0:.1f}".format(0.96)
'1.0'
>>> "{0:.1f}".format(0.9501)
'1.0'
>>> "{0:.1f}".format(0.9499999)
'0.9'
```

The "e" type code forces exponential notation. You may also wish to use the "g" (for general) type code, which selects either float or exponential notation depending on the value.

```
>>> avo = 6.022e23
>>> "{0:e}".format(avo)
'6.022000e+23'
>>> "{0:.3e}".format(avo)
```

```
'6.022e+23'
>>> "{num:g}".format(num=144)
'144'
>>> "{num:g}".format(num=avo)
'6.022e+23'
```

By default, strings are left-justified within the field size and numbers are right-justified. You can change this by placing an alignment code just after the ":": "<" to left-align the field, "^" to center it, and ">" to right-align it.

```
>>> "/{0:<6s}/".format('git')
'/git   /'
>>> "/{0:^6s}/".format('git')
'/ git  /'
>>> "/{0:>6s}/".format('git')
'/   git/'
>>> '*{count:<8d}*'.format(count=13)
'*13      *'
```

Normally, short values are padded to length with spaces. You can specify a different padding character by placing it just after the ":".

```
>>> "{0:08d}".format(17)
'00000017'
"{film:@>20s}".format(film='If')
'@@@@@@@@@@@@@@@@@@If'
>>> "{film:@^20s}".format(film='If')
'@@@@@@@@@If@@@@@@@@@'
```

If you need to produce any "{" or "}" characters in the result, you must double them within the format code.

```
>>> "Set {0}: contents {{red, green, blue}}".format('glory')
'Set glory: contents {red, green, blue}'
```

One thing we sometimes need to is to format something to a size that is not known until the program is running. For example, suppose we want to format a ticket number from a variable named ticket_no, with left zero fill, and the width is given by a variable named how_wide. This would do the job:

```
>>> how_wide = 8
>>> ticket_no = 147
>>> “Ticket {num:0{w}d}”.format(num=ticket_no, w=how_wide)
‘Ticket 00000147’
```

Here, where the width is expected, “{w}” appears. Because there is a keyword argument that is effectively w=8, the value “8” is used for the width.

Note

The string .format() method has been available only since Python 2.6. If you are looking at older code, you may see a different technique using the “%” operator. For example, ‘Attila the %s’ % ‘Bun’ yields ‘Attila the bun’. For an explanation, see the Python library documentation. However, the old format operator is deprecated.

SEQUENCE TYPES

Mathematically, a *sequence* in Python represents an ordered set.

Sequences are an example of *container classes*: values that contain other values inside them.

Type name	Contains	Examples	Mutable?
str	8-bit characters	“abc” ‘abc’ “” ‘’ ‘\n’ ‘\x00’	No
unicode	32-bit characters	u’abc’ u’\u000c’	No
list	Any values	[23, “Ruth”, 69.8] []	Yes
tuple	Any values	(23, “Ruth”, 69.8) () (44,)	No

str and unicode are used to hold text, that is, strings of characters.

- list and tuple are used for sequences of zero or more values of any type. Use a list if the contents of the sequence may change; use a tuple if the contents will not change, and in certain places where tuples are required. For example, the right-hand argument of the string format operator (see Section 3.4, “The string format method”) must be a tuple if you are formatting more than one value.
- To create a list, use an expression of the form
- [*expr*$_1$, *expr*$_1$, ...]

with a list of zero or more values between *square brackets*, “[…]”.

- To create a tuple, use an expression of the form
- (*expr*$_1$, *expr*$_1$, ...)

with a list of zero or more values enclosed in *parentheses*, "(…")".

To create a tuple with only one element *v*, use the special syntax "(*v*,)". For example, (43+1,) is a one-element tuple containing the integer 44. The trailing comma is used to distinguish this case from the expression "(43+1)", which yields the integer 44, not a tuple.

- *Mutability*: You can't change *part* of an immutable value. For example, you can›t change the first character of a string from ‹a› to ‹b›. It is, however, easy to build a new string out of pieces of other strings.

Here are some calculator-mode examples. First, we'll create a string named s, a list named L, and a tuple named t:

```
>>> s = "abcde"
>>> L = [0, 1, 2, 3, 4, 5]
>>> t = ('x', 'y')
>>> s
'abcde'
>>> L
[0, 1, 2, 3, 4, 5]
>>> t
('x', 'y')
```

Functions and Operators for Sequences

The built-in function len(*S*) returns the number of elements in a sequence *S*.

```
>>> print len(s), len(L), len(t)
5 6 2
```

Function max(*S*) returns the largest value in a sequence *S*, and function min(*S*) returns the smallest value in a sequence *S*.

```
>>> max(L)
5
>>> min(L)
0
```

```
>>> max(s)
'e'
>>> min(s)
'a'
```

To test for set membership, use the “in” operator. For a value *v* and a sequence *S*, the expression *v* in *S* returns the Boolean value True if there is at least one element of *S* that equals *v*; it returns Falseotherwise.

```
>>> 2 in L
    True
>>> 77 in L
False
```

There is an inverse operator, *v* not in *S*, that returns True if *v* does not equal any element of *S*, False otherwise.

```
>>> 2 not in L
False
>>> 77 not in L
True
```

The “+” operator is used to concatenate two sequences of the same type.

```
>>> s + "xyz"
'abcdexyz'
>>> L + L
[0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5]
>>> t + ('z',)
('x', 'y', 'z')
```

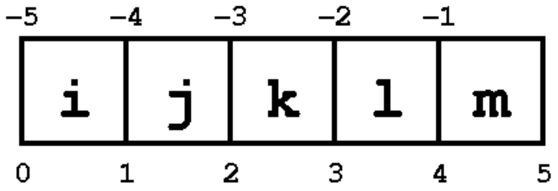
When the “*” operator occurs between a sequence *S* and an integer *n*, you get a new sequence containing *n* repetitions of the elements of *S*.

```
>>> "x" * 5
'xxxxx'
>>> "spam" * 8
'spamspamspamspamspamspamspamspam'
>>> [0, 1] * 3
[0, 1, 0, 1, 0, 1]
```

Indexing the Positions in a Sequence

Positions in a sequence refer to locations *between* the values. Positions are numbered from left to right starting at 0. You can also refer to positions in a sequence using negative numbers to count from right to left: position -1 is the position before the last element, position -2 is the position before the next-to-last element, and so on.

Here are all the positions of the string "ijklm".



To extract a single element from a sequence, use an expression of the form *S*[*i*], where *S* is a sequence, and *i* is an integer value that selects the element just *after* that position.

```
>>> s[0]
'a'
>>> s[4]
'e'
>>> s[5]
    Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    IndexError: string index out of range
```

The last line is an error; there is nothing after position 5 in string s.

```
>>> L[0]
0
>>> t[0]
'x'
```

Slicing Sequences

For a sequence *S*, and two positions *B* and *E* within that sequence, the expression *S* [*B* : *E*] produces a new sequence containing the elements of S between those two positions.

```
>>> L
```

```
[0, 1, 2, 3, 4, 5]
>>> L[2]
2
>>> L[4]
4
>>> L[2:4]
[2, 3]
>>> s = ‘abcde’
>>> s[2]
‘c’
>>> s[4]
‘e’
>>> s[2:4]
‘cd’
```

Note in the example above that the elements are selected from position 2 to position 4, which does *not* include the element L[4].

You may omit the starting position to slice elements from at the beginning of the sequence up to the specified position. You may omit the ending position to specify a slice that extends to the end of the sequence. You may even omit both in order to get a copy of the whole sequence.

```
>>> L[:4]
[0, 1, 2, 3]
>>> L[4:]
[4, 5]
>>> L[:]
[0, 1, 2, 3, 4, 5]
```

You can replace part of a list by using a slicing expression on the *left-hand* side of the “=” in an assignment statement, and providing a list of replacement elements on the right-hand side of the “=”. The elements selected by the slice are deleted and replaced by the elements from the right-hand side.

In slice assignment, it is not necessary that the number of replacement elements is the same as the number of replaced elements. In this example, the second and third elements of L are replaced by the five elements from the list on the right-hand side.

```
>>> L
[0, 1, 2, 3, 4, 5]
>>> L[2:4]
[2, 3]
>>> L[2:4] = [93, 94, 95, 96, 97]
>>> L
[0, 1, 93, 94, 95, 96, 97, 4, 5]
```

You can even delete a slice from a sequence by assigning an an empty sequence to a slice.

```
>>> L
[0, 1, 93, 94, 95, 96, 97, 4, 5]
>>> L[3]
94
>>> L[7]
4
>>> L[3:7] = []
>>> L
[0, 1, 93, 4, 5]
```

Similarly, you can insert elements into a sequence by using an empty slice on the left-hand side.

```
>>> L
[0, 1, 93, 4, 5]
>>> L[2]
93
>>> L[2:2] = [41, 43, 47, 53]
>>> L
[0, 1, 41, 43, 47, 53, 93, 4, 5]
```

Another way to delete elements from a sequence is to use Python’s del statement.

```
>>> L
```

```
[0, 1, 41, 43, 47, 53, 93, 4, 5]
>>> L[3:6]
[43, 47, 53]
>>> del L[3:6]
>>> L
[0, 1, 41, 93, 4, 5]
```

Sequence Methods

To find the position of a value *V* in a sequence *S*, use this method:

S.index(*V*)

This method returns the position of the first element of *S* that equals *V*. If no elements of *S* are equal, Python raises a ValueError exception.

```
>>> menu1
['beans', 'kale', 'tofu', 'trifle', 'sardines']
>>> menu1.index("kale")
1
>>> menu1.index("spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    ValueError: list.index(x): x not in list
```

The method *S*.count(*V*) method returns the number of elements of *S* that are equal to *V*.

```
>>> menu1[2:2] = ['spam'] * 3
>>> menu1
['beans', 'kale', 'spam', 'spam', 'spam', 'tofu', 'trifle', 'sardines']
>>> menu1.count('gravy')
0
>>> menu1.count('spam')
3
>>> "abracadabra".count("a")
5
>>> "abracadabra".count("ab")
```

```
2
>>> (1, 6, 55, 6, 55, 55, 8).count(55)
3
```

List Methods

All list instances have methods for changing the values in the list. These methods work only on lists. They do not work on the other sequence types that are not mutable, that is, the values they contain may not be changed, added, or deleted.

For example, for any list instance *L*, the .append(*v*) method appends a new value *v* to that list.

```
>>> menu1 = [‘kale’, ‘tofu’]
>>> menu1
[‘kale’, ‘tofu’]
>>> menu1.append ( ‘sardines’ )
>>> menu1
[‘kale’, ‘tofu’, ‘sardines’]
>>>
```

To insert a single new value *V* into a list *L* at an arbitrary position *P*, use this method:

L.insert(*P*, *V*)

To continue the example above:

```
>>> menu1
[‘kale’, ‘tofu’, ‘sardines’]
>>> menu1.insert(0, ‘beans’)
>>> menu1
[‘beans’, ‘kale’, ‘tofu’, ‘sardines’]
>>> menu1[3]
‘sardines’
>>> menu1.insert(3, ‘trifle’)
>>> menu1
[‘beans’, ‘kale’, ‘tofu’, ‘trifle’, ‘sardines’]
```

The method *L*.remove(*V*) removes the first element of *L* that equals *V*, if there is one. If no elements equal *V*, the method raises a ValueError exception.

```
>>> menu1
['beans', 'kale', 'spam', 'spam', 'spam', 'tofu', 'trifle', 'sardines']
>>> menu1.remove('spam')
>>> menu1
['beans', 'kale', 'spam', 'spam', 'tofu', 'trifle', 'sardines']
>>> menu1.remove('spam')
>>> menu1
['beans', 'kale', 'spam', 'tofu', 'trifle', 'sardines']
>>> menu1.remove('gravy')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

The *L*.sort() method sorts the elements of a list into ascending order.

```
>>> menu1
['beans', 'kale', 'spam', 'tofu', 'trifle', 'sardines']
>>> menu1.sort()
>>> menu1
['beans', 'kale', 'sardines', 'spam', 'tofu', 'trifle']
```

Note that the .sort() method itself does not return a value; it sorts the values of the list in place. A similar method is .reverse(), which reverses the elements in place:

```
>>> menu1
['beans', 'kale', 'sardines', 'spam', 'tofu', 'trifle']
>>> menu1.reverse()
>>> menu1
['trifle', 'tofu', 'spam', 'sardines', 'kale', 'beans']
```

The range() function: Creating Arithmetic Progressions

The term *arithmetic progression* refers to a sequence of numbers such that the difference between any two successive elements is the same. Examples: [1, 2, 3, 4, 5]; [10, 20, 30, 40]; [88, 77, 66, 55, 44, 33].

Python's built-in range() function returns a list containing an arithmetic progression. There are three different ways to call this function.

To generate the sequence [0, 1, 2, ..., *n*-1], use the form range(*n*).

```
>>> range(6)
[0, 1, 2, 3, 4, 5]
>>> range(2)
[0, 1]
>>> range(0)
[]
```

Note that the sequence will never include the value of the argument *n*; it stops one value short.

To generate a sequence [*i*, *i*+1, *i*+2, ..., *n*-1], use the form range(*i*, *n*):

```
>>> range(5,11)
[5, 6, 7, 8, 9, 10]
>>> range(1,5)
[1, 2, 3, 4]
```

To generate an arithmetic progression with a difference *d* between successive values, use the three-argument form range(*i*, *n*, *d*). The resulting sequence will be [*i*, *i*+*d*, *i*+2**d*, ...], and will stop before it reaches a value equal to *n*.

```
>>> range ( 10, 100, 10 )
[10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> range ( 100, 0, -10 )
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
>>> range ( 8, -1, -1 )
[8, 7, 6, 5, 4, 3, 2, 1, 0]
```

One Value can have Multiple Names

It is necessary to be careful when modifying mutable values such as lists because there may be more than one name bound to that value. Here is a demonstration.

We start by creating a list of two strings and binding two names to that list.

```
>>> menu1 = menu2 = ['kale', 'tofu']
>>> menu1
['kale', 'tofu']
>>> menu2
['kale', 'tofu']
```

Then we make a new list using a slice that selects all the elements of menu1:

```
>>> menu3 = menu1 [ : ]
>>> menu3
['kale', 'tofu']
```

Now watch what happens when we modify menu1's list:

```
>>> menu1.append ( 'sardines' )
>>> menu1
['kale', 'tofu', 'sardines']
>>> menu2
['kale', 'tofu', 'sardines']
>>> menu3
['kale', 'tofu']
```

If we appended a third string to menu1, why does that string also appear in list menu2? The answer lies in the definition of Python's assignment statement:

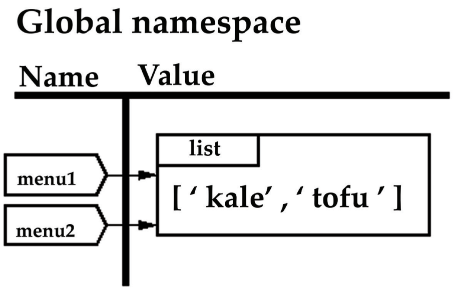
To evaluate an assignment statement of the form

$V_1 = V_2 = ... = expr$

where each V_i is a variable, and *expr* is some expression, first reduce *expr* to a single value, then bind each of the names v_i to that value.

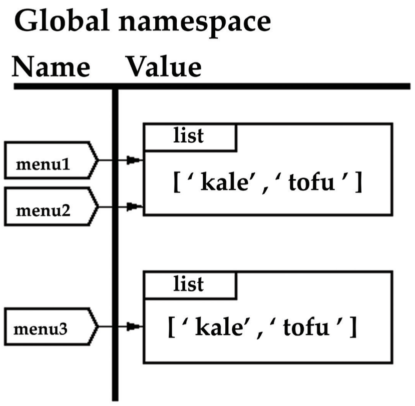
So let's follow the example one line at a time, and see what the global namespace looks like after each step. First we create a list instance and bind two names to it:

```
>>> menu1=menu2=['kale', 'tofu']
```



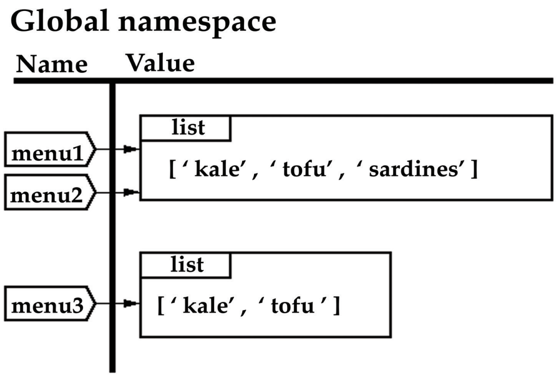
Two different names, menu1 and menu2, point to the same list. Next, we create an element-by-element copy of that list and bind the name menu3 to the copy.

```
>>> menu3 = menu1[:]
>>> menu3
['kale', 'tofu']
```



So, when we add a third string to menu1's list, the name menu2 is still bound to that same list.

```
>>> menu1.append ( 'sardines' )
>>> menu1
['kale', 'tofu', 'sardines']
>>> menu2
['kale', 'tofu', 'sardines']
```



This behavior is seldom a problem in practice, but it is important to keep in mind that two or more names can be bound to the same value.

If you are concerned about modifying a list when other names may be bound to the same list, you can always make a copy using the slicing expression “*L*[:]”.

```
>>> L1 = [‘bat’, ‘cat’]
>>> L2 = L1
>>> L3 = L1[:]
>>> L1.append(‘hat’)
>>> L2
[‘bat’, ‘cat’, ‘hat’]
>>> L3
[‘bat’, ‘cat’]
```

DICTIONARIES

Python’s dictionary type is useful for many applications involving table lookups. In mathematical terms:
A Python dictionary is a set of zero or more ordered pairs (*key*, *value*) such that:

- The *value* can be any type.
- Each *key* may occur only once in the dictionary.
- No *key* may be mutable. In particular, a key may not be a list or dictionary, or a tuple containing a list or dictionary, and so on.

The idea is that you store values in a dictionary associated with some key, so that later you can use that key to retrieve the associated value.

Operations on Dictionaries

The general form used to create a new dictionary in Python looks like this:

$\{k_1: v_1,\ k_2: v_2,\ ...\}$

To retrieve the value associated with key *k* from dictionary *d*, use an expression of this form:

d[*k*]

Here are some conversational examples:

```
>>> numberNames = {0:'zero', 1:'one', 10:'ten', 5:'five'}
>>> numberNames[10]
'ten'
>>> numberNames[0]
'zero'
>>> numberNames[999]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 999
```

Note that when you try to retrieve the value for which no key exists in the dictionary, Python raises a KeyError exception.

To add or replace the value for a key *k* in dictionary *d*, use an assignment statement of this form:

d[*k*] = *v*

For example:

```
>>> numberNames[2] = "two"
>>> numberNames[2]
'two'
>>> numberNames
{0: 'zero', 1: 'one', 10: 'ten', 2: 'two', 5: 'five'}
```

Note

The ordering of the pairs within a dictionary is undefined. Note that in the example above, the pairs do not appear in the order they were added.

You can use strings, as well as many other values, as keys:

```
>>> nameNo={“one”:1, “two”:2, “forty-leven”:4011}
>>> nameNo[“forty-leven”]
4011
```

You can test to see whether a key *k* exists in a dictionary *d* with the “in” operator, like this:

k in *d*

This operation returns True if *k* is a key in dictionary *d*, False otherwise.

The construct “*k* not in *d*” is the inverse test: it returns True if *k* is *not* a key in *d*, False if it *is* a key.

```
>>> 1 in numberNames
True
>>> 99 in numberNames
False
>>> “forty-leven” in nameNo
True
>>> “eleventeen” in nameNo
False
>>> “forty-leven” not in nameNo
False
>>> “eleventeen” not in nameNo
True
```

Python’s del (delete) statement can be used to remove a key-value pair from a dictionary.

```
>>> numberNames
{0: ‘zero’, 1: ‘one’, 10: ‘ten’, 2: ‘two’, 5: ‘five’}
>>> del numberNames[10]
>>> numberNames
{0: ‘zero’, 1: ‘one’, 2: ‘two’, 5: ‘five’}
```

```
>>> numberNames[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 10
```

Dictionary Methods

A number of useful methods are defined on any Python dictionary. To test whether a key *k* exists in a dictionary *d*, use this method:

d.has_key(*k*)

This is the equivalent of the expression "*k* in *d*": it returns True if the key is in the dictionary, False otherwise.

```
>>> numberNames
{0: 'zero', 1: 'one', 2: 'two', 5: 'five'}
>>> numberNames.has_key(2)
True
>>> numberNames.has_key(10)
False
```

To get a list of all the keys in a dictionary *d*, use this expression:

d.keys()

To get a list of the values in a dictionary *d* , use this expression:

d.values()

You can get all the keys and all the values at the same time with this expression, which returns a list of 2-element tuples, in which each tuple has one key and one value as (*k*, *v*).

d.items()

Examples:

```
>>> numberNames
{0: 'zero', 1: 'one', 2: 'two', 5: 'five'}
>>> numberNames.keys()
[0, 1, 2, 5]
>>> numberNames.values()
['zero', 'one', 'two', 'five']
>>> numberNames.items()
```

```
[(0, ‘zero’), (1, ‘one’), (2, ‘two’), (5, ‘five’)]
>>> nameNo
{‘forty-leven’: 4011, ‘two’: 2, ‘one’: 1}
>>> nameNo.keys()
[‘forty-leven’, ‘two’, ‘one’]
>>> nameNo.values()
[4011, 2, 1]
>>> nameNo.items()
[(‘forty-leven’, 4011), (‘two’, 2), (‘one’, 1)]
```

Here is another useful method:

d.get(*k*)

If *k* is a key in *d*, this method returns *d*[*k*]. However, if *k* is not a key, the method returns the special value None. The advantage of this method is that if the *k* is not a key in *d*, it is not considered an error.

```
>>> nameNo.get(“two”)
2
>>> nameNo.get(“eleventeen”)
>>> huh = nameNo.get(“eleventeen”)
>>> print huh
None
```

Note that when you are in conversational mode, and you type an expression that results in the value None, nothing is printed. However, the print statement will display the special value None visually as the example above shows.

There is another way to call the .get() method, with two arguments:

d.get(*k*, *default*)

In this form, if key *k* exists, the corresponding value is returned. However, if *k* is not a key in *d*, it returns the *default* value.

```
>>> nameNo.get(“two”, “I have no idea.”)
2
>>> nameNo.get(“eleventeen”, “I have no idea.”)
‘I have no idea.’
```

Here is another useful dictionary method. This is similar to the two-argument form of the .get() method, but it goes even further: if the key is not found, it stores a default value in the dictionary.

d.setdefault(*k*, *default*)

If key *k* exists in dictionary *d*, this expression returns the value *d*[*k*]. If *k* is not a key, it creates a new dictionary entry as if you had said "*d*[*k*] = *default*".

```
>>> nameNo.setdefault("two", "Unknown")
2
>>> nameNo["two"]
2
>>> nameNo.setdefault("three", "Unknown")
'Unknown'
>>> nameNo["three"]
'Unknown'
```

To merge two dictionaries *d1* and *d2*, use this method:

d1.update(*d2*)

This method adds all the key-value pairs from *d2* to *d1*. For any keys that exist in both dictionaries, the value after this operation will be the value from *d2*.

```
>>> colors = { 1: "red", 2: "green", 3: "blue" }
>>> moreColors = { 3: "puce", 4: "taupe", 5: "puce" }
>>> colors.update ( moreColors )
>>> colors
{1: 'red', 2: 'green', 3: 'puce', 4: 'taupe', 5: 'puce'}
```

Note in the example above that key 3 was in both dictionaries, but after the .update() method call, key 3 is related to the value from moreColors.

A Namespace is like a Dictionary

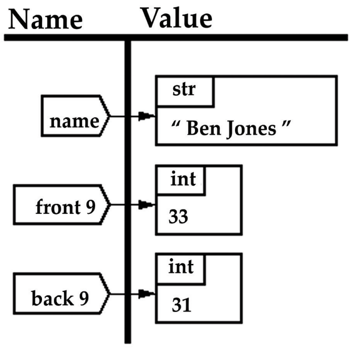
Back in Section 2.2, "The assignment statement", we first encountered the idea of a *namespace*. When you start up Python in conversational mode, the variables and functions you define live in the "global namespace".

We will see later on that Python has a number of different namespaces in addition to the global namespace. Keep in mind that namespaces are very similar to dictionaries:

- The names are like the keys of a dictionary: each one is unique.
- The values bound to those names are like the values in a dictionary. They can be any value of any type.

We can even use the same picture for a dictionary that we use to illustrate a namespace. Here is a small dictionary and a picture of it:

```
d = { 'name': 'Ben Jones', 'front9': 33, 'back9': 31 }
```



BRANCHING

By default, statements in Python are executed sequentially. *Branching* statements are used to break this sequential pattern.

- Sometimes you want to perform certain operations only in some cases. This is called a *conditional branch*.
- Sometimes you need to perform some operations repeatedly. This is called *looping*.

Before we look at how Python does conditional branching, we need to look at Python's Boolean type.

Conditions and the bool Type

Boolean algebra is the mathematics of true/false decisions. Python's bool type has only two values: True and False.

A typical use of Boolean algebra is in comparing two values. In Python, the expression $x < y$ is True if x is less than y, False otherwise.

```
>>> 2 < 5
True
>>> 2 < 2
False
```

```
>>> 2 < 0
False
```

Here are the six comparison operators:

Math symbol	**Python**	**Meaning**
<	<	Less than
≤	<=	Less than or equal to
>	>	Greater than
≥	>=	Greater than or equal to
≡	==	Equal to
≠	!=	Not equal to

The operator that compares for equality is "==". (The "=" symbol is not an operator: it is used only in the assignment statement.)

Here are some more examples:

```
>>> 2 <= 5
True
>>> 2 <= 2
True
>>> 2 <= 0
False
>>> 4.9 > 5
False
>>> 4.9 > 4.8
True
>>> (2-1)==1
True
>>> 4*3 != 12
False
```

Python has a function cmp(x, y) that compares two values and returns:

- Zero, if x and y are equal.
- A negative number if $x < y$.
- A positive number if $x > y$.

```
>>> cmp(2,5)
```

```
-1
>>> cmp(2,2)
0
>>> cmp(2,0)
1
```

The function bool(*x*) converts any value *x* to a Boolean value. The values in this list are considered False; any other value is considered True:

- Any numeric zero: 0, 0L, or 0.0.
- Any empty sequence: “” (an empty string), [] (an empty list), () (an empty tuple).
- {} (an empty dictionary).
- The special unique value None.

```
>>> print bool(0), bool(0L), bool(0.0), bool(‘’), bool([]), bool(())
False False False False False False
>>> print bool({}), bool(None)
False False
>>> print bool(1), bool(98.6), bool(‘Ni!’), bool([43, “hike”])
True True True True
```

The if Statement

The purpose of an if statement is to perform certain actions only in certain cases.

Here is the general form of a simple “one-branch” if statement. In this case, if some condition *C* is true, we want to execute some sequence of statements, but if *C* is not true, we don›t want to execute those statements.

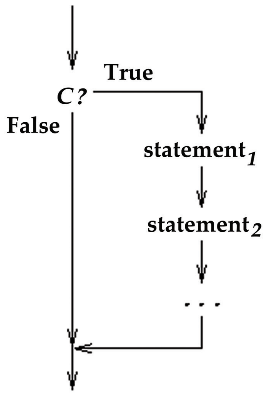
if *C*:

statement$_1$

statement$_2$

...

Here is a picture showing the flow of control through a simple if statement. Old-timers will recognize this as a *flowchart*.



There can be any number of statements after the if, but they must all be indented, and all indented the same amount. This group of statements is called a *block*.

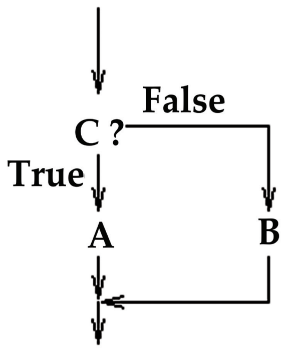
When the if statement is executed, the condition *C* is evaluated, and converted to a bool value (if it isn›t already Boolean). If that value is True, the block is executed; if the value is False, the block is skipped.

Here's an example:

```
>>> half = 0.5
>>> if half > 0:
...     print "Half is better than none."
...     print "Burma!"
...
    Half is better than none.
    Burma!
```

Sometimes you want to do some action *A* when C is true, but perform some different action *B* when C is false. The general form of this construct is:

```
if C:
    block A
    ...
else:
    block B
    ...
```



As with the single-branch if, the condition *C* is evaluated and converted to Boolean. If the result is True, *block A* is executed; if False, *block B* is executed instead.

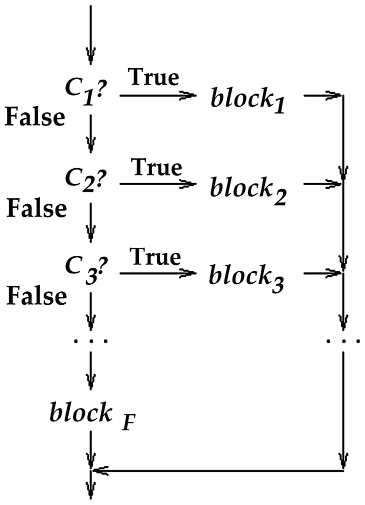
```
>>> half = 0.5
>>> if half > 0:
...     print "Half is more than none."
... else:
...     print "Half is not much."
...     print "Ni!"
...
    Half is more than none.
```

Some people prefer a more "horizontal" style of coding, where more items are put on the same line, so as to take up less vertical space. If you prefer, you can put one or more statements on the same line as the if or else, instead of placing them in an indented block. Use a semicolon ";" to separate multiple statements. For example, the above example could be expressed on only two lines:

```
>>> if half > 0: print "Half is more than none."
... else: print "Half is not much."; print "Ni!"
...
    Half is more than none.
```

Sometimes you want to execute only one out of three or four or more blocks, depending on several conditions. For this situation, Python allows you to have any number of "elif clauses" after an if, and before the else clause if there is one. Here is the most general form of a Python if statement:

```
if $C_1$:
    $block_1$
elif $C_2$:
    $block_2$
elif $C_3$:
    $block_3$
...
else:
    $block_F$
    ...
```



So, in general, an if statement can have zero or more elif clauses, optionally followed by an else clause. Example:

```
>>> i = 2
>>> if i==1: print "One"
... elif i==2: print "Two"
... elif i==3: print "Three"
... else: print "Many"
...
```

```
Two
```

You can have blocks within blocks. Here is an example:

```
>>> x = 3
>>> if  x >= 0:
...     if (x%2) == 0:
...         print “x is even”
...     else:
...         print “x is odd”
... else:
...     print “x is negative”
...
x is odd
```

A Word about Indenting Your Code

One of the most striking innovations of Python is the use of indentation to show the structure of the blocks of code, as in the if statement. Not everyone is thrilled by this feature. However, it is generally good practice to indent subsidiary clauses; it makes the code more readable. Those who argue that they should be allowed to violate this indenting practice are, in the author›s opinion, arguing against what is generally regarded as a good practice.

The amount by which you indent each level is a matter of personal preference. You can use a tab character for each level of indention; tab stops are assumed to be every 8th character. Beware mixing tabs with spaces, however; the resulting errors can be difficult to diagnose.

The for Statement: Looping

Use Python’s “for” construct to do some repetitive operation for each member of a sequence. Here is the general form:

```
for variable in sequence:
    block
    ...
```

$$\begin{tikzcd}
{} \arrow[d] \\
\textit{variable = sequence [0]} \arrow[d] \\
\textit{block} \arrow[d] \\
\cdots \arrow[d] \\
\textit{variable = sequence [1]} \arrow[d] \\
\textit{block} \arrow[d] \\
\cdots \arrow[d] \\
\textit{variable = sequence [-1]} \arrow[d] \\
\textit{block} \arrow[d] \\
{}
\end{tikzcd}$$

- The *sequence* can be any expression that evaluates to a sequence value, such as a list or tuple. The range() function is often used here to generate a sequence of integers.
- For each value in the *sequence* in turn, the *variable* is set to that value, and the *block* is executed.

As with the if statement, the block consists of one or more statements, indented the same amount relative to the if keyword.

This example prints the cubes of all numbers from 1 through 5.

```
>>> for n in range(1,6):
...     print “The cube of {0} is {1}”.format(n, n**3)
...
    The cube of 1 is 1
    The cube of 2 is 8
    The cube of 3 is 27
    The cube of 4 is 64
    The cube of 5 is 125
```

You may put the body of the loop—that is, the statements that will be executed once for each item in the sequence—on the same line as the “for” if you like. If there are multiple statements in the body, separate them with semicolons.

```
>>> for n in range(1,6): print “{0}**3={1}”.format(n, n**3),
...
1**3=1 2**3=8 3**3=27 4**3=64 5**3=125
>>> if 1 > 0: print “Yes”;print “One is still greater than zero”
...
Yes
```

One is still greater than zero

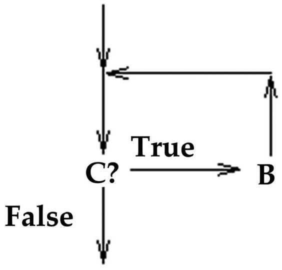
Here is an another example of iteration over a list of specific values.

```
>>> for  s in (‘a’, ‘e’, ‘i’, ‘o’, ‘u’):
...     word  =  “st” + s + “ck”
...     print  “Pick up the”, word
...
Pick up the stack
Pick up the steck
Pick up the stick
Pick up the stock
Pick up the stuck
```

The while Statement

Use this statement when you want to perform a block *B* as long as a condition *C* is true:

while *C*:
  B
  ...



Here is how a while statement is executed.

- Evaluate *C*. If the result is true, go to step 2. If it is false, the loop is done, and control passes to the statement after the end of *B*.
- Execute block *B*.
- Go back to step 1.

Here is an example of a simple while loop.

```
>>> i = 1
>>> while i < 100:
...     print i,
...     i = i * 2
...
1 2 4 8 16 32 64
```

This construct has the potential to turn into an *infinite loop*, that is, one that never terminates. Be sure that the body of the loop does something that will eventually make the loop terminate.

Special Branch Statements: break and continue

Sometimes you need to exit a for or while loop without waiting for the normal termination. There are two special Python branch statements that do this:

- If you execute a break statement anywhere inside a for or while loop, control passes out of the loop and on to the statement after the end of the loop.
- A continue statement inside a for loop transfers control back to the top of the loop, and the variable is set to the next value from the sequence if there is one. (If the loop was already using the last value of the sequence, the effect of continue is the same as break.)

Here are examples of those statements.

```
>>> i = 0
>>> while  i < 100:
...     i  =  i + 3
...     if  ( i % 5 ) == 0:
...         break
...     print i,
...
```

```
3 6 9 12
```

In the example above, when the value of i reaches 15, which has a remainder of 0 when divided by 5, the break statement exits the loop.

```
>>> for  i in range(500, -1, -1):
...     if (i % 100) != 0:
...         continue
...     print i,
...
500 400 300 200 100 0
```

HOW TO WRITE A SELF-EXECUTING PYTHON SCRIPT

So far we have used Python's conversational mode to demonstrate all the features. Now it's time to learn how to write a complete program.

Your program will live in a file called a *script*. To create your script, use your favorite text editor (emacs, vi, Notepad, whatever), and just type your Python statements into it.

How you make it executable depends on your operating system.

- On Windows platforms, be sure to give your script file a name that ends in ".py". If Python is installed, double-clicking on any script with this ending will use Python to run the script.
- Under Linux and MacOS X, the first line of your script must look like this:
- #!*pythonpath*

The *pythonpath* tells the operating system where to find Python. This path will usually be "/usr/local/bin/python", but you can use the "which" shell command to find the path on your computer:

```
$ which python
/usr/local/bin/python
```

Once you have created your script, you must also use this command to make it executable:

chmod +x *your-script-name*

Here is a complete script, set up for a typical Linux installation. This script, powersof2, prints a table showing the values of 2^n and 2^{-n} for n in the range

1, 2, ..., 12.

```
#!/usr/local/bin/python
print “Table of powers of two”
print
print “{0:>10s} {1:>2s} {2:s}”.format(“2**n”, “n”, “2**(-n)”)
for n in range(13):
    print “{0:10d} {1:2d} {2:17.15f}”.format(2**n, n, 2.0**(-n))
```

Here we see the invocation of this script under the bash shell, and the output:

```
$ ./powersof2
Table of powers of two

      2**n  n 2**(-n)
         1  0 1.000000000000000
         2  1 0.500000000000000
         4  2 0.250000000000000
         8  3 0.125000000000000
        16  4 0.062500000000000
        32  5 0.031250000000000
        64  6 0.015625000000000
       128  7 0.007812500000000
       256  8 0.003906250000000
       512  9 0.001953125000000
      1024 10 0.000976562500000
      2048 11 0.000488281250000
      4096 12 0.000244140625000
```

def: DEFINING FUNCTIONS

You can define your own functions in Python with the def statement.

- Python functions can act like mathematical functions such as len(s), which computes the length of s. In this example, values like s that are passed to the function are called *parameters* to the function.
- However, more generally, a Python function is just a container for some Python statements that do some task. A function can take any number of parameters, even zero.

Here is the general form of a Python function definition. It consists of a def statement, followed by an indented block called the *body* of the function.

def *name* (arg_0, arg_1, ...):
    block

The parameters that a function expects are called *arguments* inside the body of the function.

Here's an example of a function that takes no arguments at all, and does nothing but print some text.

```
>>> def pirateNoises():
...     for arrCount in range(7):
...         print "Arr!",
...
>>>
```

To call this function:

```
>>> pirateNoises()
Arr! Arr! Arr! Arr! Arr! Arr! Arr!
>>>
```

To call a function in general, use an expression of this form:

name ($param_0$, $param_1$, ...)

- The name of the function is followed by a left parenthesis "(", a list of zero or more *parameter* values separated by commas, then a right parenthesis ")".
- The parameter values are substituted for the corresponding arguments to the function. The value of parameter $param_0$ is substituted for argument arg_0; $param_1$ is substituted for arg_1 ; and so forth.

Here's a simple example showing argument substitution.

```
>>> def grocer(nFruits, fruitKind):
...     print "Stock: {0} cases of {1}".format(nFruits, fruitKind)
...
>>> grocer ( 37, 'kale' )
Stock: 37 cases of kale
>>> grocer(0,"bananas")
```

Stock: 0 cases of bananas

return: Returning Values from a Function

So far we have seen some simple functions that take arguments or don't take arguments. How do we define functions like len() that return a value?

Anywhere in the body of your function, you can write a return statement that terminates execution of the function and returns to the statement where it was called.

Here is the general form of this statement:

return *expression*

The *expression* is evaluated, and its value is returned to the caller.

Here is an example of a function that returns a value:

```
>>> def square(x):
...     return x**2
...
>>> square(9)
81
>>> square(2.5)
6.25
>>>
```

- You can omit the *expression*, and just use a statement of this form:
- return

In this case, the special placeholder value None is returned.

- If Python executes your function body and never encounters a return statement, the effect is the same as a return with no value: the special value None is returned.

Here is another example of a function that returns a value. This function computes the factorial of a positive integer:

The factorial of *n*, denoted *n*!, is defined as the product of all the integers from 1 to *n* inclusive.

For example, 4! = 1×2×3×4 = 24.

We can define the factorial function *recursively* like this:

- If *n* is 0 or 1, *n*! is 1.
- If *n* is greater than 1, *n*! = n × (n-1)!.

And here is a recursive Python function that computes the factorial, and a few examples of its use.

```
>>> def fact(n):
...     if n <= 1:
```

```
...         return 1
...     else:
...         return n * fact(n-1)
...
>>> for i in range(5):
...     print i, fact(i)
...
0 1
1 1
2 2
3 6
4 24
>>> fact(44)
2658271574788448768043625811014615890319638528000000000L
>>>
```

Function Argument List Features

The general form of a def shown in Section 8, "def: Defining functions" is over-simplified. In general, the argument list of a function is a sequence of four kinds of arguments:

1. If the argument is just a name, it is called a *positional* argument. There can be any number of positional arguments, including zero.
2. You can supply a default value for the argument by using the form "*name=value*". Such arguments are called *keyword* arguments. See Section 8.3, "Keyword arguments".

A function can have any number of keyword arguments, including zero.

All keyword arguments must follow any positional arguments in the argument list.

3. Sometimes it is convenient to write a function that can accept any number of positional arguments. To do this, use an argument of this form:
4. * *name*

A function may have only one such argument, and it must follow any positional or keyword arguments. For more information about this feature, see Section 8.4, "Extra positional arguments".

5. Sometimes it is also convenient to write a function that can accept any number of keyword arguments, not just the specific keyword arguments. To do this, use an argument of this form:

6. ** *name*

If a function has an argument of this form, it must be the last item in the argument list. For more information about this feature, see Section 8.5, “Extra keyword arguments”.

Keyword Arguments

If you want to make some of the arguments to your function optional, you must supply a default value. In the argument list, this looks like “*name=value*”.

Here’s an example of a function with one argument that has a default value. If you call it with no arguments, the name mood has the string value ‘bleah’ inside the function. If you call it with an argument, the name mood has the value you supply.

```
>>> def report(mood=’bleah’):
...     print “My mood today is”, mood
...
>>> report()
My mood today is bleah
>>> report(‘hyper’)
My mood today is hyper
>>>
```

If your function has multiple arguments, and the caller supplies multiple parameters, here is how they are matched up:

- The function call must supply at least as many parameters as the function has positional arguments.
- If the caller supplies more positional parameters than the function has positional arguments, parameters are matched with keyword arguments according to their position.

Here are some examples showing how this works.

```
>>> def f(a, b=”green”, c=3.5):
...     print a, b, c
...
>>> f()
Traceback (most recent call last):
  File “<stdin>”, line 1, in ?
TypeError: f() takes at least 1 argument (0 given)
>>> f(47)
47 green 3.5
>>> f(47, 48)
```

```
47 48 3.5
>>> f(47, 48, 49)
47 48 49
>>> f(47, 48, 49, 50)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: f() takes at most 3 arguments (4 given)
>>>
```

Here is another feature: the caller of a function can supply what are called *keyword parameters* of the form "*name*=*value*". If the function has an argument with a matching keyword, that argument will be set to *value*.

- If a function's caller supplies both positional and keyword parameters, all positional parameters must precede all keyword parameters.
- Keyword parameters may occur in any order.

Here are some examples of calling a function with keyword parameters.

```
>>> def g(p0, p1, k0="K-0", k1="K-1"):
...     print p0, p1, k0, k1
...
>>> g(33,44)
33 44 K-0 K-1
>>> g(33,44,"K-9","beep")
33 44 K-9 beep
>>> g(55,66,k1="whirr")
55 66 K-0 whirr
>>> g(7,8,k0="click",k1="clank")
7 8 click clank
>>>
```

Extra Positional Arguments

You can declare your function in such a way that it will accept any number of positional parameters. To do this, use an argument of the form "**name*" in your argument list.

- If you use this special argument, it must follow all the positional and keyword arguments in the list.
- When the function is called, this name will be bound to a tuple containing any positional parameters that the caller supplied, over and above parameters that corresponded to other parameters.

Here is an example of such a function.

```
>>> def h(i, j=99, *extras):
...     print i, j, extras
...
>>> h(0)
0 99 ()
>>> h(1,2)
1 2 ()
>>> h(3,4,5,6,7,8,9)
3 4 (5, 6, 7, 8, 9)
>>>
```

Extra Keyword Arguments

You can declare your function in such a way that it can accept any number of keyword parameters, in addition to any keyword arguments you declare.

To do this, place an argument of the form "***name*" last in your argument list.

When the function is called, that name is bound to a dictionary that contains any keyword-type parameters that are passed in that have names that don't match your function's keyword-type arguments. In that dictionary, the keys are the names used by the caller, and the values are the values that the caller passed.

Here's an example.

```
>>> def k(p0, p1, nickname='Noman', *extras, **extraKeys):
...     print p0, p1, nickname, extras, extraKeys
...
>>> k(1,2,3)
1 2 3 () {}
>>> k(4,5)
4 5 Noman () {}
>>> k(6, 7, hobby='sleeping', nickname='Sleepy', hatColor='green')
6 7 Sleepy () {'hatColor': 'green', 'hobby': 'sleeping'}
>>> k(33, 44, 55, 66, 77, hometown='McDonald', eyes='purple')
33 44 55 (66, 77) {'hometown': 'McDonald', 'eyes': 'purple'}
>>>
```

Documenting Function Interfaces

Python has a preferred way to document the purpose and usage of your functions. If the first line of a function body is a string constant, that string constant is saved along with the function as the *documentation string*. This string can be retrieved by using an expression of the form *f*.__doc__, where *f* is the function name.

Here's an example of a function with a documentation string.

```
>>> def pythag(a, b):
...     """Returns the hypotenuse of a right triangle with sides a and b.
...     """
...     return (a*a + b*b)**0.5
...
>>> pythag(3,4)
5.0
>>> pythag(1,1)
1.4142135623730951
>>> print pythag.__doc__
    Returns the hypotenuse of a right triangle with sides a and b.

>>>
```

USING PYTHON MODULES

Once you start building programs that are more than a few lines long, it's critical to apply this overarching principle to programming design:

Important

Divide and conquer.

In other words, rather than build your program as one large blob of Python statements, divide it into logical pieces, and divide the pieces into smaller pieces, until the pieces are each small enough to understand.

Python has many tools to help you divide and conquer. In Section 8, "def: Defining functions", we learned how to package up a group of statements into a function, and how to call that function and retrieve the result.

Way back in Section 2.3, "More mathematical operations", we got our first look at another important tool, Python's *module* system. Python does not have a built-in function to compute square roots, but there is a built-in module called math that includes a function sqrt() that computes square roots.

In general, a module is a package of functions and variables that you can import and use in your programs. Python comes with a large variety of modules, and you can also create your own. Let's look at Python's module system in detail.

- In Section 9.1, "Importing items from modules", we learn to import items from existing modules.
- Section 9.2, "Import entire modules" shows another way to use items from modules.
- Section 9.4, "Build your own modules".

Importing Items from Modules

Back in Section 2.2, "The assignment statement", we learned that there is an area called the "global namespace," where Python keeps the names and values of the variables you define.

The Python dir() function returns a list of all the names that are currently defined in the global namespace. Here is a conversational example; suppose you have just started up Python in conversational mode.

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> frankness = 0.79
>>> dir()
['__builtins__', '__doc__', '__name__', 'frankness']
>>> def oi():
...     print "Oi!"
...
>>> dir()
['__builtins__', '__doc__', '__name__', 'frankness', 'oi']
>>> type(frankness)
<type 'float'>
```

```
>>> type(oi)
<type 'function'>
>>>
```

When Python starts up, three variables are always defined: __builtins__, __doc__, and __name__. These variables are for advanced work and needn't concern us now.

Note that when we define a variable (frankness), next time we call dir(), that name is in the resulting list. When we define a function (oi), its name is also added. Note also that you can use the type()function to find the type of any currently defined name: frankness has type float, and oi has type function.

Now let's see what happens when we import the contents of the math module into the global namespace:

```
>>> from math import *
>>> dir()
['__builtins__', '__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frankness', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'oi', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>> sqrt(64)
8.0
>>> pi*10.0
31.415926535897931
>>> cos(0.0)
1.0
>>>
```

As you can see, the names we have defined (oi and frankness) are still there, but all of the variables and functions from the math module are now in the namespace, and we can use its functions and variables like sqrt() and pi.

In general, an import statement of this form copies all the functions and variables from the module into the current namespace:

from *someModule* import *

However, you can also be selective about which items you want to import. Use a statement of this form:

from *someModule* import *item*$_1$, *item*$_2$, ...

where the keyword import is followed by a list of names, separated by commas.

Here's another example. Assume that you have just started a brand new Python session, and you want to import only the sqrt() function and the constant pi:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> from math import sqrt, pi
>>> dir()
['__builtins__', '__doc__', '__name__', 'pi', 'sqrt']
>>> sqrt(25.0)
5.0
>>> cos(0.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
>>>
```

We didn't ask for the cos() function to be imported, so it is not part of the global namespace.

Import Entire Modules

Some modules have hundreds of different items in them. In cases like that, you might not want to clutter up your global namespace with all those items. There is another way to import a module. Here is the general form:

import *moduleName*

This statement adds only one name to the current namespace—the name of the module itself. You can then refer to any item inside that module using an expression of this form:

moduleName.itemName

Here is an example, again using the built-in math module. Assume that you have just started up a new Python session and you have added nothing to the namespace yet.

```
>>> dir()
['__builtins__', '__doc__', '__name__']
```

```
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', 'math']
>>> type(math)
<type 'module'>
>>> math.sqrt(121.0)
11.0
>>> math.pi
3.1415926535897931
>>> math.cos(0.0)
1.0
>>>
```

As you can see, using this form of import adds only one name to the namespace, and that name has type module.

There is one more additional feature of import we should mention. If you want to import an entire module M_1, but you want to refer to its contents using a different name M_2, use a statement of this form:

import M_1 as M_2

An example:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> import math as crunch
>>> dir()
['__builtins__', '__doc__', '__name__', 'crunch']
>>> type(crunch)
<type 'module'>
>>> crunch.pi
3.1415926535897931
>>> crunch.sqrt(888.888)
29.81422479287362
>>>
```

You can apply Python's built-in dir() function to a module object to find out what names are defined inside it:

```
>>> import math
>>> dir()
```

```
['__builtins__', '__doc__', '__name__', 'math']
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil',
'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot',
'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'tan', 'tanh']
>>>
```

A Module is a Namespace

Modules are yet another example of a Python namespace, just as we've discussed in Section 2.2, "The assignment statement" and Section 5.3, "A namespace is like a dictionary".

When you import a module using the form "import *moduleName*", you can refer to some name *N* inside that module using the period operator: "*moduleName.N*".

So, like any other namespace, a module is a container for a unique set of names, and the values to which each name is connected.

Build Your Own Modules

If you have a common problem to solve, chances are very good that there are modules already written that will reduce the amount of code you have to write.

- Python comes with a large collection of built-in modules. See the *Python Library Reference*.
- The python.org site also hosts a collection of thousands of third-party modules: see the *Python package index*.

You can also build your own modules. A module is similar to a script (see Section 7, "How to write a self-executing Python script"): it is basically a text file containing the definitions of Python functions and variables.

To build your own module, use a common text editor to create a file with a name of the form "*moduleName*.py". The *moduleName* you choose must be a valid Python name—it must start with a letter or underbar, and consist entirely of letters, underbars, and digits.

Inside that file, place Python function definitions and ordinary assignment statements.

Here is a very simple module containing one function and one variable. It lives in a file named cuber.py.

```
def cube(x):
    return x**3

cubeVersion = “1.9.33”
```

Here is an example interactive session that uses that module:

```
>>> dir()
[‘__builtins__’, ‘__doc__’, ‘__name__’]
>>> from cuber import *
>>> dir()
[‘__builtins__’, ‘__doc__’, ‘__name__’, ‘cube’, ‘cubeVersion’]
>>> cube(3)
27
>>> cubeVersion
‘1.9.33’
>>>
```

There is one more refinement we suggest for documenting the contents of a module. If the first line of the module’s file is a string constant, it is saved as the module’s “documentation string.” If you later import such a module using the form “import *moduleName*”, you can retrieve the contents of the documentation string using the expression “*moduleName*.__doc__”.

Here is an expanded version of our cuber.py with a documentation string:

```
“””cuber.py:  Simple homemade Python module

  Contents:
    cube(x):  Returns the cube of x
    cubeVersion:  Current version number of this module
“””
def cube(x):
    return x**3

cubeVersion = “1.9.33”
```

Finally, an example of how to retrieve the documentation string:

```
>>> import cuber
>>> print cuber.__doc__
```

```
cuber.py:  Simple homemade Python module

  Contents:
    cube(x):  Returns the cube of x
    cubeVersion:  Current version number of this module

>>> cuber.cube(10)
1000
>>>
```

INPUT AND OUTPUT

Python makes it easy to read and write files. To work with a file, you must first open it using the built-in open() function. If you are going to read the file, use the form "open(*filename*)", which returns a *file object*. Once you have a file object, you can use a variety of methods to perform operations on the file.

Reading Files

For example, for a file object *F*, the method *F*.readline() attempts to read and return the next line from that file. If there are no lines remaining, it returns an empty string.

Let's start with a small text file named trees containing just three lines:

```
yew
oak
alligator juniper
```

Suppose this file lives in your current directory. Here is how you might read it one line at a time:

```
>>> treeFile = open ( 'trees' )
>>> treeFile.readline()
'yew\n'
>>> treeFile.readline()
'oak\n'
>>> treeFile.readline()
'alligator juniper\n'
>>> treeFile.readline()
```

```
‘’
```

Note that the newline characters (‘\n’) are included in the return value. You can use the string .rstrip() method to remove trailing newlines, but beware: it also removes any other trailing whitespace.

```
>>> ‘alligator juniper\n’.rstrip()
‘alligator juniper’
>>> ‘eat all my trailing spaces        \n’.rstrip()
‘eat all my trailing spaces’
```

To read all the lines in a file at once, use the .readlines() method. This returns a list whose elements are strings, one per line.

```
>>> treeFile=open(“trees”)
>>> treeFile.readlines()
[‘yew\n’, ‘oak\n’, ‘alligator juniper\n’]
```

A more general method for reading files is the .read() method. Used without any arguments, it reads the entire file and returns it to you as one string.

```
>>> treeFile = open (“trees”)
>>> treeFile.read()
‘yew\noak\nalligator juniper\n’
```

To read exactly *N* characters from a file *F*, use the method *F*.read(*N*). If *N* characters remain in the file, you will get them back as an *N*-character string. If fewer than *N* characters remain, you will get the remaining characters in the file (if any).

```
>>> treeFile = open ( “trees” )
>>> treeFile.read(1)
‘y’
>>> treeFile.read(5)
‘ew\noa’
>>> treeFile.read(50)
‘k\nalligator juniper\n’
>>> treeFile.read(80)
‘’
```

One of the easiest ways to read the lines from a file is to use a for statement. Here is an example:

```
>>> >>> treeFile=open('trees')
>>> for treeLine in treeFile:
...     print treeLine.rstrip()
...
yew
oak
alligator juniper
```

As with the .readline() method, when you iterate over the lines of a file in this way, the lines will contain the newline characters. If the above example did not trim these lines with .rstrip(), each line of output would be followed by a blank line, because the print statement adds a newline.

File Positioning for Random-access Devices

For random-access devices such as disk files, there are methods that let you find your current position within a file, and move to a different position.

- *F*.tell() returns your current position in file *F*.
- *F*.seek(*N*) moves your current position to *N*, where a position of zero is the beginning of the file.
- *F*.seek(*N*, 1) moves your current position by a distance of *N* characters, where positive values of *N* move toward the end of the file and negative values move toward the beginning.

For example, *F*.seek(80, 1) would move the file position 80 characters further from the start of the file.

- *F*.seek(*N*, 2) moves to a position *N* characters relative to the end of the file. For example, *F*.seek(0, 2) would move to the end of the file; *F*.seek(-200, 2) would move your position to 200 bytes before the end of the file.

```
>>> treeFile = open ( "trees" )
>>> treeFile.tell()
0L
>>> treeFile.read(6)
'yew\noa'
>>> treeFile.tell()
6L
>>> treeFile.seek(1)
>>> treeFile.tell()
```

```
1L
>>> treeFile.read(5)
'ew\noa'
>>> treeFile.tell()
6L
>>> treeFile.seek(1, 1)
>>> treeFile.tell()
7L
>>> treeFile.seek(-3, 1)
>>> treeFile.tell()
4L
>>> treeFile.seek(0, 2)
>>> treeFile.tell()
26L
>>> treeFile.seek(-3, 2)
>>> treeFile.tell()
23L
>>> treeFile.read()
'er\n'
```

Writing Files

To create a disk file, open the file using a statement of this general form:

F = open (*filename*, "w")

The second argument, "w", specifies write access. If possible, Python will create a new, empty file by that name. If there is an existing file by that name, and if you have write permission to it, the existing file will be deleted.

To write some content to the file you are creating, use this method:

F.write(*s*)

where *s* is any string expression.

Warning

The data you have sent to a file with the .write() method may not actually appear in the disk file until you close it by calling the .close() method on the file.

This is due to a mechanism called *buffering*. Python accumulates the data you have sent to the file, until a certain amount is present, and then it "flushes" that data to the physical file by writing it. Python also flushes the data to the file when you close it.

If you would like to make sure that the data you have written to the file is actually physically present in the file without closing it, call the .flush() method on the file object.

```
>>> sports = open ( “sportfile”, “w” )
>>> sports.write ( “tennis\nrugby\nquoits\n” )
>>> sports.close()
>>> sportFile = open ( “sportfile” )
>>> sportFile.readline()
‘tennis\n’
>>> sportFile.readline()
‘rugby\n’
>>> sportFile.readline()
‘quoits\n’
>>> sportFile.readline()
‘’
```

Here is a lengthy example demonstrating the action of the .flush() method.

```
>>> sporting = open(‘sports’, ‘w’)
>>> sporting.write(‘golf\n’)
>>> echo = open(‘sports’)
>>> echo.read()
‘’
>>> echo.close()
>>> sporting.flush()
>>> echo = open(‘sports’)
>>> echo.read()
‘golf\n’
>>> echo.close()
>>> sporting.write(‘soccer’)
>>> sporting.close()
>>> open(‘sports’).read()
‘golf\nsoccer’
```

Note that you must explicitly provide newline characters in the arguments to .write().

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

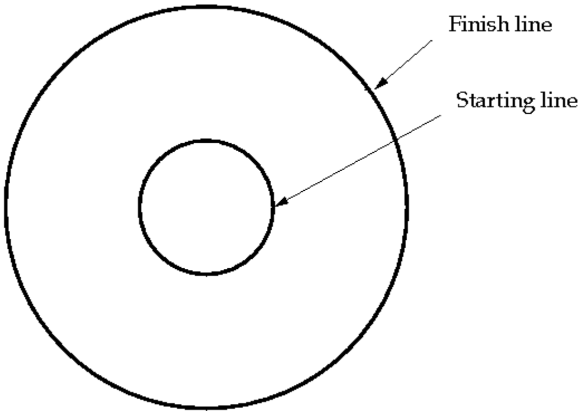
So far we have used a number of Python's built-in types such as int, float, list, and file.

Now it is time to begin exploring some of the more serious power of Python: the ability to create your own types.

This is a big step, so let's start by reviewing some of the historical development of computer language features.

A Brief History of Snail Racing Technology

An entrepreneur name Malek Ology would like to develop a service to run snail races to help non-profit organizations raise funds. Here is the proposed design for Malek's snail-racing track:



At the start of the race, the snails, with their names written on their backs in organic, biodegradable ink, are placed inside the starting line, and Malek starts a timer. As each snail crosses the finish line, Malek records their times.

Malek wants to write a Python program to print the race results. We'll look at the evolution of such a program through the history of programming. Let's start around 1960.

Scalar Variables

Back around 1960, the hot language was FORTRAN. A lot of the work in this language was done using *scalar variables*, that is, a set of variable names, each of which held one number.

Suppose we've just had a snail race, and Judy finished in 87.3 minutes, while Kyle finished in 96.6 minutes. We can create Python variables with those values like this:

```
>>> judy = 87.3
>>> kyle = 96.6
```

To find the winner, we can use some if statements like this:

```
>>> if judy < kyle:
...     print "Judy wins with a time of", judy
... elif judy > kyle:
...     print "Kyle wins with a time of", kyle
... else:
...     print "Judy and Kyle are tied with a time of", judy
...
Judy wins with a time of 87.3
>>>
```

If Judy and Kyle are the only two snails, this program will work fine. Malek puts this all into a script. After each race, he changes the first two lines that give the finish times, and then runs the script.

This will work, but there are a number of objections:

- The person who prepares the race results has to know Python so they can edit the script.
- It doesn't really save any time. Any second-grader can look at the times and figure out who won.
- The names of the snails are part of the program, so if different snails are used, we have to write a new program.
- What if there are *three* snails? There are a lot more cases: three cases where a snail clearly wins; three more possible two-way ties; and a three-way tie. What if Malek wants to race ten snails at once? Too complicated!

Snail-oriented Data Structures: Lists

Let's consider the general problem of a race involving any number of snails. Malek is considering diversifying into amoeba racing, so there might be thousands of competitors in a race. So let's not limit the number of competitors in the program.

Also, to make it possible to use cheaper labor for production runs, let's write a general-purpose script that will read a file with the results for each race, so a relatively less skilled person can prepare that file, and then run a script that will review the results.

We'll use a very simple text file format to encode the race results. Here's an example file for that first race between Judy and Kyle:

```
87.3 Judy
96.6 Kyle
```

And here is a script that will process that file and report on the winning time. The script is called snailist.py. First, reads a race results file named results and stores the times into a list. The .split()method is used to break each line into parts, with the first part containing the elapsed time.

```
#!/usr/local/bin/python
#=========================================================================
# snailist.py:  First snail racing results script.
#------------------------------------------------------------

#--
# Create an empty list to hold the finish times.
#--
timeList = []

#--
# Open the file containing the results.
#--
resultsFile = open ( 'results' )

#--
# Go through the lines of that file, storing each finish time.
#--
for  resultsLine in resultsFile:
    #--
    # Create a list of the fields in the line, e.g., ['87.3', 'Judy\n'].
    #--
    fieldList  =  resultsLine.split()

    #--
    # Convert the finish time into a float and append it to timeList.
    #--
```

```
   timeList.append ( float ( fieldList[0] ) )
```

At this point, timeList is a list of float values. We use the .sort() method to sort the list into ascending order, so that the winning time will be in the first element.

```
#--
# Sort timeList into ascending order, then set ‘winningTime’ to
# the best time.
#--
timeList.sort()
print “The winning time is”, timeList[0]
```

Try building the results file and the script yourself to verify that they work. Try some cases where there are ties.
This script is fine as far as it goes. However, there is one major drawback: it doesn’t tell you who won!

Snail-oriented Data Structures: A List of Tuples

To improve on the script above, let’s modify the script so that it keeps each snail’s time and name together in a two-element tuple such as (87.3, ‘Judy’).

In the improved script, the timeList list is a list of these tuples, and not just a list of times. We can then sort this list, using an interesting property of tuples. If you compare two tuples, and their first elements are not equal, the result is the same as if you compared their first elements. However, if the first elements are equal, Python then compares the second elements of each tuple, and so on until it either finds two unequal values, or finds that all the elements are equal.

Here’s an example. Recall that the function cmp(*a*, *b*), function compares two arbitrary values and returns a negative number if *a* comes before *b*, or a positive number if *a* comes after *b*, or zero if they are considered equal:

```
>>> cmp(50,30)
1
>>> cmp(30,50)
-1
>>> cmp(50,50)
0
>>>
```

If you compare two tuples and the first elements are unequal, the result is the same as if you compared the first two elements. For example:

```
>>> cmp ( (50,30,30), (80,10,10) )
```

```
-1
>>>
```

If, however, the first elements are equal, Python then compares the second elements, or the third elements, until it either finds two unequal elements, or finds that all the elements are equal:

```
>>> cmp ( (50,30,30), (80,10,10) )
-1
>>> cmp ( (50,30,30), (50,10,10) )
1
>>> cmp ( (50,30,30), (50,30,80) )
-1
>>> cmp ( (50,30,30), (50,30,30) )
0
>>>
```

So, watch what happens when we sort a list of two-tuples containing snail times and names:

```
>>> timeList = [ (87.3, 'Judy'), (96.6, 'Kyle'), (63.0, 'Lois') ]
>>> timeList.sort()
>>> timeList
[(63.0, 'Lois'), (87.299999999999997, 'Judy'), (96.599999999999994,
'Kyle')]
>>>
```

Now we have a list that is ordered the way the snails finished. Here is our modified script:

```
#!/usr/local/bin/python
#==================================================
============
# snailtuples.py:  Second snail racing results script.
#--------------------------------------------------------------

#--
# Create an empty list to hold the result tuples.
#--
timeList = []

#--
# Open the file containing the results.
#--
resultsFile = open ( 'results' )
```

```
#--
# Go through the lines of that file, storing each finish time.
# Note that ‘resultsLine’ is set to each line of the file in
# turn, including the terminating newline (‘\n’).
#--
for  resultsLine in resultsFile:
   #--
   # Create a list of the fields in the line, e.g., [‘87.3’, ‘Judy\n’].
   # We use the second argument to .split() to limit the number
   # of fields to two maximum; the first argument (None) means
   # split the line wherever there is any whitespace.
   #--
   fieldList  =  resultsLine.split(None, 1)

   #--
   # Now create a tuple (time,name) and append it to fieldList.
   # Use .rstrip to remove the newline from the second field.
   #--
   snailTuple = (float(fieldList[0]), fieldList[1].rstrip())
   timeList.append ( snailTuple )

#--
# Sort timeList into ascending order.
#--
timeList.sort()

#--
# Print the results.
#--
print “Finish  Time  Name”
print “------ ------ ----”
for  position in range(len(timeList)):
   snailTuple  =  timeList[position]
   print “{0:4d}   {1:6.1f} {2}”.format(position+1, snailTuple[0],
                                snailTuple[1])
```

Here is a sample run with our original two-snail results file:

```
Finish  Time  Name
------ ------ ----
```

```
  1     87.3 Judy
  2     96.6 Kyle
```

Let’s try a larger results file with some names that have spaces in them, just to exercise the script. Here’s the input file:

```
93.3 Queen Elizabeth I
138.4 Erasmus
88.2 Jim Ryun
```

And the output for this run:

```
Finish  Time  Name
------ ------ ----
  1     88.2 Jim Ryun
  2     93.3 Queen Elizabeth I
  3    138.4 Erasmus
```

Abstract Data Types

The preceding section shows how you can use a Python tuple to combine two simple values into a compound value. In this case, we use a 2-element tuple whose first element is the snail’s time and the second element is its name.

We might say that this tuple is an *abstract data type*, that is, a way of combining Python’s basic types (such as floats and strings) into new combinations.

The next step is to combine values *and functions* into an abstract data type. Historically, this is how object-oriented programming arose. The “objects” are packages containing simpler values inside them. However, in general, these packages can also contain *functions*.

Before we start looking at how we build abstract data types in Python, let’s define some import terms and look at some real-world examples.

class

When we try to represent in our program some items out in the real world, we first look to see which items are similar, and group them into *classes*. A class is defined by one or more things that share the same qualities.

For example, we could define the class of *fountains* by saying that they are all permanent man-made structures, that they hold water, that they are outdoors in a public place, and that they keep the water in a decorative way.

It should be easy to determine whether any item is a member of the class or not, by applying these defining rules. For example, Trevi Fountain in Rome fits all the rules: it is man-made, holds water, is outdoors, and is decorative. Lake Geneva has water spraying out of it, but it's not man-made, so it's not a fountain.

instance

One of the members of a class. For example, the class of *airplanes* includes the Wright Biplane of 1903, and the Spirit of St. Louis that Charles Lindbergh flew across the Atlantic.

An instance is always a single item. "Boeing 747" is not an instance, it is a class. However, a specific Boeing 747, with a unique tail number like N1701, is an instance.

attribute

Since the purpose of most computer applications is in record-keeping, within a program, we must often track specific qualities of an instance, which we call *attributes*.

For example, attributes of an airplane include its wingspan, its manufacturer, and its current location, direction, and airspeed.

We can classify attributes into *static* and *dynamic* attributes, depending on whether they change or not. For example, the wingspan and model number of an airplane do not change, but its location and velocity can.

operations

Each class has characteristic operations that can be performed on instances of the class. For example, operations on airplanes include: manufacture; paint; take off; change course; land.

Here is a chart showing some classes, instances, attributes, and operations.

Class	**Instance**	**Attribute**	**Operation**
Airplane	Wright Flyer	Wingspan	Take off
Mountain	Socorro Peak	Altitude	Erupt
Clock	Skeen Library Clock	Amount slow per day	Decorate

Important

You have now seen definitions for most of the important terms in object-oriented programming. Python classes and instances are very similar to these real-world classes and instances. Python instances have attributes too.

For historical reasons, the term *method* is the object-oriented programming equivalent of “operation.”

The term *constructor method* is the Python name for the operation that creates a new instance.

So what is an *object?* This term is used in two different ways:

- An *object* is just an instance.
- *Object-oriented programming* means programming with classes.

Abstract Data Types in Python

We saw how you can use a two-element tuple to group a snail’s time and name together. However, in the real world, we might need to track more than two attributes of an instance.

Suppose Malek wants to keep track of more attributes of a snail, such as its age in days, its weight in grams, its length in millimeters, and its color. We could use a six-element tuple like this:

(87.3, ‘Judy’, 34, 1.66, 39, ‘tan’)

The problem with this approach is that we have to remember that for a tuple T, the time is in T[0], the name in T[1], the age in T[2], and so on.

A cleaner, more natural way to keep track of attributes is to give them names. We might encode those six attributes in a Python dictionary like this:

```
T = { ‘time’:87.3, ‘name’:’Judy’, ‘age’:34, ‘mass’:1.66,
    ‘length’:39, ‘color’:’tan’}
```

With this approach, we can retrieve the name as T[‘name] or the weight as T[‘mass’]. However, now we have lost the ability to put several of these dictionaries into a list and sort the list—how is Python supposed to know which dictionary comes first? What we need is something like a dictionary, but with more features. What we need is Python’s object-oriented features.

Now we’re to look at actual Python classes and instances in action.

class SnailRun: A Very Small Example Class

Let’s start building a snail-racing application for Malek the object-oriented Python way. Let’s assume that all we’re tracking about a particular snail is its name and its finishing time. We need to define a class named SnailRun, whose instances track just these two attributes.

Here is the general form of a class declaration in Python:

```
class ClassName:
   def method_1(self, ...):
      block_1
   def method_2(self, ...):
      block_2
   ... etc.
```

A class declaration starts out with the keyword class, followed by the name of the class you are defining, then a colon (:). The methods of the class follow; each method starts with "def", just as you use to define a function.

Before we look at the construction of the class, let's see how it works in practice. To create an instance in Python, you use the name of the class as if it were a function call, followed by a list of arguments in parentheses. Our SnailRun constructor method will need two arguments: the snail›s name and its finish time. Once we have defined the class, we can build a new instance like this:

```
judyRace9 = SnailRun ( 'judy', 87.3 )
```

To get the snail's name and time attributes from an instance, we use the instance name, followed by a dot (.), followed by the attribute name:

```
>>> judyRace9.name
'judy'
>>> print judyRace9.time
87.3
```

Our example class, SnailRun, will have just two methods:

- All classes have a *constructor* method named "__init__". This method is used to create a new instance.
- We'll write a .show() method to format the contents of the instance for display.

Continuing our example from above, here's an example of the use of the .show() method:

```
>>> print judyRace9.show()
Snail 'judy' finished in 87.3 minutes.
```

Here is the entire class definition:

```
class SnailRun:
   def __init__ ( self, snailName, finishTime ):
        self.name = snailName
        self.time = finishTime

    def show ( self ):
        return ( "Snail '{0}' finished in {1:.1d} minutes.".format(
                self.name, self.time )
```

Instantiation means the construction of a new instance. Here is how instantiation works.

- Somewhere in a Python program, the programmer starts the construction of a new instance by using the class's name followed by parentheses and a list of arguments. Let's call the arguments (a_1, a_2, ...).
- Python creates a new namespace that will hold the instance's attributes. Inside the constructor, this namespace is referred to as self.

Important

The instance *is* basically a namespace, that is, a container for attribute names and their definitions. For other examples of Python namespaces, see Section 2.2, "The assignment statement", Section 5.3, "A namespace is like a dictionary", and Section 9.3, "A module is a namespace".

- The __init__() (constructor) method of the class is executed with the argument list (self, a_1, a_2, ...).

Note that if the constructor takes *N* arguments, the caller passes only the last *N-1* arguments to it.

- When the constructor method finishes, the instance is returned to the caller. From then one, the caller can refer to some attribute *A* of the instance *I* as "*A.I*".

Let's look again in more detail at the constructor:

```
    def __init__ ( self, snailName, finishTime ):
        self.name = snailName
        self.time = finishTime
```

All the constructor does is to take the snail's name and finish time and store these values in the instance's namespace under the names .name and .time, respectively.

Note that the constructor method does not (and cannot) include a return statement. The value of self is implicitly returned to the statement that called the constructor.

As for the other methods of a class, their definitions also start with the special argument self that contains the instance namespace. For any method that takes *N* arguments, the caller passes only the last *N-1*arguments to it.

In our example class, the def for the .show() method has one argument named self, but the caller invokes it with no arguments at all:

```
>>> kyleRace3=SnailRun('Kyle', 96.6)
>>> kyleRace3.show()
"Snail 'Kyle' finished in 96.6 minutes."
```

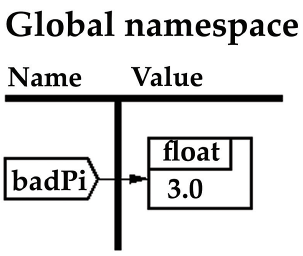
Life Cycle of An Instance

To really understand what is going on inside a running Python program, let's follow the creation of an instance of the SnailRun class from the preceding section.

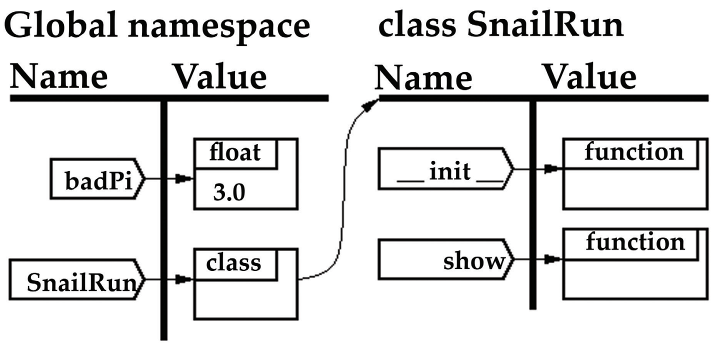
Just for review, let's assume you are using conversational mode, and you create a variable like this:

```
>>> badPi = 3.00
```

Whenever you start up Python, it creates the "global namespace" to hold the names and values you define. After the statement above, here's how it looks.



Next, suppose you type in the class definition as above. As it happens, a class is a namespace too—it is a container for methods. So the global namespace now has two names in it: the variable badPi and the class SnailRun. Here is a picture of the world after you define the class:



Next, create an instance of class SnailRun like this:

>>> j1 = SnailRun ('Judy', 87.3)

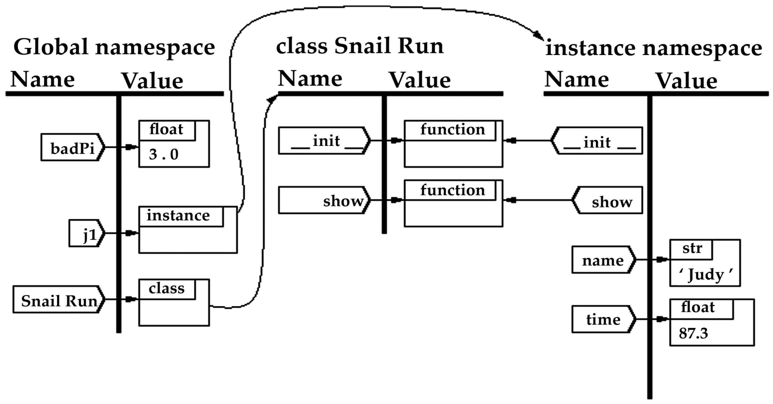
Here is the sequence of operations:

1. Python creates a new instance namespace. This namespace is initially a *copy of the class's namespace*: it contains the two methods .__init__() and .show().
2. The constructor method starts execution with these arguments:
   - The name self is bound to the instance namespace.
   - The name snailName is bound to the string value ‹Judy›.
   - The name finishTime is bound to the float value 87.3.
3. This statement in the constructor
4.   self.name = snailName

creates a new attribute .name in the instance namespace, and assigns it the value 'Judy'.

5. The next statement in the constructor creates an attribute named .time in the instance namespace, and binds it to the value 87.3.
6. The constructor completes, and back in conversational mode, in the global namespace, variable j1 is bound to the instance namespace.

Here's a picture of the world after all this:



Special Methods: Sorting Snail Race Data

Certain method names have special meaning to Python; each of these *special method names* starts with two underbars, "__".

A class's constructor method, __init__(), is an example of a special method. Whenever you use the class's name as if it were a function, in an expression like "SnailRun('Judy', 67.3)", Python executes the constructor method to build the new instance.

There is a full list of all the Python special method names in the *Python quick reference*. Next we will look at another special method, __cmp__, that Python calls whenever you compare two instances of that class.

Going back to our snail-racing application, an instance of the SnailRun class contains everything we need to know about one snail›s performance: its name in the .name attribute and its finish time in the .timeattribute.

However, using the tuple representation back in Section 11.4, "Snail-oriented data structures: A list of tuples", we were able to put a collection of these tuples into a list, and sort the list so that they were ordered by finish time, with the winner first. Let's see what we need to add to class SnailRun so that we can sort a list of them into finish order by calling the .sort() method on the list.

First, a bit of review. Back in Section 6.1, "Conditions and the bool type", we learned about the built-in Python function cmp(*x*, *y*), which returns:

- a negative number if *x* is less than *y*;
- a positive number if *x* is greater than *y*; or
- zero if *x* equals *y*.

In a Python class, if you define a method named "__cmp__", that method is called whenever Python compares two instances of the class. It must return a result using the same conventions as the built-in cmp()function: negative for "<", zero for "==", positive for ">".

In the case of "class SnailRun", we want the snail with the better finishing time to be considered less than the slower snail. So here is one way to define the __cmp__ method for our class:

```
def __cmp__ ( self, other ):
  """Define how to compare two SnailRun instances.
  """
  if self.time < other.time:
    return -1
  elif self.time > other.time:
    return 1
  else:
    return 0
```

When this method is called, self is an instance of class SnailRun, and other should also be an instance of SnailRun.

However, this logic exactly duplicates what the built-in cmp() function does to compare two float values, so we can simplify it like this:

```
def __cmp__ ( self, other ):
  """Define how to compare two SnailRun instances.
  """
  return cmp(self.time, other.time)
```

Let's look at another special method, __str__(). This one defines how Python converts an instance of a class into a string. It is called, for example, when you name an instance in a print statement, or when you pass an instance to Python's built-in str() function.

The __str__() method of a class returns a string value. It is up to the writer of the class what string value gets returned. As usual for Python methods, the self argument contains the instance. In the case of class SnailRun, we'll want to display the snail's name (.name attribute) and finishing time (.time attribute). Here's one possible version:

```
def __str__ ( self ):
  """Return a string representation of self.
  """
  return "{0:8.1f} {1}".format(self.time, self.name)
```

This method will format the finishing time into an 8-character string, with one digit after the decimal point, followed by one space, then the snail's name.
Let's assume that the __cmp__() and __str__() methods have been added to our snails.py module, and show their use in some conversational examples.

```
>>> from snails import *
>>> sally4 = SnailRun('Sally', 88.8)
>>> jim4=SnailRun('Jim', 76.5)
>>>
```

Now that we have two SnailRun instances, we can show how the __str__() method formats them for printing:

```
>>> print sally4
    88.8 Sally
>>> print jim4
    76.5 Jim
>>>
```

We can also show the various ways that Python compares two instances using our new __cmp__() method.

```
>>> cmp(sally4,jim4)
1
>>> sally4 > jim4
True
>>> sally4 <= jim4
False
>>> sally4 < jim4
False
>>>
```

Now that we have defined how instances are to be ordered, we can sort a list of them in order by finish time. First we throw them into the list in any old order:

```
>>> judy4 = SnailRun ( 'Judy', 67.3 )
>>> blake4 = SnailRun ( 'Blake', 181.4 )
>>> race4 = [sally4, jim4, judy4, blake4]
>>> for run in race4:
...     print run
...
    88.8 Sally
    76.5 Jim
    67.3 Judy
```

```
  181.4 Blake
>>>
```

The .sort() method on a list uses Python's cmp() function to compare items when sorting them, and this in turn will call our class's __cmp__() method to sort them by finishing time.

```
>>> race4.sort()
>>> for run in race4:
...     print run
...
   67.3 Judy
   76.5 Jim
   88.8 Sally
  181.4 Blake
>>>
```

For an extended example of a class that implements a number of special methods, see *rational.py: An example Python class*. This example shows how to define a new kind of numbers, and specify how operators such as "+" and "*" operate on instances.

CHAPTER 9

PATTERN FOR PYTHON

Tom De Smedt and Walter Daelemans

CLiPS Computational Linguistics Group University of Antwerp 2000 Antwerp, Belgium

ABSTRACT

Pattern is a package for Python 2.4+ with functionality for web mining (Google + Twitter + Wikipedia, web spider, HTML DOM parser), natural language processing (tagger/chunker, n-gram search, sentiment analysis, WordNet), machine learning (vector space model, k-means clustering, Naive Bayes + k-NN + SVM classifiers) and network analysis (graph centrality and visualization). It is well documented and bundled with 30+ examples and 350+ unit tests. The source code is licensed under BSD and available from http://www.clips.ua.ac.be/pages/pattern.

Keywords: Python, data mining, natural language processing, machine learning, graph networks

Citation: Tom De Smedt, Walter Daelemans, "Pattern for Python", Journal for Machine Learning Research, Vol. 13 (Jun), pp. 2063−2067, 2012.

INTRODUCTION

The World Wide Web is an immense collection of linguistic information that has in the last decade gathered attention as a valuable resource for tasks such as machine translation, opinion mining and trend detection, that is, "Web as Corpus" (Kilgarriff and Grefenstette, 2003). This use of the WWW poses a challenge since the Web is interspersed with code (HTML markup) and lacks metadata (language identification, part-of-speech tags, semantic labels).

"Pattern" (BSD license) is a Python package for web mining, natural language processing, machine learning and network analysis, with a focus on ease-of-use. It offers a mash-up of tools often used when harnessing the Web as a corpus, which usually requires several independent toolkits chained together in a practical application. Several such toolkits with a user interface exist in the scientific community, for example ORANGE (Demsar et al., 2004) for machine learning and ˇ GEPHI (Bastian et al., 2009) for graph visualization. By contrast, PATTERN is more related to toolkits such as NLTK (Bird et al., 2009), PYBRAIN (Schaul et al., 2010) and NETWORKX (Hagberg et al., 2008), in that it is geared towards integration in the user's own programs. Also, it does not specialize in one domain but provides general cross-domain functionality.

The package aims to be useful to both a scientific and a non-scientific audience. The syntax is straightforward. Function names and parameters were so chosen as to make the commands selfexplanatory. The documentation assumes no prior knowledge. We believe that PATTERN is valuable as a learning environment for students, as a rapid development framework for web developers, and in research projects with a short development cycle.

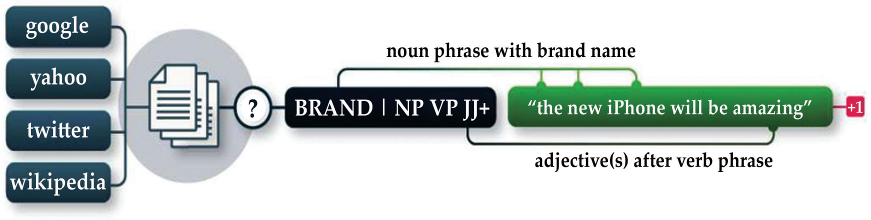

google
yahoo
twitter
wikipedia
?
noun phrase with brand name
BRAND | NP VP JJ+
"the new iPhone will be amazing"
+1
adjective(s) after verb phrase

Figure 1: Example workflow. Text is mined from the web and searched by syntax and semantics. Sentiment analysis (positive/negative) is performed on matching phrases.

PACKAGE OVERVIEW

PATTERN is organized in separate modules that can be chained together, as shown in Figure 1. For example, text from Wikipedia (pattern.web) can be parsed for part-of-speech tags (pattern.en), queried by syntax and semantics (pattern.search), and used to train a classifier (pattern.vector).

pattern.web Tools for web data mining, using a download mechanism that supports caching, proxies, asynchronous requests and redirection. A SearchEngine class provides a uniform API to multiple web services: Google, Bing, Yahoo!, Twitter, Wikipedia, Flickr and news feeds using FEED PARSER (packages.python.org/feedparser). The module includes an HTML parser based on BEAUTIFUL SOUP (crummy.com/software/beautifulsoup), a PDF parser based on PDFMINER (unixuser.org/ euske/python/pdfminer), a web crawler, and a webmail interface.

pattern.en Fast, regular expressions-based shallow parser for English (identifies sentence constituents, e.g., nouns, verbs), using a finite state part-of-speech tagger (Brill, 1992) extended with a tokenizer, lemmatizer and chunker. Accuracy for Brill's tagger is 95% and up. A parser with higher accuracy (MBSP) can be plugged in. The module has a Sentence class for parse tree traversal, functions for singularization/pluralization (Conway, 1998), conjugation, modality and sentiment analysis. It comes bundled with WORDNET3 (Fellbaum, 1998) and PYWORDNET.

pattern.nl Lightweight implementation of pattern.en for Dutch, using the BRILL-NL language model (Geertzen, 2010). Contributors are encouraged to read the developer documentation on how to add support for other languages.

pattern.search N-gram pattern matching algorithm for Sentence objects. The algorithm uses an approach similar to regular expressions. Search queries can include a mixture of words, phrases, part-of-speech-tags, taxonomy terms (e.g., pet = dog, cat or goldfish) and control characters (e.g., + = multiple, * = any, () = optional) to extract relevant information.

pattern.vector Vector space model using a Document and a Corpus class. Documents are lemmatized bag-of-words that can be grouped in a sparse corpus to compute TF-IDF, distance metrics (cosine, Euclidean, Manhattan, Hamming) and dimension reduction (Latent Semantic Analysis). The module includes a hierarchical and a k-means clustering algorithm, optimized with the kmeans++ initialization algorithm (Arthur and Vassilvitskii, 2007) and triangle inequality (Elkan, 2003). A Naive Bayes, a k-NN, and a SVM classifier using LIBSVM (Chang and Li, 2011) are included, with tools for feature selection (information gain) and K-fold cross validation.

pattern.graph Graph data structure using Node, Edge and Graph classes, useful (for example) for modeling semantic networks. The module has algorithms for shortest path finding, subgraph partitioning, eigenvector centrality and betweenness centrality (Brandes, 2001). Centrality algorithms were ported from NETWORKX. The module has a force-based layout algorithm that positions nodes in 2D space. Visualizations can be exported to HTML and manipulated in a browser (using our canvas.js helper module for the HTML5 Canvas2D element).

pattern.metrics Descriptive statistics functions. Evaluation metrics including a code profiler, functions for accuracy, precision and recall, confusion matrix, inter-rater agreement (Fleiss' kappa), string similarity (Levenshtein, Dice) and readability (Flesch).

pattern.db Wrappers for CSV files and SQLITE and MYSQL databases.

EXAMPLE SCRIPT

As an example, we chain together four PATTERN modules to train a k-NN classifier on adjectives mined from Twitter. First, we mine 1,500 tweets with the hashtag #win or #fail (our classes), for example: "$20 tip off a sweet little old lady today #win". We parse the part-of-speech tags for each tweet, keeping adjectives. We group the adjective vectors in a corpus and use it to train the classifier. It predicts "sweet" as WIN and "stupid" as FAIL. The results may vary depending on what is currently buzzing on Twitter.

The source code is shown in Figure 2. Its size is representative for many real-world scenarios, although a real-world classifier may need more training data and more rigorous feature selection.

```
from pattern.web    import Twitter
from pattern.en     import Sentence, parse
from pattern.search import search
from pattern.vector import Document, Corpus, KNN

corpus = Corpus()
for i in range(1,15):
    for tweet in Twitter().search('#win OR #fail', start=i, count=100):
        p = '#win' in tweet.description.lower() and 'WIN' or 'FAIL'
        s = tweet.description.lower()
        s = Sentence(parse(s))
        s = search('JJ', s) # JJ = adjective
        s = [match[0].string for match in s]
        s = ' '.join(s)
        if len(s) > 0:
```

```
        corpus.append(Document(s, type=p))

classifier = KNN()
for document in corpus:
    classifier.train(document)
print classifier.classify('sweet') # yields 'WIN'
print classifier.classify('stupid') # yields 'FAIL'd
```

Figure 2: Example source code for a k-NN classifier trained on Twitter messages.

CASE STUDY

As a case study, we used PATTERN to create a Dutch sentiment lexicon (De Smedt and Daelemans, 2012). We mined online Dutch book reviews and extracted the 1,000 most frequent adjectives. These were manually annotated with positivity, negativity, and subjectivity scores. We then enlarged the lexicon using distributional expansion. From the TWNC corpus (Ordelman et al., 2007) we extracted the most frequent nouns and the adjectives preceding those nouns. This results in a vector space with approximately 5,750 adjective vectors with nouns as features. For each annotated adjective we then computed k-NN and inherited its scores to neighbor adjectives. The lexicon is bundled into PATTERN 2.3.

DOCUMENTATION

PATTERN comes bundled with examples and unit tests. The documentation contains a quick overview, installation instructions, and for each module a detailed page with the API reference, examples of use and a discussion of the scientific principles. The documentation assumes no prior knowledge, except for a background in Python programming. The unit test suite includes a set of corpora for testing accuracy, for example POLARITY DATA SET V2.0 (Pang and Lee, 2004).

SOURCE CODE

PATTERN is written in pure Python, meaning that we sacrifice performance for development speed and readability (i.e., slow clustering algorithms). The package runs on all platforms and has no dependencies, with the exception of NumPy when LSA is used. The source code is annotated with developer comments. It is hosted online on GitHub (github.com) using the Git revision control system. Contributions are welcomed.

The source code is released under a BSD license, so it can be incorporated into proprietary products or used in combination with other open source packages such as SCRAPY (web mining), NLTK (natural language processing), PYBRAIN and PYML (machine learning) and NETWORKX (network analysis). We provide an interface to MBSP FOR PYTHON (De Smedt et al., 2010), a robust, memory-based shallow parser built on the TIMBL machine learning software. The API's for the PATTERN parser and MBSP are identical.

ACKNOWLEDGMENTS

Development was funded by the Industrial Research Fund (IOF) of the University of Antwerp.

REFERENCES

1. David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. Proceedingsof the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 1027–1035,2007.
2. Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software forexploring and manipulating networks. Proceedings of the Third International ICWSM Conference,2009.
3. Steven Bird, Ewan Klein, and Edward Loper. Natural Language Processing with Python. O'ReillyMedia, 2009.
4. Ulrik Brandes. A faster algorithm for betweenness centrality. The Journal of Mathematical Sociology,25(2):163–177, 2001.
5. Eric Brill. A simple rule-based part of speech tagger. Proceedings of the Third Conference on Applied Natural Language Processing, pages 152–155, 1992.
6. Chih-Chung Chang and Chih-Jen Li. LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2(3), 2011.
7. Damian Conway. An algorithmic approach to english pluralization. Proceedings of the Second Annual Perl Conference, 1998.
8. Tom De Smedt and Walter Daelemans. Vreselijk mooi! (terribly beautiful): A subjectivity lexicon for dutch adjectives. Proceedings of the 8th Language Resources and Evaluation Conference (LREC'12), pages 3568—-3572, 2012.
9. Tom De Smedt, Vincent Van Asch, and Walter Daelemans. Memory-based shallow parser for python. CLiPS Technical Report Series, 2, 2010.
10. Janez Demsar, Bla ˇ z Zupan, Gregor Leban, and Tomaz Curk. Orange: From experimental ˇ machine learning to interactive data mining. Knowledge Discovery in Databases, 3202:537–539, 2004.
11. Charles Elkan. Using the triangle inequality to accelerate k-means. Proceedings of the Twentieth International Conference on Machine Learning, pages 147–153, 2003.
12. Christiane Fellbaum. WordNet: An Electronic Lexical Database. MIT Press, Cambridge, 1998.
13. Jeroen Geertzen. Jeroen geertzen :: software & demos : Brill-nl, June

2010. URL http: //cosmion.net/jeroen/software/brill_pos/.

14. Aric Hagberg, Daniel Schult, and Pieter Swart. Exploring network structure, dynamics, and function using networkx. Proceedings of the 7th Python in Science Conference, pages 11–15, 2008.
15. Adam Kilgarriff and Gregory Grefenstette. Introduction to the special issue on the web as corpus. Computational Linguistics, 29(3):333–347, 2003.
16. Roeland Ordelman, Franciska de Jong, Arjan van Hessen, and Hendri Hondorp. TwNC: A multifaceted dutch news corpus. ELRA Newsletter, 12:3–4, 2007.
17. Bo Pang and Lillian Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. Proceedings of the ACL, pages 271–278, 2004.
18. Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Ruckstieß, ¨ and Jurgen Schmidhuber. Pybrain. ¨ Journal of Machine Learning Research, pages 743–746, 2010.

CHAPTER
10

PYSTRUCT - LEARNING STRUCTURED PREDICTION IN PYTHON

Andreas C. M¨uller and Sven Behnke

Institute of Computer Science, Department VI University of Bonn Bonn, Germany

ABSTRACT

Structured prediction methods have become a central tool for many machine learning applications. While more and more algorithms are developed, only very few implementations are available.

PyStruct aims at providing a general purpose implementation of standard structured prediction methods, both for practitioners and as a baseline for researchers. It is written in Python and adapts paradigms and types from the scientific Python community for seamless integration with other projects.

Keywords: structured prediction, structural support vector machines, conditional random fields, Python

Citation: A. C. Müller, S. Behnke, "PyStruct - Learning Structured Prediction in Python", Journal for Machine Learning Research, Vol. 15 (Jun), pp. 2055–2060, 2014.

INTRODUCTION

In recent years there has been a wealth of research in methods for learning structured prediction, as well as in their application in areas such as natural language processing and computer vision. Unfortunately only few implementations are publicly available—many applications are based on the non-free implementation of Joachims et al. (2009).

PyStruct aims at providing a high-quality implementation with an easy-to-use interface, in the high-level Python language. This allows practitioners to efficiently test a range of models, as well as allowing researchers to compare to baseline methods much more easily than this is possible with current implementations. PyStruct is BSD-licensed, allowing modification and redistribution of the code, as well as use in commercial applications. By embracing paradigms established in the scientific Python community and reusing the interface of the widely-used scikit-learn library (Pedregosa et al., 2011), PyStruct can be used in existing projects, replacing standard classifiers. The online documentation and examples help new users understand the somewhat abstract ideas behind structured prediction.

STRUCTURED PREDICTION AND CASTING IT INTO SOFTWARE

Structured prediction can be defined as making a prediction f(x) by maximizing a compatibility function between an input x and the possible labels y (Nowozin and Lampert, 2011). Most current approaches use linear functions, leading to:

$$f(x) = \operatorname*{arg\,max}_{y \in \mathcal{Y}} \; \theta^T \Psi(x, y). \tag{1}$$

Here, y is a structured label, Ψ is a joint feature function of x and y, and θ are parameters of the model. Structured means that y is more complicated than a single output class, for example a label for each word in a sentence or a label for each pixel in an image. Learning structured prediction means learning the parameters θ from training data.

Using the above formulation, learning can be broken down into three sub-problems:

1. Optimizing the objective with respect to θ.
2. Encoding the structure of the problem in a joint feature function Ψ.

3. Solving the maximization problem in Equation 1.

The later two problems are usually tightly coupled, as the maximization in Equation 1 is usually only feasible by exploiting the structure of Ψ, while the first is treated as independent. In fact, when 3. can not be done exactly, learning θ strongly depends on the quality of the approximation. However, treating approximate inference and learning as a joint optimization problem is currently out of the scope of the package, and we implement a more modular setup. PyStruct takes an object-oriented approach to decouple the task-dependent implementation of 2. and 3. from the general algorithms used to solve 1.

Estimating θ is done in learner classes, which currently support cutting plane algorithms for structural support vector machines (SSVMs Joachims et al. (2009)), subgradient methods for SSVMs Ratliff et al. (2007), Block-coordinate Frank-Wolfe (BCFW) (LacosteJulien et al., 2012), the structured perceptron and latent variable SSVMs (Yu and Joachims, 2009). The cutting plane implementation uses the cvxopt package (Andersen et al., 2012) for quadratic optimization. Encoding the structure of the problem is done using model classes, which compute Ψ and encode the structure of the problem. The structure of Ψ determines the hardness of the maximization in Equation (1) and is a crucial factor in learning. PyStruct implements models (corresponding to particular forms of Ψ) for many common cases, such as multi-class and multi-label classification, conditional random fields with constant or data-dependent pairwise potentials, and several latent variable models. The maximization for finding y in Equation 1 is carried out using external libraries, such as OpenGM (Kappes et al., 2013), LibDAI (Mooij, 2010) and others. This allows the user to choose from a wide range of optimization algorithms, including (loopy) belief propagation, graph-cuts, QPBO, dual subgradient, MPBP, TRWs, LP and many other algorithms. For problems where exact inference is infeasible, PyStruct allows the use of linear programming relaxations, and provides modified loss and feature functions to work with the continuous labels. This approach, which was outlined in Finley and Joachims (2008) allows for principled learning when exact inference is intractable. When using approximate integral solvers, learning may finish prematurely and results in this case depend on the inference scheme and learning algorithm used.

Table 1 lists algorithms and models that are implemented in PyStruct and compares them to other public structured prediction libraries: Dlib (King, 2009), SVM^{struct} (Joachims et al., 2009) and CRFsuite (Okazaki, 2007). We also give the programming language and the project license.

Table 1: Comparison of structured prediction software packages. CP stands for cutting plane optimization of SSVMs, SG for online subgradient optimization of SSVMs, LV for latent variable SSVMs, ML for maximum likelihood learning, Chain for chain-structured models with pairwise interactions, Graph for arbitrary graphs with pairwise interactions, and LDCRF for latent dynamic CRF (Morency et al., 2007). 1PyStruct itself is BSD licensed, but uses the GPL-licensed package cvxopt for cuttingplane learning

Package	Language	License	Algorithms				Models			
			CP	SG	BCFW	LV	ML	Chain	Graph	LDCRF
PyStruct	Python	BSD[1]	✓[1]	✓	✓	✓	×	✓	✓	✓
SVM^{struct}	C++	non-free	✓	×	×	✓	×	×	×	×
Dlib	C++	boost	✓	×	×	×	×	✓	✓	×
CRFsuite	C++	BSD	×	×	×	×	✓	✓	×	×

USAGE EXAMPLE: SEMANTIC IMAGE SEGMENTATION

Conditional random fields are an important tool for semantic image segmentation. We demonstrate how to learn an n-slack support vector machine (Tsochantaridis et al., 2006) on a superpixel-based CRF on the popular Pascal data set. We use unary potentials generated using TextonBoost from Kr¨ahenb¨uhl and Koltun (2012). The superpixels are generated using SLIC (Achanta et al., 2012).1 Each sample (corresponding on one entry of the list X) is represented as a tuple consisting of input features and a graph representation.

Listing 1: Example of defining and learning a CRF model

```
1 model = crfs.EdgeFeatureGraphCRF(
2     class_weight=inverse_frequency, symmetric_edge_features=[0, 1],
3     antisymmetric_edge_features=[2], inference_method='qpbo')
4
5 ssvm = learners.NSlackSSVM(model, C=0.01, n_jobs=-1)
6 ssvm.fit(X, Y)
```

The source code is shown in Listing 1. Lines 1-3 declare a model using parametric edge potentials for arbitrary graphs. Here class weight re-weights the hamming loss according to inverse class frequencies. The parametric pairwise interactions have three features: a constant feature, color similarity, and relative vertical position. The first two are declared to be symmetric with respect to the direction of an edge, the last is antisymmetric. The inference method used is QPBO-fusion moves. Line 5 creates a learner object that will learn the parameters for the given model using the n-slack cutting plane method, and line 6 performs the actual learning. Using this simple setup, we achieve an accuracy of 30.3 on the validation set following the protocol of Kr¨ahenb¨uhl and Koltun (2012), who report 30.2 using a more complex approach. Training the structured model takes approximately 30 minutes using a single i7 core.

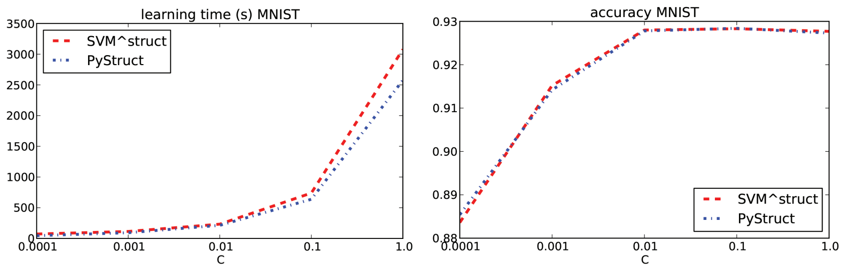


Figure 1: Runtime comparison of PyStruct and SVMstruct for multi-class classification.

EXPERIMENTS

While PyStruct focuses on usability and covers a wide range of applications, it is also important that the implemented learning algorithms run in acceptable time. In this section, we compare our implementation of the 1-slack cutting plane algorithm (Joachims et al., 2009) with the implementation in SVM^{struct}. We compare performance of the CrammerSinger multi-class SVM with respect to learning time and accuracy on the MNIST data set of handwritten digits. While multi-class classification is not very interesting from a structured prediction point of view, this problem is well-suited to benchmark the cutting plane solvers with respect to accuracy and speed.

Results are shown in Figure 1. We report learning times and accuracy for varying regularization parameter C. The MNIST data set has 60 000 training examples, 784 features and 10 classes. The figure indicates that PyStruct has competitive performance, while using a high-level interface in a dynamic programming language.

CONCLUSION

This paper introduced PyStruct, a modular structured learning and prediction library in Python. PyStruct is geared towards ease of use, while providing efficient implementations. PyStruct integrates itself into the scientific Python eco-system, making it easy to use with existing libraries and applications. Currently, PyStruct focuses on max-margin and perceptron-based approaches. In the future, we plan to integrate other paradigms, such as sampling-based learning (Wick et al., 2011), surrogate objectives (for example pseudo-likelihood), and approaches that allow for a better integration of inference and learning (Meshi et al., 2010).

ACKNOWLEDGMENTS

The authors would like to thank Vlad Niculae and Forest Gregg for their contributions to PyStruct and Andr´e Martins for his help in integrating the AD^3 solver with PyStruct. This work was partially funded by the B-IT research school.

REFERENCES

1. Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine S¨usstrunk. SLIC superpixels compared to state-of-the-art superpixel methods. PAMI, 2012.
2. Martin S. Andersen, Joachin Dahl, and Lieven Vandenberghe. CVXOPT: A Python package for convex optimization, version 1.1.5. Available at http://cvxopt.org/, 2012.
3. Thomas Finley and Thorsten Joachims. Training structural SVMs when exact inference is intractable. In ICML, 2008.
4. Thorsten Joachims, Thomas Finley, and Chun-Nam John Yu. Cutting-plane training of structural SVMs. JMLR, 77(1), 2009.
5. J¨org H Kappes, Bjoern Andres, Fred A Hamprecht, Christoph Schn¨orr, Sebastian Nowozin, Dhruv Batra, Sungwoong Kim, Bernhard X Kausler, Jan Lellmann, Nikos Komodakis, et al. A comparative study of modern inference techniques for discrete energy minimization problems. In CVPR, 2013.
6. Davis E. King. Dlib-ml: A machine learning toolkit. JMLR, 10, 2009.
7. Philipp Kr¨ahenb¨uhl and Vladlen Koltun. Efficient inference in fully connected CRFs with Gaussian edge potentials. In NIPS, 2012.
8. Simon Lacoste-Julien, Mark Schmidt, and Francis Bach. A simpler approach to obtaining an O(1/t) convergence rate for projected stochastic subgradient descent. arXiv preprint arXiv:1212.2002, 2012.
9. Ofer Meshi, David Sontag, Tommi Jaakkola, and Amir Globerson. Learning efficiently with approximate inference via dual losses. In ICML, 2010.
10. Joris M. Mooij. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. JMLR, 2010.
11. L-P Morency, Ariadna Quattoni, and Trevor Darrell. Latent-dynamic discriminative models for continuous gesture recognition. In CVPR, 2007.
12. Sebastian Nowozin and Christoph H. Lampert. Structured Learning and Prediction in Computer Vision. Now Publishers Inc., 2011.
13. Naoaki Okazaki. CRFsuite: A fast implementation of conditional random fields (CRFs), 2007. URL http://www.chokkan.org/software/crfsuite/.

14. Fabian Pedregosa, Ga¨el Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. JMLR, 2011.
15. Nathan Ratliff, J. Andrew (Drew) Bagnell, and Martin Zinkevich. (Online) subgradient methods for structured prediction. In AISTATS, 2007.
16. Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, Yasemin Altun, and Yoram Singer. Large margin methods for structured and interdependent output variables. JMLR, 6(2), 2006.
17. Michael Wick, Khashayar Rohanimanesh, Kedar Bellare, Aron Culotta, and Andrew McCallum. Samplerank: Training factor graphs with atomic gradients. In ICML, 2011.
18. Chun-Nam John Yu and Thorsten Joachims. Learning structural SVMs with latent variables. In ICML, 2009.

SECTION IV
MACHINE LEARNING WITH PYTHON

CHAPTER
11

PYTHON ENVIRONMENT FOR BAYESIAN LEARNING: INFERRING THE STRUCTURE OF BAYESIAN NETWORKS FROM KNOWLEDGE AND DATA

Abhik Shah and Peter Woolf

Department of Chemical Engineering 3320 G.G. Brown Ann Arbor, MI 48103, USA

ABSTRACT

In this paper, we introduce PEBL, a Python library and application for learning Bayesian network structure from data and prior knowledge that provides features unmatched by alternative software packages: the ability to use interventional data, flexible specification of structural priors, modeling

Citation: Abhik Shah, Peter Woolf; "Python Environment for Bayesian Learning: Inferring the Structure of Bayesian Networks from Knowledge and Data", vol. 10 (Feb):159--162, 2009.

with hidden variables and exploitation of parallel processing. PEBL is released under the MIT open-source license, can be installed from the Python Package Index and is available at http://pebl-project.googlecode.com.

Keywords: Bayesian networks, python, open source software

INTRODUCTION

Bayesian networks (BN) have become a popular methodology in many fields because they can model nonlinear, multimodal relationships using noisy, inconsistent data. Although learning the structure of BNs from data is now common, there is still a great need for high-quality open-source software that can meet the needs of various users. End users require software that is easy to use; supports learning with different data types; can accommodate missing values and hidden variables; and can take advantage of various computational clusters and grids. Researchers require a framework for developing and testing new algorithms and translating them into usable software. We have developed the Python Environment for Bayesian Learning (PEBL) to meet these needs.

PEBL FEATURES

PEBL provides many features for working with data and BNs; some of the more notable ones are listed below.

Structure Learning

PEBL can load data from tab-delimited text files with continuous, discrete and class variables and can perform maximum entropy discretization. Data collected following an intervention is important for determining causality but requires an altered scoring procedure (Pe'er et al., 2001; Sachs et al., 2002). PEBL uses the BDe metric for scoring networks and handles interventional data using the method described by Yoo et al. (2002).

```
from pebl import data, result
from pebl.learner.greedy import GreedyLearner
from pebl.taskcontroller.xgrid import XgridController

dataset = data.fromfile('mydata.txt')
runner = XgridController('grid.com', 'pass')
learners = [GreedyLearner(dataset) for i in range(10)]
results = runner.run(learners)
result.merge(results).tohtml('./myoutput')
```

(a) Python script

```
[data]
filename = mydata.txt
[learner]
type = greedy.GreedyLearner
numtasks = 10
[taskcontroller]
type=xgrid.XgridController
[xgrid]
controller = grid.com
password = pass
[result]
output = ./myoutput
```

(b) PEBL configuration file

Figure 1: Two ways of using PEBL: with a Python script and a configuration file. Both methods create 10 greedy learners with default parameters and run them on an Apple Xgid. The Python script can be typed in an interactive shell, run as a script or included as part of a larger application.

PEBL can handle missing values and hidden variables using exact marginalization and Gibbs sampling (Heckerman, 1998). The Gibbs sampler can be resumed from a previously suspended state, allowing for interactive inspection of preliminary results or a manual strategy for determining satisfactory convergence.

A key strength of Bayesian analysis is the ability to use prior knowledge. PEBL supports structural priors over edges specified as 'hard' constraints or 'soft' energy matrices (Imoto et al., 2003) and arbitrary constraints specified as Python functions or lambda expressions.

PEBL includes greedy hill-climbing and simulated annealing learners and makes writing custom learners easy. Efficient implementaion of learners requires careful programming to eliminate redundant computation. PEBL provides components to alter, score and rollback changes to BNs in a simple, transactional manner and with these, efficient learners look remarkably similar to pseudocode.

Convenience and Scalability

PEBL includes both a library and a command line application. It aims for a balance between ease of use, extensibility and performance. The majority of PEBL is written in Python, a dynamically-typed programming language that runs on all major operating systems. Critical sections use the numpy library

(Ascher et al., 2001) for high-performance matrix operations and custom extensions written in ANSI C for portability and speed.

PEBL's use of Python makes it suitable for both programmers and domain experts. Python provides interactive shells and notebook interfaces and includes an extensive standard library and many third-party packages. It has a strong presence in the scientific computing community (Oliphant, 2007). Figure 1 shows a script and configuration file example that showcase the ease of using PEBL.

Table 1: Comparing the features of popular Bayesian network structure learning software

	BANJO	**BNT**	**Causal Explorer**	**Deal**	**LibB**	**PEBL**
Latest Version	2.0.1	1.04	1.4	1.2-25	2.1	0.9.10
License	Academic [1]	GPL	Academic [1]	GPL	Academic [1]	MIT
Scripting Language	Matlab [2]	Matlab	Matlab	R	N/A	Python
Application	Yes	No	No	No	Yes	Yes
Interventional Data	No	Yes	No	No	No	Yes
DBN	Yes	Yes	No	No	No	No
Structural Priors	Yes [3]	No	No	No	No	Yes
Missing Data	No	Yes	No	No	Yes	Yes
Parallel Execution	No	No	No	No	No	Yes

[1] Custom academic, non-commercial license; not OSI approved.
[2] Via a Matlab-Java bridge.
[3] Only constraints/hard-priors supported.

While many tasks related to Bayesian learning are embarrassingly parallel in theory, few software packages take advantage of it. PEBL can execute learning tasks in parallel over multiple processors or CPU cores, an Apple Xgrid,1 an IPython cluster2 or the Amazon EC2 platform.3 The EC2 platform is especially attractive for scientists because it allows one to rent processing power on an on-demand basis and execute PEBL tasks on them.

With appropriate configuration settings and the use of parallel execution, PEBL can be used for large learning tasks. Although PEBL has been tested successfully with data sets with 10000 variables and samples, BN structure learning is a known NP-Hard problem (Chickering et al., 1994) and analysis using data sets with more than a few hundred variables is likely to result in poor results due to poor coverage of the search space.

PEBL DEVELOPMENT

The benefits of open source software derive not just from the freedoms afforded by the software license but also from the open and collaborative development model. PEBL's source code repository and issue tracker are hosted at Google Code and freely available to all. Additionally, PEBL includes over 200 automated unit tests and mandates that every source code submission and resolved error be accompanied with tests.

RELATED SOFTWARE

While there are many software tools for working with BNs, most focus on parameter learning and inference rather than structure learning. Of the few tools for structure learning, few are open-source and none provide the set of features included in PEBL. As shown in Table 1, the ability to handle interventional data, model with missing values and hidden variables, use soft and arbitrary priors and exploit parallel platforms are unique to PEBL. PEBL, however, does not currently provide any features for inference or learning Dynamic Bayesian Networks (DBN). Despite its use of optimized matrix libraries and custom C extension modules, PEBL can be an order of magnitude or more slower than software written in Java or C/C++; the ability to use a wider range of data and priors, the parallel processing features and the ease-of-use, however, should make it an attractive option for many users.

CONCLUSION AND FUTURE WORK

We have developed a library and application for learning BNs from data and prior knowledge. The set of features found in PEBL is unmatched by alternative packages and we hope that our open development model will convince others to use PEBL as a platform for BN algorithms research.

ACKNOWLEDGMENTS

We would like to acknowledge support for this project from the NIH grant #U54 DA021519.

REFERENCES

1. D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. An open source project: Numerical python. Technical Report UCRL-MA-128569, Lawrence Livermore National Laboratory, September 2001.
2. D. M. Chickering, D. Geiger, and D. Heckerman. Learning bayesian networks is np-hard. Technical Report MSR-TR-94-17, Microsoft Research, November 1994.
3. D. Heckerman. A tutorial on learning with bayesian networks. In Learning in Graphical Models, pages 301–354. The MIT Press, 1998.
4. S. Imoto, T. Higuchi, T. Goto, K. Tashiro, S. Kuhara, and S. Miyano. Combining microarrays and biological knowledge for estimating gene networks via bayesian networks. Bioinformatics Conference, 2003. CSB 2003. Proceedings of the 2003 IEEE, pages 104–113, 2003.
5. T. E. Oliphant. Python for scientific computing. Computing in Science & Engineering, pages 10–20, 2007.
6. D. Pe'er, A. Regev, G. Elidan, and N. Friedman. Inferring subnetworks from perturbed expression profiles. Bioinformatics, 1(1):1–9, 2001.
7. K. Sachs, D. Gifford, T. Jaakkola, P. Sorger, and D. Lauffenburger. Bayesian network approach to cell signaling pathway modeling. Science's STKE, 2002.
8. C. Yoo, V. Thorsson, and G. F. Cooper. Discovery of causal relationships in a gene-regulation pathway from a mixture of experimental and observational DNA microarray data. Pac Symp Biocomput, 7:498–509, 2002.

CHAPTER

12

SCIKIT-LEARN: MACHINE LEARNING IN PYTHON

Fabian Pedregosa[1], Gael Varoquaux[1], Alexandre Gramfort[1], Vincent Michel[1], Bertrand Thirion[1], Olivier Grisel[2], Mathieu Blondel[3], Peter Prettenhofer[4], Ron Weiss[5], Vincent Dubourg[6], Jake Vanderplas[7], Alexandre Passos[8], David Cournapeau[9], Matthieu Brucher[10], Matthieu Perrot[11], and Edouard Duchesnay[11]

[1]Parietal, INRIA Saclay Neurospin, Bat 145, CEA Saclay ˆ 91191 Gif sur Yvette – France

[2]Nuxeo 20 rue Soleillet 75 020 Paris – France

[3]Kobe University 1-1 Rokkodai, Nada Kobe 657-8501 – Japan

[4]Bauhaus-Universitat Weimar ¨ Bauhausstr. 11 99421 Weimar – Germany

[5]Google Inc 76 Ninth Avenue New York, NY 10011 – USA

[6]Clermont Universite, IFMA, EA 3867, LaMI ´ BP 10448, 63000 Clermont-Ferrand – France

[7]Astronomy Department University of Washington, Box 351580 Seattle, WA 98195 – USA

[8]IESL Lab UMass Amherst Amherst MA 01002 – USA

[9]Enthought 21 J.J. Thompson Avenue Cambridge, CB3 0FA – UK

Citation: F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay; "Scikit-learn: Machine Learning in Python", Journal for Machine Learning Research, vol. 12 (Oct): 2825−2830, 2011.

[10]Total SA, CSTJF avenue Larribau 64000 Pau – France

[11]LNAO Neurospin, Bat 145, CEA Saclay ˆ 91191 Gif sur Yvette – France

ABSTRACT

Scikit-learn is a Python module integrating a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems. This package focuses on bringing machine learning to non-specialists using a general-purpose high-level language. Emphasis is put on ease of use, performance, documentation, and API consistency. It has minimal dependencies and is distributed under the simplified BSD license, encouraging its use in both academic and commercial settings. Source code, binaries, and documentation can be downloaded from http://scikit-learn.sourceforge.net.

Keywords: Python, supervised learning, unsupervised learning, model selection

INTRODUCTION

The Python programming language is establishing itself as one of the most popular languages for scientific computing. Thanks to its high-level interactive nature and its maturing ecosystem of scientific libraries, it is an appealing choice for algorithmic development and exploratory data analysis (Dubois, 2007; Milmann and Avaizis, 2011). Yet, as a general-purpose language, it is increasingly used not only in academic settings but also in industry.

Scikit-learn harnesses this rich environment to provide state-of-the-art implementations of many well known machine learning algorithms, while maintaining an easy-to-use interface tightly integrated with the Python language. This answers the growing need for statistical data analysis by non-specialists in the software and web industries, as well as in fields outside of computer-science, such as biology or physics. Scikit-learn differs from other machine learning toolboxes in Python for various reasons: i) it is distributed under the BSD license ii) it incorporates compiled code for efficiency, unlike MDP (Zito et al., 2008) and pybrain (Schaul et al., 2010), iii) it depends only on numpy and scipy to facilitate easy distribution, unlike pymvpa (Hanke et al., 2009) that has optional dependencies such as R and shogun, and iv) it focuses on imperative programming, unlike pybrain which uses a data-flow framework. While the package is mostly written in Python, it incorporates the C++ libraries LibSVM (Chang and Lin, 2001) and LibLinear (Fan et al., 2008) that provide reference implementations of SVMs and generalized linear models with compatible licenses. Binary packages are available on a rich set of platforms including Windows and any POSIX platforms.

Furthermore, thanks to its liberal license, it has been widely distributed as part of major free software distributions such as Ubuntu, Debian, Mandriva, NetBSD and Macports and in commercial distributions such as the "Enthought Python Distribution".

PROJECT VISION

Code quality: Rather than providing as many features as possible, the project's goal has been to provide solid implementations. Code quality is ensured with unit tests—as of release 0.8, test coverage is 81%—and the use of static analysis tools such as pyflakes and pep8. Finally, we strive to use consistent naming for the functions and parameters used throughout a strict adherence to the Python coding guidelines and numpy style documentation.

BSD licensing: Most of the Python ecosystem is licensed with non-copyleft licenses. While such policy is beneficial for adoption of these tools by commercial projects, it does impose some restrictions: we are unable to use some existing scientific code, such as the GSL.

Bare-bone design and API: To lower the barrier of entry, we avoid framework code and keep the number of different objects to a minimum, relying on numpy arrays for data containers.

Community-driven development: We base our development on collaborative tools such as git, github and public mailing lists. External contributions are welcome and encouraged.

Documentation: Scikit-learn provides a ~300 page user guide including narrative documentation, class references, a tutorial, installation instructions, as well as more than 60 examples, some featuring real-world applications. We try to minimize the use of machine-learning jargon, while maintaining precision with regards to the algorithms employed.

UNDERLYING TECHNOLOGIES

Numpy: the base data structure used for data and model parameters. Input data is presented as numpy arrays, thus integrating seamlessly with other scientific Python libraries. Numpy's viewbased memory model limits copies, even when binding with compiled code (Van der Walt et al., 2011). It also provides basic arithmetic operations.

Scipy: efficient algorithms for linear algebra, sparse matrix representation, special functions and basic statistical functions. Scipy has bindings for many Fortran-based standard numerical packages, such as LAPACK. This is important for ease of installation and portability, as providing libraries around Fortran code can prove challenging on various platforms.

Cython: a language for combining C in Python. Cython makes it easy to reach the performance of compiled languages with Python-like syntax and high-level operations. It is also used to bind compiled libraries, eliminating the boilerplate code of Python/C extensions.

CODE DESIGN

Objects specified by interface, not by inheritance: To facilitate the use of external objects with scikit-learn, inheritance is not enforced; instead, code conventions provide a consistent interface. The central object is an estimator, that implements a fit method, accepting as arguments an input data array and, optionally, an array of labels for supervised problems. Supervised estimators, such as SVM classifiers, can implement a predict method. Some estimators, that we call transformers, for example, PCA, implement a transform method, returning modified input data. Estimators may also provide a score method, which is an increasing evaluation of goodness of fit: a loglikelihood, or a negated loss function.

Table 1: Time in seconds on the Madelon data set for various machine learning libraries exposed in Python: MLPy (Albanese et al., 2008), PyBrain (Schaul et al., 2010), pymvpa (Hanke et al., 2009), MDP (Zito et al., 2008) and Shogun (Sonnenburg et al., 2010). For more benchmarks see http://github.com/scikit-learn

	scikit-learn	mlpy	pybrain	pymvpa	mdp	shogun
Support Vector Classification	**5.2**	9.47	17.5	11.52	40.48	5.63
Lasso (LARS)	**1.17**	105.3	-	37.35	-	-
Elastic Net	**0.52**	73.7	-	1.44	-	-
k-Nearest Neighbors	0.57	1.41	-	**0.56**	0.58	1.36
PCA (9 components)	**0.18**	-	-	8.93	0.47	0.33
k-Means (9 clusters)	1.34	0.79	⋆	-	35.75	**0.68**
License	BSD	GPL	BSD	BSD	BSD	GPL

-: Not implemented. ⋆: Does not converge within 1 hour.

The other important object is the cross-validation iterator, which provides pairs of train and test indices to split input data, for example K-fold, leave one out, or stratified cross-validation.

Model selection: Scikit-learn can evaluate an estimator's performance or select parameters using cross-validation, optionally distributing the computation to several cores. This is accomplished by wrapping an estimator in a GridSearchCV object, where the "CV" stands for "cross-validated". During the call to fit, it selects the parameters on a specified parameter grid, maximizing a score (the score method of the underlying estimator). predict, score, or transform are then delegated to the tuned estimator. This object can therefore be used transparently as any other estimator. Cross validation can be made more efficient for certain estimators by exploiting specific properties, such as warm restarts or regularization paths (Friedman et al., 2010). This is supported through special objects, such as the LassoCV. Finally, a Pipeline object can combine several transformers and an estimator to create a combined estimator to, for example, apply dimension reduction before fitting. It behaves as a standard estimator, and GridSearchCV therefore tune the parameters of all steps.

HIGH-LEVEL YET EFFICIENT: SOME TRADE OFFS

While scikit-learn focuses on ease of use, and is mostly written in a high level language, care has been taken to maximize computational efficiency. In Table 1, we compare computation time for a few algorithms implemented in the major machine learning toolkits accessible in Python. We use the Madelon data set (Guyon et al., 2004), 4400 instances and 500 attributes, The data set is quite large, but small enough for most algorithms to run.

SVM: While all of the packages compared call libsvm in the background, the performance of scikitlearn can be explained by two factors. First, our bindings avoid memory copies and have up to 40% less overhead than the original libsvm Python bindings. Second, we patch libsvm to improve efficiency on dense data, use a smaller memory footprint, and better use memory alignment and pipelining capabilities of modern processors. This patched version also provides unique features, such as setting weights for individual samples.

LARS: Iteratively refining the residuals instead of recomputing them gives performance gains of 2–10 times over the reference R implementation

(Hastie and Efron, 2004). Pymvpa uses this implementation via the Rpy R bindings and pays a heavy price to memory copies.

Elastic Net: We benchmarked the scikit-learn coordinate descent implementations of Elastic Net. It achieves the same order of performance as the highly optimized Fortran version glmnet (Friedman et al., 2010) on medium-scale problems, but performance on very large problems is limited since we do not use the KKT conditions to define an active set.

kNN: The k-nearest neighbors classifier implementation constructs a ball tree (Omohundro, 1989) of the samples, but uses a more efficient brute force search in large dimensions.

PCA: For medium to large data sets, scikit-learn provides an implementation of a truncated PCA based on random projections (Rokhlin et al., 2009).

k-means: scikit-learn's k-means algorithm is implemented in pure Python. Its performance is limited by the fact that numpy's array operations take multiple passes over data.

CONCLUSION

Scikit-learn exposes a wide variety of machine learning algorithms, both supervised and unsupervised, using a consistent, task-oriented interface, thus enabling easy comparison of methods for a given application. Since it relies on the scientific Python ecosystem, it can easily be integrated into applications outside the traditional range of statistical data analysis. Importantly, the algorithms, implemented in a high-level language, can be used as building blocks for approaches specific to a use case, for example, in medical imaging (Michel et al., 2011). Future work includes online learning, to scale to large data sets.

REFERENCES

1. D. Albanese, G. Merler, S.and Jurman, and R. Visintainer. MLPy: high-performance python package for predictive modeling. In NIPS, MLOSS Workshop, 2008.
2. C.C. Chang and C.J. Lin. LIBSVM: a library for support vector machines. http://www.csie. ntu.edu.tw/cjlin/libsvm, 2001.
3. P.F. Dubois, editor. Python: Batteries Included, volume 9 of Computing in Science & Engineering. IEEE/AIP, May 2007.
4. R.E. Fan, K.W. Chang, C.J. Hsieh, X.R. Wang, and C.J. Lin. LIBLINEAR: a library for large linear classification. The Journal of Machine Learning Research, 9:1871–1874, 2008.
5. J. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. Journal of Statistical Software, 33(1):1, 2010.
6. I Guyon, S. R. Gunn, A. Ben-Hur, and G. Dror. Result analysis of the NIPS 2003 feature selection challenge, 2004.
7. M. Hanke, Y.O. Halchenko, P.B. Sederberg, S.J. Hanson, J.V. Haxby, and S. Pollmann. PyMVPA: A Python toolbox for multivariate pattern analysis of fMRI data. Neuroinformatics, 7(1):37–53, 2009.
8. T. Hastie and B. Efron. Least Angle Regression, Lasso and Forward Stagewise. http://cran. r-project.org/web/packages/lars/lars.pdf, 2004.
9. V. Michel, A. Gramfort, G. Varoquaux, E. Eger, C. Keribin, and B. Thirion. A supervised clustering approach for fMRI-based inference of brain states. Patt Rec, page epub ahead of print, April 2011. doi: 10.1016/j.patcog.2011.04.006.
10. K.J. Milmann and M. Avaizis, editors. Scientific Python, volume 11 of Computing in Science & Engineering. IEEE/AIP, March 2011.
11. S.M. Omohundro. Five balltree construction algorithms. ICSI Technical Report TR-89-063, 1989.
12. V. Rokhlin, A. Szlam, and M. Tygert. A randomized algorithm for principal component analysis. SIAM Journal on Matrix Analysis and Applications, 31(3):1100–1124, 2009.
13. T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Ruckstieß, and J. Schmidhuber. ¨ PyBrain. The Journal of Machine Learning Research, 11:743–746, 2010.

14. S. Sonnenburg, G. Ratsch, S. Henschel, C. Widmer, J. Behr, A. Zien, F. de Bona, A. Binder ¨ , C. Gehl, and V. Franc. The SHOGUN machine learning toolbox. Journal of Machine Learning Research, 11:1799–1802, 2010.
15. S. Van der Walt, S.C Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. Computing in Science and Engineering, 11, 2011.
16. T. Zito, N. Wilbert, L. Wiskott, and P. Berkes. Modular toolkit for data processing (MDP): A Python data processing framework. Frontiers in Neuroinformatics, 2, 2008.

CHAPTER 13

AN EFFICIENT PLATFORM FOR THE AUTOMATIC EXTRACTION OF PATTERNS IN NATIVE CODE

Javier Escalada[1], Francisco Ortin[1] , and Ted Scully[2]

[1]Computer Science Department, University of Oviedo, Calvo Sotelo s/n, 33007 Oviedo, Spain

[2]Cork Institute of Technology, Computer Science Department, Rossa Avenue, Bishopstown, Cork, Ireland

ABSTRACT

Different software tools, such as decompilers, code quality analyzers, recognizers of packed executable files, authorship analyzers, and malware detectors, search for patterns in binary code. The use of machine learning algorithms, trained with programs taken from the huge number of applications in the existing open source code repositories, allows finding patterns not detected with the manual approach. To this end, we have

Citation: Javier Escalada, Francisco Ortin, and Ted Scully, "An Efficient Platform for the Automatic Extraction of Patterns in Native Code," Scientific Programming, vol. 2017, Article ID 3273891, 16 pages, 2017. doi:10.1155/2017/3273891.

created a versatile platform for the automatic extraction of patterns from native code, capable of processing big binary files. Its implementation has been parallelized, providing important runtime performance benefits for multicore architectures. Compared to the single-processor execution, the average performance improvement obtained with the best configuration is 3.5 factors over the maximum theoretical gain of 4 factors.

INTRODUCTION

Many software tools analyze programs, looking for specific patterns defined beforehand. When a pattern is found, an action is then performed by the tool (e.g., improve the quality, security, or performance of the input program). Patterns are defined for both high-level and binary code. Different examples of tools that analyze high-level patterns are refactoring tools, code quality analyzers, or detectors of common programming mistakes. In the case of binary code, examples are decompilers, packed executable file recognizers, authorship analyzers, or malware detectors.

In the traditional approach, an expert identifies those patterns to be found by the software tool. His or her expertise is used to define the code patterns that should be searched (e.g., for improving the application). On the contrary, a machine learning approach can be used to build models, which can then be applied to large repositories of code to effectively search for and identify specific patterns. This second approach allows the analysis of huge amounts of programs, sometimes detecting patterns not found with the traditional approach. Additionally, this approach could automatically evaluate the accuracy of the patterns obtained.

An emerging research topic called big code has recently appeared [1]. Big code is based on the idea that open source code repositories (e.g., GitHub, SourceForge, BitBucket, and CodePlex) can be used to create new kinds of programming tools and services to improve software reliability and construction, using machine learning and probabilistic reasoning. One of the research lines included in big code is finding extrapolated patterns, detecting software anomalies, or computing the cooccurrence of different patterns in the same program [2].

One example (more real examples can be consulted in [2]) of detecting patterns in programs is the vulnerability discovery method proposed by the MLSEC research group [3]. It assists the security analyst during auditing of source code, by extracting ASTs from the programs and determining structural patterns in them. Given a known vulnerability, their patterns are identified and extrapolated to a code base, such that functions potentially suffering from the same flaw can be suggested to the analyst. With that method, they managed to identify 18 previously unknown vulnerabilities in the source code of the Linux kernel.

The research context of finding code patterns using machine learning algorithms is based on 3 ideas. First, machine learning techniques can be used to build predictive models to identify patterns in data; besides, these techniques do not require domain specific knowledge about the problem domain [4]. Second, the existing open source code repositories create new opportunities for gathering massive program repositories to be analyzed. Third, the existing big data technologies and platforms facilitate the analysis of large datasets.

When processing native code, a platform to extract binary patterns should also be able to use the debug information (if any) generated by the compiler. This information would be very valuable to extract those binary patterns, which may later be used by a machine learning algorithm to create predictive tools. A large number of patterns may be extracted from a small binary program, since the number of assembly instructions is much higher than in its source high-level program. Therefore, the need of processing debug information, plus the potentially huge number of patterns to be extracted, makes it critical to use highly parallelized and efficient tools for extracting those patterns.

A platform capable of extracting patterns from programs should also be highly parameterized. The individuals (rows in the dataset generated) to be detected by the platform must be defined by the user. For instance, we may be interested in finding patterns for functions, snippets, function entry points, or specific regions of binary code. The same parameterization is also required to specify the features of each individual (columns in the dataset). For example, we may define the feature <mov> <generic ax>,<any> to represent the occurrence of any assembly instruction that moves any value to the accumulator register (ah, al, ax, eax or rax). If that pattern (feature or column) occurs in one given region of binary code (individual or row), then the corresponding value in the dataset (row and column) will be 1.

The traditional method to extract features from binary code is to identify a syntactically fixed unit of code, such as functions or basic blocks, and extract the binary code inside them [5]. However, pattern extraction does not always follow this scheme. Sometimes, nonsequential patterns such as subgraphs of control flow and data dependency graphs need to be extracted. In these cases, a binary pattern extraction platform should allow the association of patterns to pieces of code outside their basic blocks, representing subgraph structures (Section 3.7). Another scenario where the traditional method is not sufficient is the analysis of binary code between two memory addresses, where the inconsistency overlapping problem caused by the variable-length instruction set must be tackled [6]. This kind of binary code analysis has been used for different purposes such as function entry points detection [6], compiler recognition [7], authorship attribution [8], and malware detection [9, 10]. Therefore, a generic platform for pattern extraction must be flexible enough to support any binary pattern extraction method (not just the traditional one) and reduce development and execution times.

The main contribution of this paper is a platform for the automatic extraction of patterns in native code. The platform is highly parameterized so that it could be used in different scenarios. Its parallel implementation provides important runtime performance benefits when multicore architectures are used. It also uses the debug information that may be provided by a compiler. The extracted patterns may be used by other tools for different purposes. We present an evaluation of binary pattern extraction, measuring the execution time of different configurations for a large number of programs. The parallelization provides significant performance improvements, and its efficiency is maintained for big volumes of programs.

The rest of this paper is structured as follows. Section 2 describes a motivating example, and the platform is described in Section 3. Section 4 presents an evaluation of the platform and Section 5 discusses the related work. The conclusions and future work are presented in Section 6.

MOTIVATING EXAMPLE

We use a motivating example to explain our platform. The example is the extraction of patterns in native code that can be later used to improve the information inferred by a decompiler. A decompiler extracts high-level information from a native program, aimed at obtaining the original source program used to generate the native code. Existing decompilers are able to infer part of this information. However, some elements of the original high-level source programs are not inferred by any decompiler.

Algorithm 1 shows a C function that returns a string (char in C). The function returns a substring from the str parameter, starting in the beginth position of str up to the endth position. The values of begin and end could be negative, following the slicing behavior of the Python [] operator.

Algorithm 1: Example high-level C program.

```
char * str_slice(const char * str, int begin, int end) {
    /* Check str */
    if (str == NULL)
        return NULL;
    /* Check ranges */
    int s = strlen(str);
    if ((!s) || (begin >= s) || (begin < s * -1) || (end >= s + 1)
            || (end < s * -1))
        return NULL;
    /* Normalice values */
    size_t n_begin = begin >= 0 ? begin: s + begin;
    size_t n_end = end >= 0 ? end: s + end;
    /* Check begin >= end */
    if (n_end <= n_begin)
        return NULL;
    /* Alloc mem */
    size_t amount = n_end - n_begin;
    char * new_str = malloc((amount + 1) * sizeof(char));
    if (new_str == NULL)
        return NULL;
    /* Copy */
    memcpy(new_str, str + n_begin, amount);
    new_str[amount] = 0;
    return new_str;
}
```

After compiling the str_slice function with Microsoft's cl compiler, the decompiled function generated by Hex-Ray 1.5 has the following signature:

int __cdec1 sub_401780 (int al, int q2, int a3)

We can see how the original return type of the function (char) is not the same as the one obtained by the decompiler (int). This is because, in native machine code, there is no type difference between integers and pointers. The difference between both might be obtained by analyzing how the programmer uses the value returned by the function. In general, the programmer performs indirections with pointers, but not with integers. Since this rule is not always fulfilled, the decompiler does not tell the difference between these two types.

By analyzing the usage patterns of each variable, a decompiler may infer the actual high-level type of the variables (and functions). Since this is not a deterministic mechanism, the use of machine learning seems to be appropriate for this kind of problems [11]. Some recognized limitations of most decompilers include the following:(i)Types of variables, including function arguments and return types (e.g., the example in Algorithm 1) [12].(ii)Functions: the identification of function entry points is a complex task, mainly due to indirect function invocations [6]. Similarly, detecting the function body is an open challenge because its instructions may not be contiguous, have multiple entry points, be in-line, or not be reachable [5].(iii)Control flow structures: its recognition is commonly based on control flow graph (CFG) analysis [12]. However, CFGs might not be completely recovered by static analysis if indirect jumps appear, which are typically generated for switch structures [13].(iv)Elements of a specific paradigm: when another paradigm different to the structured one (e.g., object-orientation or functional) is used, the specific elements of that paradigm are barely recognized. For example, C++ decompilers commonly fail in the reconstruction of polymorphic classes, class hierarchies, member functions, and exception handling constructs [14].

The platform presented in this paper is being used to extract binary patterns from native code, which are later used to improve certain high-level information gathered by existing decompilers. Particularly, we face the problem of detecting the type returned by a function. An excerpt of the dataset generated by our platform for this particular case is shown in Table 2: individuals (rows) are functions in the module; features (columns) are sequences of binary patterns found at the end of the function body (return patterns) and after invoking the function (call post patterns); the target column is the returned type. Please, notice that the work of this paper is the platform itself, not in the algorithm for decompilation improvement.

PLATFORM ARCHITECTURE

This platform generates one dataset table to classify fragments of binary code (individuals or rows in the dataset) by considering the occurrence of a finite set of binary patterns (features or columns in the dataset). All the individuals and patterns (features) are obtained from a collection of binary programs, which are processed by the platform.

In our motivating example, the individuals in the dataset are functions; and the features are the generalized assembly code patterns extracted by the platform. The classification variable (the target) is the return type (we consider all the C built-in types; for compound types (structs, unions, pointers, and arrays), we only consider the type constructor, e.g., int* and char** are classified as pointers) of each function (individual). The generated dataset may be used later to build a machine learning model that classifies the return type depending on the patterns found in the binary code.

The platform has two working modes. The most versatile is the one shown in Figure 1. The system receives the high-level source program that will be used to generate the binary application. In this mode, the platform allows instrumenting the high-level program and uses the debug information produced by the compiler. When the high-level program is not available, we provide another configuration to process binary files, described in Section 3.6.

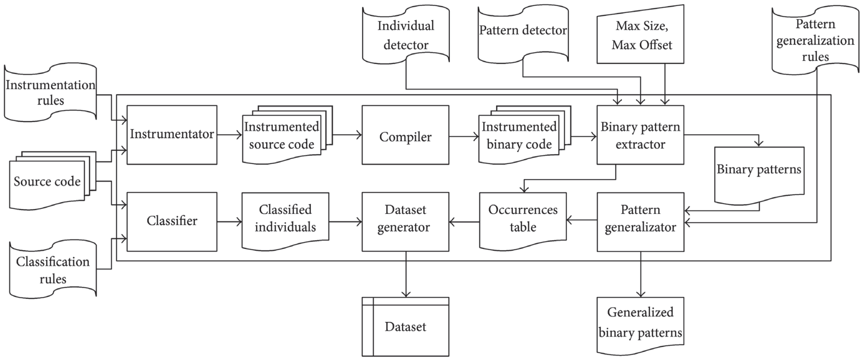


Figure 1: Platform architecture, receiving high-level code.

Instrumentator

This module allows code instrumentation of the high-level input program. The objective is to add information to the input program, so that it will be easier to find the patterns in the corresponding binary code generated by the compiler. It can also be used to delimitate those sections of the generated binary code we want to extract patterns from (Section 3.2), ignoring the rest of the program. Notice that once the machine learning model has been trained with the dataset generated by the platform, the binary files passed to the model will not include that instrumented code. Therefore, the instrumentation module should not be used to extract patterns that cannot be later recognized from stripped binaries.

This module traverses the Abstract Syntax Tree (AST) of the Source Program and evaluates the Instrumentation Rules provided by the user. Traversing the AST, if the precondition of one instrumentation rule is fulfilled, its corresponding action is executed. The action will modify the AST with the instrumented code, which will be the new input for the compiler (next module in Figure 1).

In our motivating example, we have defined the instrumentation rule shown in the pseudocode in Algorithm 2 (in Section 4.1 we describe how they are implemented). For all the return statements in a program, the rule adds a dummy label before the statement. This label has the function identifier (id_{func}) followed by a consecutive number (a function body may have different return statements).This label will be searched later in the binary code (using the debug information) to know the binary instructions generated by the compiler for the return high-level statements. These binary instructions will be used to identify the binary patterns (Section 3.2).

Algorithm 2: Instrumentation rule for return statements.

```
Function **return_instrumentation**(*program*)
  **for all** *stmt* **in** *program* **do**
    **if** *stmt* **is** $type_{return}\ id_{func}((type_{arg}\ id_{arg})^*)stmt_{body}{}^*$ **then**
      $labels \leftarrow 1$
      **for all** *stmt* **in** $stmt_{body}{}^*$ **do**
        **if** *stmt* **is** return *exp* **then**
          $stmt \leftarrow$ __RETURN_$id_{func}$_$labels$__: $stmt$
          $labels \leftarrow label + 1$
        **end if**
      **end for**
    **end if**
  **end for**
**end**
```

Algorithm 3 shows another example of one instrumentation rule. Recall that the previous instrumentation rule was aimed at finding binary instructions between a __RETURN_ label and a RET assembly instruction. However, C functions returning void usually do not have an ending return statement. Therefore, the instrumentation rule in Algorithm 3 adds both the expected label and the return statement.

Algorithm 3: Instrumentation rule for procedures.

```
Function procedure_instrumentation($program$)
  $labels \leftarrow 1$
  for all $stmt$ in $program$ do
    if $stmt$ is $type_{return}\ id_{func}\ ((type_{arg}\ id_{arg})^{*})\ stmt_{body}^{*}$
          and $type_{return}$ = void then
      $stmt_{body}^{*} \leftarrow stmt_{body}^{*}$; __RETURN_$id_{func}$_$labels$__: return;
      $labels \leftarrow labels + 1$
    end if
  end for
end
```

Adding labels is an easy way to instrument code. However, more sophisticated approaches can be used. For example, expressions may be translated into dummy function invocations that are actually used as marks to be identified in the pattern extraction phase (Section 3.2). Another typical approach is adding innocuous sequences of assembly instructions (e.g., NOPs) to be found in the pattern extraction phase. The user must be careful when selecting the instrumentation approach, checking that the instrumented code does not produce unexpected changes to the generated binaries, or to the patterns he/she wants to extract.

With the Instrumentation Rules, the source code is translated into instrumented code. The instrumented code is then compiled, producing the Instrumented Binary Code.

Binary Pattern Extractor

This module performs 3 tasks. First, it identifies the binary code fragments representing the individuals (rows) in the generated dataset. Second, it extracts the binary patterns (columns) detected for each individual. These patterns are used as features to later classify the individuals. The third task is to store the individuals and patterns in an Occurrence Table, which will be later used to generate the final dataset. We now detail these 3 tasks.

The Individual Detector initially recognizes each individual in the binary code. It must implement a function to collect all the individuals. Algorithm 4 shows the Individual Detector of our example, recognizing functions as individuals. In the figure, is_function returns whether the parameter is the first instruction in a function, using the debug information generated by the compiler. Once one function is detected, its label is added to the individuals list, the returned value.

Algorithm 4: Individual detector to recognize functions.

```
Function individual_detector(program)
    individuals ← []
    instruction ← program[0]
    repeat
        if is_function(instruction) then
            individuals ← individuals + label(instruction)
        end if
        instruction ← next(instruction)
    until not next(instruction)
    return individuals
end
```

After identifying the individuals, we must extract the binary patterns we are looking for. To this end, the user should provide a Pattern Detector, which comprises a collection of predicate functions. These functions receive one instruction of the instrumented binary program. In case that instruction is not included in the expected pattern, null must be returned. If the pattern is identified, a pair containing the individual and the range of instructions in the pattern (another pair) is returned.

Algorithm 5 presents a Pattern Detector of our example. It recognizes the return pattern added by theInstrumentator. If the instruction label is __ RETURN, the Pattern Detector recognizes the pattern. The corresponding function is returned as the first element of the pair. The second one is the range of instructions comprising the pattern: the first one (the one labeled __RETURN) and the next instruction after the following RET.

Algorithm 5: Pattern detector rule to recognize RET patterns.

```
Function RET_pattern_detector(instruction)
    if instruction is not __RETURN_$id_{func}$_n then
        return null
    end if
    begin_instruction ← instruction
    while not instruction is RET do
        instruction ← next(instruction)
    end while
    return ($id_{func}$, (begin_instruction, next(instruction)))
end
```

Algorithm 6 shows another Pattern Detector used in our example. It detects as a pattern the instructions after one CALL (we call it call post). In this case, the individual associated with the pattern is not the function the instruction belongs to, but the function being called. Similarly, we have also specified a pattern with the instructions before CALL, called call pre, not shown in the algorithm. The idea of these two patterns is that the usage of the value returned by a function (call post) and the code to push its parameters (call pre) may be valuable to infer the types of the function signature (return and parameter types).

Algorithm 6: Pattern detector rule to recognize call post patterns.

```
Function CALL_Post_pattern_detector(instruction)
    if instruction is CALL $id_{func}$ then
        begin_instruction ← instruction
        for $i = 0$ to $MaxSize + MaxOffset$ do
            instruction ← next(instruction)
        end for
        return($id_{func}$, (begin_instruction, instruction))
    end if
    return null
end
```

At this point, the module has three types of extracted patterns: ret patterns, including the assembly code of return statements, and call pre and call post patterns, representing the code before and after invoking a function. Each of these patterns may include a significant number of contiguous binary instructions. However, we could be interested in a small portion of contiguous instructions inside the bigger patterns. For this reason, the Binary Extractor Pattern has been designed to divide the patterns found into a collection of subpatterns (different partitions of the original pattern).

The algorithm to obtain the subpatterns is parameterized by the Max Size and Max Offset parameters shown in Figure 1. This algorithm starts with one-instruction length subpatterns ($size = 1$), increasing this value up to Max Size contiguous instructions. Additionally, other subpatterns are extracted leaving offset instructions between the instruction detected by the Pattern Detector and the subpatterns. The algorithm described above (the one that increases size) was for offset = 0. The same algorithm is applied for offset = 1 and offset = −1 (i.e., the first instruction before and after the detected instruction, which is not included in the subpattern). The absolute value of offset is increased up to Max Offset.

Figure 2 shows 4 example subpatterns. From a call pre pattern the $size$ = 5 and offset = 0 and $size$ = 2 and offset = −2 are shown. From another call post pattern, Figure 2 displays the $size$ = 3 and offset = 2 and $size$ = 3 and offset = 1 subpatterns.

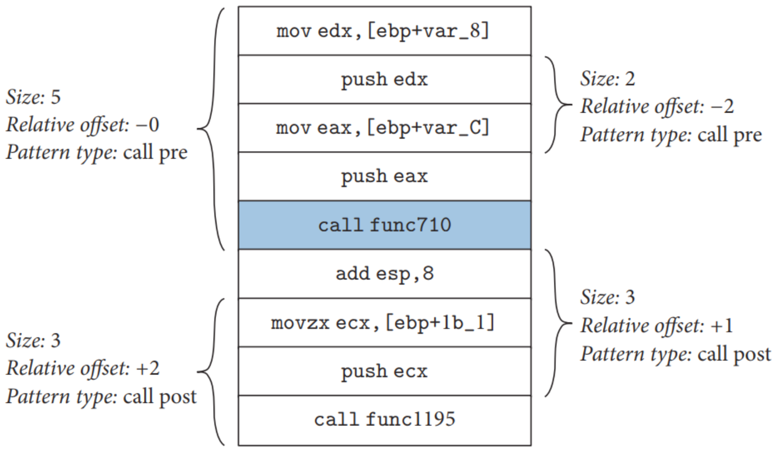


Figure 2: Example of 4 subpatterns extracted from 2 patterns.

The last task to be undertaken by the Pattern Detector is to associate the individuals with their patterns and make this association explicit by writing the Occurrences Table. This process is done generating as many table rows as individuals found by the Individual Detector (in Algorithm 4), and associating them with the rows representing each of the subpatterns found for that individual by the Pattern Detector functions (Algorithms 5and 6).

Pattern Generalizator

Sometimes, the subpatterns found are too specific. For example, the MOV eax,5 and MOV ax,1 subpatterns are recognized as two different ones. However, for detecting whether a function is returning a value or not, they may be considered as the same pattern, meaning that a literal has been assigned to the accumulator register (i.e., a MOV <generic ax>,<literal> pattern). To this end, the objective of the Pattern Generalizator module is to allow the user to reduce the number of subpatterns, by generalizing them.

This necessity of generalizing (or normalizing) assembly instructions for binary pattern extraction was already detected in previous works. In [6], the * wildcard matches any one instruction, and the absence of an operand means any value. In further works, they also identify the necessity of eliding memory addresses and literal values [7]. The generalization proposed by Bao et al. uses regular expressions to generalize literal values and even instruction mnemonics [5]. Another coarser normalization just ignores all the operands of assembly instructions [15].

To identify the generalization requirements of a generic platform, we analyzed the decompiler case scenario described in Section 2. Some examples of those generalizations are shown in Table 1. First, the user should be able to generalize instruction operands, including literals, registers, variables, and memory addresses. Second, the platform should allow the generalization of instructions with the same purpose. Finally, the user may need to generalize variable-instruction-length subpatterns, such as function caller and callee epilogues.

Table 1: Example generalization of subpatterns

	Example pattern	Generalization
Operand	mov 5,eax	mov <literal>,<generic ax>
	mov [ecx],al	mov [ecx],<register>
	movsd xmm0,var_0	movsd xmm0,<var>
	mov edx,[ebp+var_1]	mov edx,[<var>]
	call func1493	call <address>
Mnemonic	movzx eax,al	<mov> <generic ax>,<any>
	movss [esp+54h+var_2],xmm0	<mov>[esp+54h+var_2],xmm0
	movsd xmm0,var_3	<mov> xmm0,<var>
	mov edx,[ebp+var_4]	<mov> edx,[<var>]
	movsx ecx,[ebp+var_5]	<mov> ecx,[<var>]
Instruction group	pop esi; mov esp,ebp; pop ebp; retn	<callee epilogue>
	mov esp,ebp; pop ebp; retn	<callee epilogue>
	pop ebp; retn	<callee epilogue>
	call func123; add esp,8	<caller epilogue>
	call func123	<caller epilogue>

Table 2: Example dataset generated to predict the returned type of functions

	(Return pattern) `<mov> <generic ax>,<any>; <callee epilogue>;`	*(Call post pattern)* `<caller epilogue>; Fld <any>;`	...	Return type (target)
func_710	1	0		int
func_1195	0	1		double
func_295	0	0		long long
⋮				

The analysis of the decompiler use case indicates that a highly expressive generalization mechanism should be provided by a generic binary pattern extraction platform. For instance, it should allow the generalization of variable-length groups of instructions, not supported by the existing approaches. Therefore, we propose a programmatic system that takes advantage of the expressiveness of a full-fledged programming language to describe those generalizations.

In our platform, generalizations are expressed as Pattern Generalization Rules. As shown in Algorithm 7, those rules are implemented as functions receiving one instruction and returning their generalized pattern (the current instruction if no generalization is required) and the following instruction to be analyzed. This second value allows the implementation of variable-instruction-length generalizations. The rule in Algorithm 7generalizes the move instructions that save into the accumulator register any value.

Algorithm 7: Pattern generalization rule of move instructions.

```
Function MOVE_generalization(instruction)
  if instruction.mnemonic in [mov, movzx, movsd, movss, movsx]
            and instruction.operands[1].type = register
            and instruction.operands[1].value in [eax, ax, ah, al] then
    instruction.mnemonic ← <mov>
    instruction.operands[1].value ← <generic ax>
    instruction.operands[2].value ← <any>
  end if
  return (instruction, next(instruction))
end
```

The generalized patterns and their associations with the individuals are added to the existing Occurrence Tableproduced by the Binary Pattern Extractor.

Classifier

This module is aimed at computing the value of the classifier variable (i.e., the target or the dependent variable) for each individual. The input is a representation of the high-level program; the output is a mapping between each individual and the value of the classifier variable. These associations are described by the user with theClassification Rules.

Algorithm 8 shows one Classification Rule for our example. We iterate along the statements in the program. For each function, we associate its identifier with the returned type, which is the classifier variable for our problem (we predict the return type of functions).

Algorithm 8: Classification rule associating each function with its return type.

```
Function function_classification(program)
    individuals ← {}
    for all stmt in program do
        if stmt is $type_{return}$ $id_{func}$ $((type_{arg}\ id_{arg})^*)$ then
            individuals ← individuals[$id_{func} \mapsto type_{return}$]
        end if
    end for
    return individuals
end
```

Dataset Generator

Finally, the Dataset Generator generates the dataset from the Occurrence Table (Section 3.3) and the individual classification (Section 3.4): one row per individual, one column per subpattern (generalized or not), and another row for the classifier variable. Cells in the dataset are Boolean values indicating the occurrence of the subpattern in the individual. Classifier or target cells may have different values. Table 2 shows an example dataset.

Processing Binary Files

As mentioned, the platform has two working modes. Many times, we do not have the high-level source program used to generate the native code, and we are interested in finding patterns in binary files. Different examples of this scenario include authorship, compiler, and malware detection.

In order to show this second working mode of our platform, we use the research work done by Rosenblum et al. [6] as an example. They extract patterns from stripped binary files to detect function entry points (FEP), which existing dissemblers do not detect perfectly yet [5]. They analyze consecutive bytes in binary files, representing them as 3 grams of assembly instructions. Once the 3 grams are extracted, they formulate the FEP identification problem as structured classification using Conditional Random Fields (CRF) [16]. An initial flat model is later enriched with the evidence that a call instruction indicates the existence of a FEP in the callee address. The model obtained detects FEPs more accurately than GCC, ICC, and MSVS compilers [6].

Figure 3 shows the changes to the platform architecture when we want to process binary files, and the high-level program is not available. White elements are the same as in the previous architecture. Blue elements are modifications of the previous working mode. All the modules related to processing high-level programs are not present.

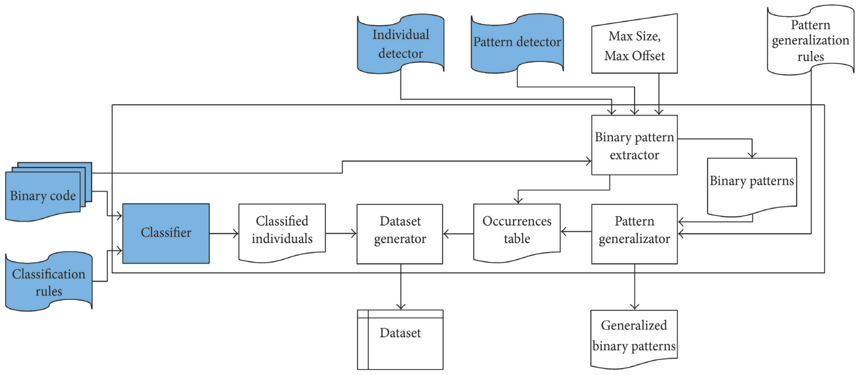


Figure 3: Platform architecture to process binary code.

Although the behavior of the Binary Pattern Extractor is the same, the rules for detecting individuals and patterns are different. The main difference is that no instrumented code is added, since the source code is not available. Depending on the case, debug information is not available either (i.e., stripped binaries are used). Regarding the Classifier module, the Classification Rules must consider a plain binary file instead of a high-level program representation.

In the example of FEP detection in binary files, this is how the platform has been used to generate a dataset valid to create the CRF model. In the output dataset, individuals (rows) are instruction offsets in the binary file; one feature (column) will be created for each 1, 2, and 3 grams in the binary code, indicating the occurrence of that pattern in each individual; another call <offset> feature is added, associating that function invocation with the <offset> individual. Finally, the classifier variable (target) is 1 if the individual is a FEP and 0 otherwise (debug information is available).

In order to create the dataset described above, the Individual Detector creates as many individuals as instruction offsets in the binary file. The Pattern Detector extracts 1, 2, and 3 grams for each offset and a call feature for each different function. In this second case, the feature is not associated with the offset where the pattern is detected, but to the offset (memory address) being called (as done in Algorithm 6). Pattern Generalization is done as the normalization process described in [6]. Finally, the Classification Rules use the debug information to set 1 to one individual identified as a FEP and 0 otherwise.

This platform configuration (and the previous one) to extract datasets valid to create the CRF model proposed by Rosenblum et al. is available for download at [17].

Representing Nonsequential Patterns

In the analysis of binary applications, it is common to require the detection of nonsequential patterns, such as subgraphs of control flow and data dependency graphs. The detection of these subgraphs can be used for many different purposes, such as the FEP detection problem described in the previous subsection.

Although the Binary Pattern Detector module of our platform (Figures 1 and 3) is aimed at extracting patterns made up of contiguous binary instructions, the rest of the modules can be used to represent nonsequential structures such as graphs. This functionality is provided by the versatile way our platform considers the sequential patterns (features), permitting the definition of different criteria to associate these features to the corresponding individuals.

One example of this functionality is present in the decompiler scenario. Algorithm 5 shows how RET features are associated with the function (individual) where the pattern was detected. In Algorithm 5, this association is represented by the first element in the tuple returned, which is the function id the RET instruction belongs to. Thus, the output dataset will have 1 in the cell corresponding to that function (row or individual) and pattern (column or feature). However, CALL patterns are associated with individuals in a different way. Algorithm 6shows how this type of feature is not associated with the function where the pattern is detected, but to the function being invoked. Therefore, a machine learning algorithm trained with the generated dataset may associate nonsequential patterns (e.g., there must exist a RET pattern inside the function and, in any part of the program, a CALL pattern invoking the same function) to identify the type returned by a function.

Another example of this functionality is the FEP identification problem described in Section 3.6. The dataset generated by out platform can be used to create the proposed CRF model, which uses graphs for structural prediction and classification [16]. Those graphs are obtained from the dataset by using the versatile association of features to individuals already discussed. Its implementation and a sample dataset can be consulted in [17].

EVALUATION

Platform Implementation

We have implemented the proposed platform and it is freely available at http://www.reflection.uniovi.es/decompilation/download/2016/sp/. The Instrumentator and Classifier modules have been implemented in C++, since they use clang [18] to process the high-level representation of C programs. The rest of the platform has been implemented in Python. For the disassembly services we have used IDAPython [19].

The implementation is highly parallelized, providing important performance benefits when multicore architectures are used. The parallelization follows a pipeline scheme, where both data and task parallelism are used. Figure 4 shows the concrete approach followed. These have been the issues tackled to parallelize the platform implementation.

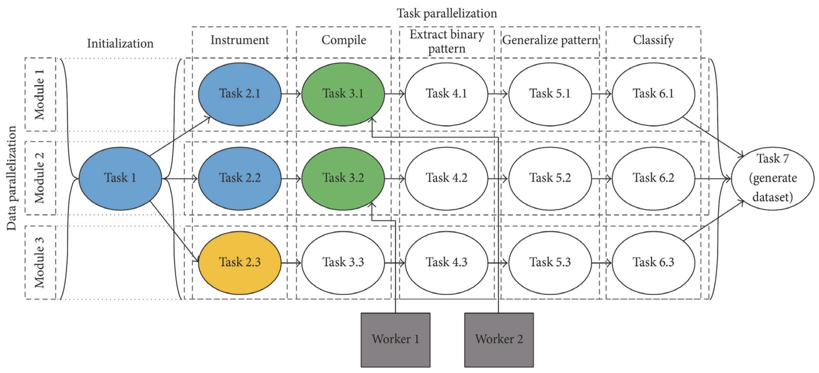


Figure 4: Parallelization of the platform implementation.

(1) Data Parallelization. We identify each module in a program (obj files in the compiler used) as a different portion of data to work in parallel. This obj files can be combined in lib or exe files to produce bigger modules. In the example in Figure 4, three different modules are processed in parallel.

(2) Task Identification. The tasks to be parallelized are those identified as modules in the platform architecture (Section 3). As shown in Figure 4, an additional initialization task was defined to initialize the database and create a temporary folder where the input files are copied.

(3) Task Dependency. After identifying the tasks, we defined the dependencies among them with a Directed Acyclic Graph (DAG). These dependencies define when two tasks can run in parallel, and when a task has to wait for others to end. As shown in Figure 4, the instrumentation, compilation, binary pattern extraction, generalization, and classification tasks can run in parallel. For the same piece of data, one has to wait for the previous one to complete. The initialization (at the beginning) and dataset generation (at the end) tasks cannot be parallelized. The last one waits for all the classification tasks to process all the data.

(4) Task Implementation. Tasks should be mapped to threads or processes. The current implementation uses the Python programming language to combine all the different modules of the architecture (implemented in Python itself or C++). Since most implementations of Python use the Global Interpreter Lock (GIL) to synchronize the execution of threads [20], we implemented

tasks as processes to obtain a better runtime performance improvement with multicore architectures [21].

(5) Concurrent Workers. To parameterize the level of parallelization of the platform, we configured its implementation to run with a different number of worker processes (Section 4.2). A scheduler analyzes the task DAG and tells each worker which is the following task to be executed. In Figure 4, two workers are running in parallel. Tasks 1, 2.1, and 2.2 have already been executed; Tasks 3.1 and 3.2 are run by Workers 1 and 2, respectively; and Task 2.3 is the following one to be executed, once one worker is free.

(6) Communication between Tasks. Since we implemented tasks as processes, communication between them is costly. However, the dependency between tasks shown in Figure 4 indicates that the output of one task is taken as the input of the following one. Therefore, this data communication was implemented through a database, appropriately configured to obtain the expected runtime performance.

(7) Task Synchronization. Workers should indicate when they terminate executing one task, and the scheduler should tell them which task should be executed next. To synchronize this process, we used a Queue object in the multiprocessing module.

(8) Tool Parameterization. We configured the IDA disassembler to allow the concurrent processing of the same input file. The compilation task is represented with a Python class that can be parameterized to use different compilers, package managers, compiler options, and automating software. The external tools used write information in the standard output (e.g., the C compiler). We captured those messages and sent them to a concurrent logger, adding additional information of the processes.

Methodology

The runtime performance of our platform depends on the following variables:(i)Number or independent modules of compilation (or programs). We may process different programs in parallel, or different modules of the same program, to create a dataset.(ii)Number of workers: as mentioned, the platform may run different tasks at the same time. A task is run by a worker. Depending on the number of real processors, the number of workers may produce an important benefit on runtime performance.(iii)The number of cores: we have run our platform with different multicore computers.(iv)The size of each program (or module), according to the number of individuals it may contain.(v)Subpattern extraction: as described in Section 3.2, different subpatterns are automatically extracted from the patterns found. The Max Size and Max Offset parameters have influence on the execution time.(vi) The number of patterns: the proposed platform recognizes patterns by means of the Pattern Detectorfunctions specified by the user. We analyze runtime performance depending on the number of patterns defined.

We evaluate the influence of these variables on the runtime performance of the platform, and how they are related to the parallelization level. In order to evaluate that, we fix all the variables except one and measure the runtime performance for different values of the free variable [22]. This process is repeated for all the variables.

We evaluate the platform with the real example of predicting the return type in binary programs, using their C source code (the first working scheme of our system, shown in Figure 1). We extract return, call pre, and call post patterns, divide them into different subpatterns, and perform a generalization of the subpatterns found.

The programs used for the experiments were synthetically generated by a C program generator. It was very helpful to generate a rich battery of programs. Besides, we were able to generate different configurations of the same program, changing the number of individuals (functions in our example) per module. In this way, we do not introduce the bias of measuring different programs.

In order to be able to change the number of cores, all the tests were carried out on a Hyper-V virtual machine with 4 processors and 8 GB of RAM, running an updated 64-bit version of Windows 8.1. The host computer was a 3.60 GHz Intel Core i7-4790 system with 16 GB of RAM, running an updated 64-bit version of Windows 10. The tests were executed after system reboot, removing the extraneous load, and waiting for the operating system to be loaded [23].

Increasing Number of Modules

In this first experiment, we increase the modules in a program from 1 to 8, fixing the number of cores and workers to 4. For this experiment and the following ones, the value of Max Size is 4 and Max Offset is 0. We also extract return, call pre, and call post patterns.

The program to be analyzed has 10,000 functions (individuals), so we have 1 module with 10,000 functions, 2 modules with 5,000 functions, and so on, up to 8 modules with 1,250 functions. Therefore, all the configurations have the same dataset with 10,000 functions, and the processed program is the same.

Figure 5 shows the benefits of parallelization. The execution time of processing the same program drops when the number of modules is increased until 4 modules (the number of cores and workers). In that point, the platform processes the program 3.33 times faster than the same program with one module (i.e., the sequential implementation). According to Amdahl's law, the maximum theoretical performance benefit for that configuration is 4 factors [24].

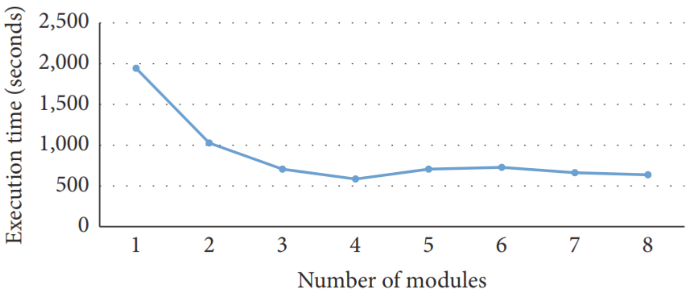


Figure 5: Execution time for an increasing number of modules.

For more than 4 modules, there is no significant benefit, since there are only 4 cores in this configuration. Besides, there is no penalty for 8 programs, showing that a number of programs higher that the number of cores do not cause a significant penalty. The slight worsening for 5, 6, and 7 programs is caused by the selection of 4 workers and cores. After processing 4 programs in parallel, the processing of the fifth one makes the rest of the workers wait for completion, causing a slight performance drop.

Increasing Number of Workers

In this case, the number of workers goes from 1 to 8, fixing the number of cores and modules to 4. Each module has 2,500 functions (10,000 for the whole program).

Figure 6 shows how execution time is reduced as the number of workers increases. With 4 workers, the platform reaches the lowest value, 3.5 times faster than the sequential execution. For 5 workers or more, there is no benefit because those extra workers keep waiting for tasks to end.

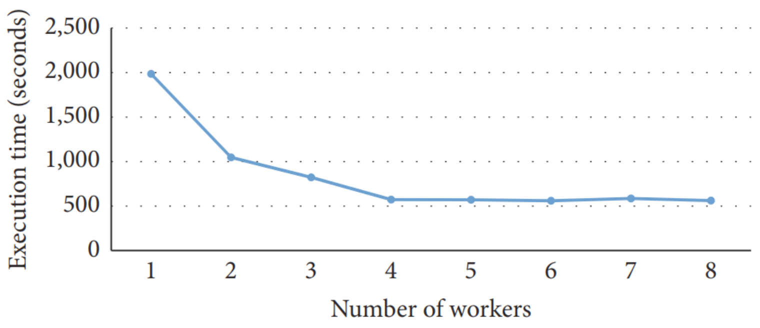


Figure 6: Execution time for an increasing number of workers.

Increasing Number of Cores

In this case we change the number of cores of the virtual machine configuration. Fixing the configuration to 4 workers and modules, we increase the number of cores from 1 to 4. We have not used more cores because, in the computer used (see Section 4.2), the virtualization software drops its performance with 5 cores or more. The number of individuals per module is 2,500.

We can see in Figure 7 how our platform takes advantage of multicore architectures. The computer with 4 cores runs 3.7 times faster than the one with one single core. The benefit is close to the maximum theoretical one [24].

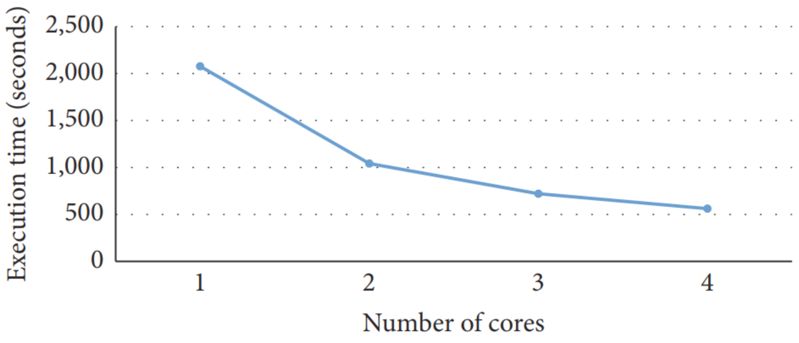


Figure 7: Execution time for an increasing number of cores.

Increasing Number of Modules and Workers

This experiment increases two variables at the same time. It is intended to represent a typical use case scenario. Assuming we have a multicore computer (4 cores in our case), we set the number of workers equal to the number of modules (or programs) to be processed. The idea is to try to obtain the higher level of parallelization with a given computer. Therefore, we increase the number of modules and workers from 1 to 16. The number of functions is always 10,000, equally distributed over the different modules of the program.

Figure 8 shows how execution time keeps reducing until 4 modules and workers (3.5 factors of benefit). From 4 to 7, differences among the values are lower than 1% (practically the same values). With 8 and beyond, the figure displays a slight increase of execution time due to the cost of context switching. Therefore, the results of the experiments seem to indicate that the optimal value for workers and modules range from the number of cores to twice this value.

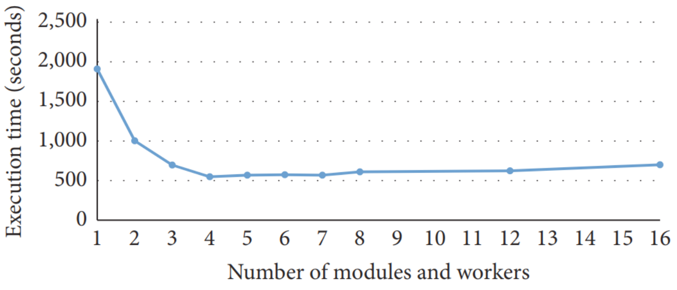


Figure 8: Execution time for an increasing number of modules and workers.

Increasing Number of Functions

In order to see how the platform behaves for increasing sizes of programs, this experiment increases the number of functions in the program from 1,000 to 15,000. We selected this maximum value because it was the biggest program supported by the IDA disassembler. The number of cores and workers is 4.

Figure 9 shows the linear increase of runtime performance depending on the number of functions (i.e., the size of the programs). Besides, it supports the analysis of really big modules with 15,000 functions.

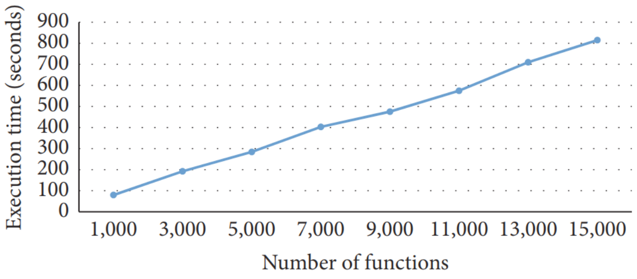


Figure 9: Execution time for an increasing number of functions.

Figure 10 presents another view of the same data. That figure displays the execution time performance per function, increasing the number of functions in the program. For small programs, there is an initialization penalty causing a higher execution time to process a low number of functions. When the program size grows, this initialization cost becomes negligible. From 5,000 functions on, the execution time per function converges (the standard deviation is lower than 3.4%), showing that the performance of the platform is not decreased for big input programs.

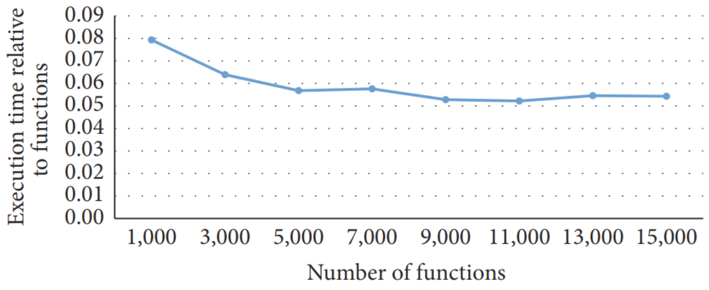


Figure 10: Execution time per function, increasing the number of functions.

Increasing Max Offset and Max Size

We now modify the values of the Max Offset and Max Size parameters used to obtain the binary subpatterns. We used 4 modules, each one implemented with 750. Max Offset is incremented from 0 to 8, fixing Max Size in 4. We apply the same method to analyze the influence of Max Size in runtime performance, increasing its value from 1 to 8 and fixing Max Offset to 4.

Figure 11 shows both variables. We can see how Max Offset has a linear influence on execution time. The regression line shown in Figure 11 has a slope of 51, representing the cost in seconds of increasing one unit inMax Offset. For Max Size, the best regression obtained is quadratic (Figure 11). The user should be aware of that, meaning that choosing high values for Max Size may involve much greater increases of the execution times.

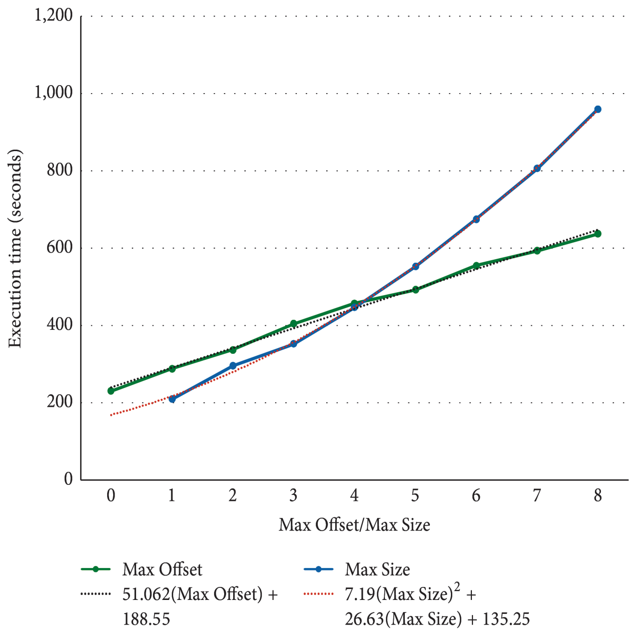


Figure 11: Execution time for an increasing number of Max Offset and Max Sizeparameters.

Increasing Types of Patterns

The last variable to be measured is the number of patterns to be recognized. The patterns are specified withPattern Detection functions provided by the user. In our decompiler example, we identified 3 patterns: return,call pre, and call post. We measure runtime performance of the 7 different combinations of these 3 patterns. Modules, workers, cores, Max Size, and Max Offset are fixed to 4, and each module contains 750 functions (3,000 in total).

Figure 12 shows the results. The 3 first bars show the execution time consumed to extract each pattern individually. The 3 next bars display the execution time for two patterns in parallel, compared to the costs of extracting them individually. We can see how the platform obtains an average benefit of 1.65 factors due to the parallelization. When the platform extracts 3 patterns at the same time, this benefit increases to 2.1 factors.

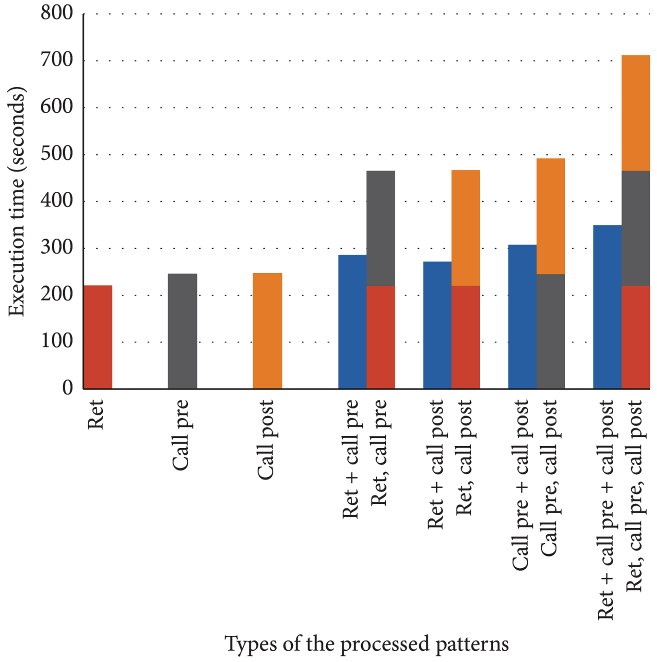


Figure 12: Execution time when extracting different types of patterns.

Execution Time for a Real Case Scenario

We have also measured execution time for the particular scenario of inferring the return type of a function. As mentioned, this is an existing problem of existing decompilers. The purpose of this section is not to present how this problem may be solved with machine learning, but to measure the execution time required to extract the binary patterns and to build the model.

To predict the type returned by a function, we extract binary code patterns before ret instructions and before and after function invocations. We found out that the number of functions required to build an accurate model for this problem is very high, so a huge program database would be needed. Instead, we implemented a code generation tool that writes synthetic C functions considering the language grammar and its type system. This way, we can generate any number of random functions (and invocations to them) for all the different types in the language (C built-in types plus type constructors for compound types (structs, unions, pointers, and arrays)). These functions are then passed to our platform to generate the output dataset. Then, the dataset is used to build a J48 classifier using Weka.

As mentioned, we can generate any number of C functions to be passed to our platform. Therefore, we must work out the number of functions necessary to build an accurate model. For this purpose, we used the following method: we create 1000 functions for each C type; we extract the binary patterns in that functions with our platform; and we use Weka with the generated dataset to compute the accuracy rate using 10-fold stratified cross validation. These steps are repeated in a loop, incrementing the number of functions in 1000 for each type. We stop when the Coefficient of Variation of the last 5 accuracy values is lower than 2%, representing that the increase of functions (individuals) does not represent a significant improvement of the accuracy. Finally, we build the J48 model with the dataset generated in the last iteration.

Following the method described above, we created a dataset with 160,000 functions and 3,321 binary patterns (the dataset file was 998 MB). The platform generated the dataset in 2 hours, 11 minutes, and 56 seconds (4 workers and CPUs). We also measured the sequential version, taking 7 hours 41 minutes and 46 seconds to generate the same dataset. For comparison purposes, we also evaluated the execution time to build a J48 model with the 160,000-function dataset, taking 11 hours, 55 minutes, and 15 seconds to build the model. Notice that Weka builds the model sequentially, not taking advantage of all the cores in the system.

RELATED WORK

There exist different works aimed at extracting assembly patterns from existing applications. To the knowledge of the authors, none of them have built a platform to extract those patterns automatically. They define custom processes and, some of them, even manual procedures.

Rosenblum et al. extract every combination of 1, 2, and 3 consecutive assembly instructions from a big set of executable files [6]. Then, they use forward feature selection to filter the most significant patterns and later train a Conditional Random Fields to detect the function entry points. The same authors use this methodology to detect the compiler used to generate the executables [7]. This research work was later extended to consider the compiler options and programming language used in the source application [8], to identify the programmer that coded the application [25], and to identify the functions belonging to the operating system [26].

BYTEWEIGHT provides another approach to find function entry points [5]. They apply machine learning to recognize the patterns, so that different compilers and optimization options may be used. Analyzing the training binaries, an extraction process generates prefix trees from sequences of bytes or normalized instructions. The prefix tree represents possible function start sequences. Then, they assign a weight representing the likelihood that the path from the root to the node is a function start in the training set. Finally, the weighted prefix tree is used to classify the input binary file.

Apart from assembly patterns extraction, there are situations where other parts of the binary files need to be processed. One example is the detection of packed executable files [27]. To this end, it is necessary to recognize not only assembly patterns, but also other types of information existing in the binaries, such as header patterns, entropy values, and characteristics of the file sections. Ugarte-Pedrero et al. propose a custom collective-learning-based process to solve this problem, detecting packed executables upon structural features, and heuristics [28].

Regarding decompilation, Cifuentes et al. identified the existing limitations on recognizing high-level control structures [29]. They later define a technique to recover jump tables and their target addresses and incorporated it in the DCC decompiler [30]. The Phoenix decompiler uses a structuring algorithm to detect control flow structures, being able to decompile more structures than Hex-Rays [13]. Regarding decompilation of high-level types, Mycroft proposes a constraint-based algorithm to infer types from binary code [31]. Another type recovery approach is the VSA algorithm based on value propagation [32]. Laika is a system that uses Bayesian unsupervised learning to detect high-level data structures, analyzing the process memory images [33].

CONCLUSIONS

We propose a platform for the automatic extraction of patterns in binary files, capable of analyzing big executable files. The platform is highly parameterized to be used in different scenarios. The extracted patterns can be used to predict features in native code, when the high-level source code and the debug information are not available.

The platform implementation has been parallelized to increase its runtime performance on multicore architectures. Both data and task parallelization schemes have been followed. We have evaluated its performance, obtaining a performance benefit of 3.5 factors over the maximum theoretical value of 4 factors. The evaluation presented also documents how the different parameters of the platform should be used to obtain the best performance.

We are currently using the proposed platform to extract patterns that are later used to improve the information inferred by existing decompilers. We generate patterns of high-level type information to train a classifier using different machine learning algorithms. We are currently focused on the return types of functions, but we hope to apply it to parameters and local and global variables.

We plan to use clustering algorithms to the dataset generated for a big battery of programs taken from open source code repositories. The objective is to obtain classifications of (sections of) applications depending on the patterns found inside them. The classes obtained may be helpful to identify the code that performs common input/output, network, and computing intensive or multithreaded operations.

The platform implementation, its source code, the 3 different configurations used in this article (return type, function or procedure identification, and FEPs extraction in binary files), and all the examples used in the evaluation are available for download at http://www.reflection.uniovi.es/decompilation/download/2016/sp/.

ACKNOWLEDGMENTS

This work has been funded by the European Union, through the European Regional Development Funds (ERDF), and the Principality of Asturias, through its Science, Technology and Innovation Plan (Grant GRUPIN14-100). The authors have also received funds from the Banco Santander through its support to the Campus of International Excellence.

REFERENCES

1. Defense Advanced Research Projects Agency, MUSE envisions mining “big code” to improve software reliability and construction, 2014, http://www.darpa.mil/news-events/2014-03-06a.
2. F. Ortin, J. Escalada, and O. Rodriguez-Prieto, “Big code: new opportunities for improving software construction,” Journal of Software, vol. 11, no. 11, pp. 1083–1008, 2016.
3. F. Yamaguchi, M. Lottmann, and K. Rieck, “Generalized vulnerability extrapolation using abstract syntax trees,” in Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC ‘12), pp. 359–368, ACM, Los Angeles, Calif, USA, December 2012.
4. E. Alpaydin, Introduction to Machine Learning, The MIT Press, 2nd edition, 2010.
5. T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, “Byteweight: learning to recognize functions in binary code,” in Proceedings of the 23rd USENIX Conference on Security Symposium (SEC ‘14), pp. 845–860, USENIX Association, San Diego, Calif, USA, August 2014.
6. N. Rosenblum, X. Zhu, B. Miller, and K. Hunt, “Learning to analyze binary computer code,” in Proceedings of the 23rd National Conference on Artificial Intelligence—Volume 2 (AAAI ‘08), pp. 798–804, AAAI Press, 2008.
7. N. E. Rosenblum, B. P. Miller, and X. Zhu, “Extracting compiler provenance from program binaries,” in Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE ‘10), pp. 21–28, ACM, Toronto, Canada, June 2010.
8. N. Rosenblum, B. P. Miller, and X. Zhu, “Recovering the toolchain provenance of binary code,” in Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA ‘11), pp. 100–110, ACM, Ontario, Canada, July 2011.
9. I. Santos, Y. K. Penya, J. Devesa, and P. G. Bringas, “N-grams-based file signatures for malware detection,” in Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS ‘09), pp. 317–320, AIDSS, 2009.
10. C. Liangboonprakong and O. Sornil, “Classification of malware families based on N-grams sequential pattern features,” in Proceedings of the 8th IEEE Conference on Industrial Electronics and Applications

(ICIEA ‘13), pp. 777–782, June 2013.

11. V. Raychev, M. Vechev, and A. Krause, “Predicting program properties from ‘big code’,” in Proceedings of the 42nd Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL ‘15), pp. 111–124, 2015.
12. K. Troshina, A. Chernov, and Y. Derevenets, “C decompilation: is it possible?” in Proceedings of the International Workshop on Program Understanding (PSI ‘09), pp. 18–27, Altai Mountains, Russia, 2009.
13. E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, “Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring,” in Proceedings of the 22nd USENIX Security Symposium, USENIX, pp. 353–368, Washington, DC, USA, 2013.
14. A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina, “SmartDec: approaching C^{++} decompilation,” in Proceedings of the 18th Working Conference on Reverse Engineering (WCRE ‘11), pp. 347–356, IEEE, October 2011.
15. Y. Fan, Y. Ye, and L. Chen, “Malicious sequential pattern mining for automatic malware detection,” Expert Systems with Applications, vol. 52, pp. 16–25, 2016.
16. J. D. Lafferty, A. McCallum, and F. C. N. Pereira, “Conditional random fields: probabilistic models for segmenting and labeling sequence data,” in Proceedings of the 18th International Conference on Machine Learning (ICML ‘01), pp. 282–289, Morgan Kaufmann, 2001.
17. J. Escalada and F. Ortin, Source code for the article: An efficient platform for the automatic extraction of patterns in native code, 2016, http://www.reflection.uniovi.es/decompilation/download/2016/sp.
18. LLVM, clang: a C language family frontend for LLVM, 2016, http://clang.llvm.org.
19. E. Bachaalany, GitHub: IDAPython, 2016, https://github.com/idapython.
20. D. Beazley, “Understanding the python GIL,” in Proceedings of the PyCON Python Conference, Atlanta, Ga, USA, February 2010.
21. D. Phillips, Python 3 Object-Oriented Programming, Packt Publishing Ltd, Livery Place, Birmingham, UK, 2nd edition, 2015.
22. J. M. Redondo, F. Ortin, and J. M. C. Lovelle, “Optimizing reflective primitives of dynamic languages,” International Journal of Software

Engineering and Knowledge Engineering, vol. 18, no. 6, pp. 759–783, 2008.

23. F. Ortin, L. Vinuesa, and J. M. Felix, “The DSAW aspect-oriented software development platform,” International Journal of Software Engineering and Knowledge Engineering, vol. 21, no. 7, pp. 891–929, 2011.
24. G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in Proceedings of the Spring Joint Computer Conference, pp. 483–485, Atlantic City, NJ, USA, April 1967.
25. N. Rosenblum, X. Zhu, and B. P. Miller, “Who wrote this code? Identifying the authors of program binaries,” in Computer Security—ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12–14,2011. Proceedings, vol. 6879 of Lecture Notes in Computer Science, pp. 172–189, Springer, Berlin, Germany, 2011.
26. E. R. Jacobson, N. Rosenblum, and B. P. Miller, “Labeling library functions in stripped binaries,” in Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE ‘11), pp. 1–8, ACM, Szeged, Hungary, September 2011.
27. I. Santos, X. Ugarte-Pedrero, B. Sanz, C. Laorden, and P. G. Bringas, “Collective classification for packed executable identification,” in Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference (CEAS ‘11), pp. 23–30, Perth, Australia, September 2011.
28. X. Ugarte-Pedrero, I. Santos, and P. G. Bringas, “Structural feature based anomaly detection for packed executable identification,” in Computational Intelligence in Security for Information Systems: 4th International Conference, CISIS 2011, Held at IWANN 2011, Torremolinos-Málaga, Spain, June 8–10, 2011. Proceedings, vol. 6694 of Lecture Notes in Computer Science, pp. 230–237, Springer, Berlin, Germany, 2011.
29. C. Cifuentes, D. Simon, and A. Fraboulet, “Assembly to high-level language translation,” in Proceedings of the IEEE International Conference on Software Maintenance (ICSM ‘98), pp. 228–237, IEEE, Bethesda, Md, USA, November 1998.

30. C. Cifuentes and M. Van Emmerik, “Recovery of jump table case statements from binary code,” Science of Computer Programming, vol. 40, no. 2-3, pp. 171–188, 2001.

31. A. Mycroft, “Type-based decompilation,” in Proceedings of the European Symposium on Programming (ESOP ‘99), pp. 208–223, 1999.

32. G. Balakrishnan and T. Reps, “Divine: discovering variables in executables,” in Verification, Model Checking, and Abstract Interpretation: 8th International Conference, VMCAI 2007, Nice, France, January 14–16, 2007. Proceedings, vol. 4349 of Lecture Notes in Computer Science, pp. 1–28, Springer, Berlin, Germany, 2007.

33. A. Cozzie, F. Stratton, H. Xue, and S. T. King, “Digging for data structures,” in Proceedings of the 8th Conference on Operating Systems Design and Implementation (OSDI ‘08), pp. 255–266, San Diego, Calif, USA, December 2008.

CHAPTER 14

POLYGLOT PROGRAMMING IN APPLICATIONS USED FOR GENETIC DATA ANALYSIS

Robert M. Nowak

Institute of Electronic Systems, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland

ABSTRACT

Applications used for the analysis of genetic data process large volumes of data with complex algorithms. High performance, flexibility, and a user interface with a web browser are required by these solutions, which can be achieved by using multiple programming languages. In this study, I developed a freely available framework for building software to analyze genetic data, which uses C++, Python, JavaScript, and several libraries. This system was used to build a number of genetic data processing applications and it reduced the time and costs of development.

Citation: Robert M. Nowak, "Polyglot Programming in Applications Used for Genetic Data Analysis," BioMed Research International, vol. 2014, Article ID 253013, 7 pages, 2014. doi:10.1155/2014/253013.

BACKGROUND

The number of computer programs for the analysis of genetic data is increasing significantly, but it still needs to be improved greatly because of the importance of result analysis with appropriate methods and the exponential growth in the volume of genetic data.

Genetic data are typically represented by a set of strings [1], where each string is a sequence of symbols from a given alphabet. The string representation, called primary structure, reflects the fact that the molecules storing genetic information (DNA and RNA) are biopolymers of nucleotides, while proteins are polypeptide chains. The secondary, tertiary, and quaternary structures need to be considered to understand the interactions among nucleotides or amino acids, but they are used less frequently in computer programs. The secondary structure includes the hydrogen bonds between nucleotides in DNA and RNA and the hydrogen bonds between peptide groups in proteins, where the molecules are represented by graphs. The tertiary structure refers to the positions of atoms in three-dimensional space, and the quaternary structure represents the higher level of organization of molecules. The representations of molecules are extended based on connections between sequences or subsequences, which denotes similarity from various perspectives. Moreover, these data are supplemented with human-readable descriptions, which facilitate an understanding of the biological meanings of the sequence, that is, its function and/or its structure.

The large number of possible candidate solutions during the analysis of genetic data means that the employed algorithms must be selected carefully [2]. Exhaustive search algorithms must be supported by heuristics based on biological properties of the modeled objects. Of particular importance in this field are dynamic programming algorithms, which allow us to find the optimal alignment of biological sequences (i.e., arranging the sequences by inserting gaps to identify regions of similarity [1]) in polynomial time, although the search space grows exponentially. Dynamic programming is used to search for similarity (local or global), to generate a multisequence representation (profile), and to examine sequences with hidden Markov models. In addition, backtracking algorithms are used to search for motifs (i.e., identifying meaningful patterns in genetic sequences), greedy algorithms to detect genome rearrangements and to sort by reversals, divide-and-conquer algorithms to perform space-efficient sequence alignments, and graph algorithms for DNA assembly.

A characteristic feature of the computer programs applied to genetic data is the necessity to analyze large amounts of data using complex algorithms, which means that high performance is crucial. Different user and system requirements mean that the flexibility of software is also important. Finally, users prefer a graphical interface that is accessible from a web browser and applications that update automatically.

Scientists are becoming increasingly involved in software development [3]. They should use software engineering practices and tools to avoid common mistakes and to speed up the development tasks [4]. The architecture of working application with explanation of development decisions could help in developing new computer programs. Biological and medical terminology is simplified to invite developers to discuss the presented solutions.

In this study, I describe the bioweb framework, including application architecture, the programming languages, libraries, and tools, used to develop applications for processing genetic data. I propose a multilanguage platform using C++, Python, and JavaScript. The use of appropriate and tested architectures, libraries, and tools decreases the risk of failure in software system development as well as reduces the costs and time requirements. The use of appropriate systems also facilitates rapid prototyping, which allows us to verify concepts by obtaining the requisite information from end users: biologists and doctors.

RESULTS

Deployment Model

A three-layer software architecture was selected where the presentation layer, data processing layer, and data storage layer were kept separate. The use of a multilayered model makes computer programs flexible and reusable, because applications have different responsibilities. Thus, it is beneficial to segregate models into layers that communicate via well-defined interfaces. Layers help to separate different subsystems, and the code is easier to maintain, clean, and well structured.

Four possible deployment models were considered for the three-layer architecture: the desktop, the database server, the thin client, and the web application, as shown in Figure 1. The desktop architecture (Figure 1(a)) was rejected because the framework was designed to support multiuser applications. Collaboration features were hard to implement in this architecture because of the lack of central data server that could be accessed by multiple users. The offline mode is rarely used because the Internet is available almost everywhere and the transmission costs are negligible compared with the costs of maintaining the system. Furthermore, sequence databases are publicly available via the Internet, so an Internet connection is essential for the analysis of genetic data.

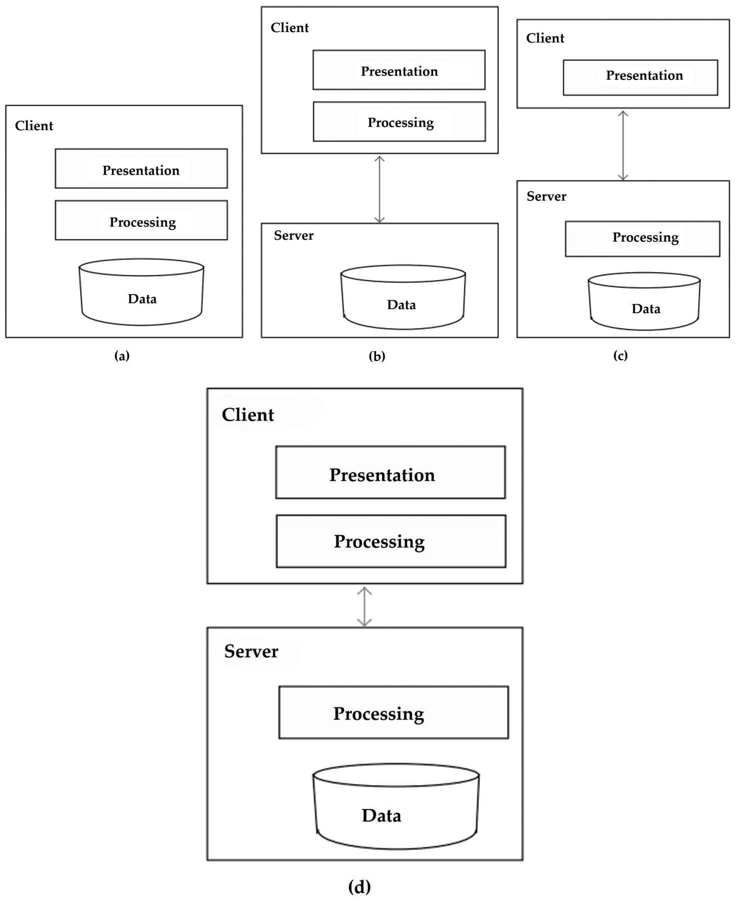


Figure 1: Three-layer application deployment models: desktop application (a), database server (b), thin client (c), and web application (d). This solution supports the creation of applications using a web application architecture.

An application architecture with a shared database and data processing modules deployed on client machine (Figure 1(b)) was rejected because of the requirement for high client computer performance. Another problem is the need to update the software on the client side when changes and additions are made, which is time consuming and requires support for a wide range of platforms so the development costs are high.

Deploying the calculation modules on a server machine allows the execution of these modules by clients on different platforms, which reduces the development costs. The computational power of the server is important because it determines the computational time, which means that poorly equipped client machines can be used. The optimum solutions are a thin client architecture, as shown in Figure 1(c), and a web application architecture, as shown in Figure 1(d).

Deploying the calculation modules on a server machine, as shown in Figures 1(c) **and** 1(d), allows the use of many platforms on the client side, which reduces the development costs. Importantly, the computational power of the server is used, so the computational time can be relatively short, even for poorly equipped client machines. These solutions simplify scalability if the size of the problem or the number of clients grows, because only the servers need to be upgraded.

Web applications have advantages compared with application produced with a thin client architecture because the client contains a portion of the data processing layer, which can handle activities such as output reformatting, graph generation, and user input validation. Client-based processing reduces the amount and frequency of client-server traffic, and it reduces the load on the server while the reactions to user actions are faster. This solution uses web browser plugins (such as Flash) or HTML5/JavaScript programs on the client side. The client modules are downloaded during initialization, which helps to avoid the issue of updating the software.

Architecture and Programming Languages

The software used by presented framework and the framework itself were created with C++, Python, and JavaScript with HTML5. The use of multiple languages in a single project is quite common and it is an alternative to using PHP, NET, or Java. The set of used languages facilitates high performance, versatility, customizable modules, and the production of a web browser interface. The modules produced for a typical application based on bioweb **using these programming languages are shown in Figure** 2.

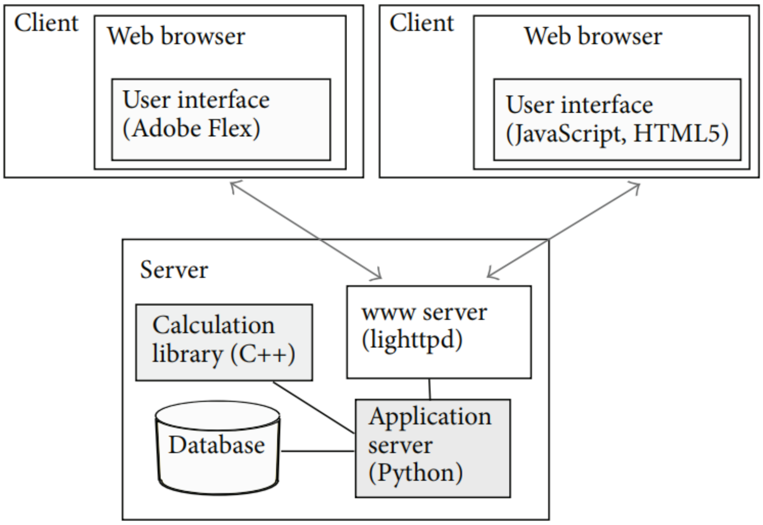

Client
Web browser
User interface (Adobe Flex)
Client
Web browser
User interface (JavaScript, HTML5)
Server
Calculation library (C++)
www server (lighttpd)
Database
Application server (Python)

Figure 2: Modules produced for a typical application based on the proposed framework using various programming languages.

The algorithms are implemented in C++. The source code is translated (compiled) into machine language, which makes algorithm execution more efficient because the code is executed directly by the processor. The language has higher-level abstractions missing in other languages translated into binary code (C and Fortran). C++ supports object-oriented programming by providing virtual functions and multibase inheritance and exceptions and facilitates functional and genetic programming, including templates and lambda functions. The standard C++ library is compact but it is well tested and efficient. It includes support for inputs and outputs, strings and string operations such as regular expressions, and sets of collections, such as vectors, lists, sets, and associative arrays using trees and/or hash tables. It should be mentioned that concurrency support mechanisms are included in the C++11 standard (ISO/IEC 14882:2011), so the full capabilities of modern computers with multiple processors and/or multiple cores can be exploited. If an older C++ compiler that does not support C++11 is used, it may be necessary to employ the Boost [5] libraries: Boost.Thread to create portable multithread applications, Boost.Regex for regular expressions, and Boost.Chrono for time utilities. In addition, vector calculations provided by modern graphics processing unit (GPU) are available in C++ and the OpenCL [6] standard is applied.

The server application uses the Python language in presented solution, mainly because this type of development is faster compared with C++. Modules that do not constitute a bottleneck during calculations should be implemented in Python. Python is a scripting language, so it is small and has a simple, regular syntax. This language is dynamically type-checked, uses a uniform data model, and provides reference counting memory management, so there is no problem with memory leaks. The Python repository of software (PIP) https://pypi.python.org/pypi contains over 30,000 packages and a number of ready-made solutions can be used, particularly the packages for exchanging data and packages that support the creation of the web applications I used. It should be noted that the Biopython library [7] provides a set of tools for biological computation which are written in Python.

In bioweb **the** Boost.Python [5] library enables interoperability between the C++ modules and the Python modules. Other solutions, such as using C API from Python directly, code generation using Simplified Wrapper and Interface Generator (SWIG), Py++, Pyrex, and cython, were considered to be less useful because the interface was less convenient and there was a lack of support for the techniques used in genetic data software development. The Boost.Python uses C Python API and metaprogramming techniques, which allows the exposure of C++ classes, functions, and objects to Python and vice-versa, thereby supporting the use of Python facilities inside C++ code. Boost.Python allows the exposure of elements and the register of conversions using a simple syntax and being easy to learn.

The use of a compiler and an interpreter makes the developed software more flexible. The application customization requires the use of an interpreter in any case, because changing the settings should not demand the software rebuilding. The use of Python to store the user settings simplifies the customization of applications greatly, because the settings do not need to be lists of names and values, and the Python control instructions can be used.

A client application request is sent to the standard port using the HTTP protocol and it is retransmitted by the web server using interprocess communication mechanisms (e.g., sockets and named pipes) to the server application. Three web servers were investigated: Apache http://httpd.apache.org, Lighttpd http://www.lighttpd.net, and Nginx http://nginx.org. The Lighttpd configuration is known to be simple and its performance is very good, so the presented solution only includes settings for this web server, but bioweb **is also able to use Apache and** Nginx. So scripts available on project website only include a setting for this web server. Lighttpd retransmission uses mod_fastcgi and a socket mechanism. Three open source Python libraries were considered: Flup from PIP, Web2py http://www.web2py.com, and Django https://www.djangoproject.com. The libraries support the Web Server Gateway Interface (WSGI), the Python standard interface between web servers and applications.

Flup is a simple WSGI server but its library is small (256 kB), so the facilities are limited to the python function call when an http request is received from a client and the function results are sent back to the client application using a web server. More advanced libraries are Web2Py (9 MB) and Django (22 MB), where the facilities include parameter conversion, authentication, authorization, and database support using object-relational mapping. All Flup, Web2py, and Django were tested in the present study, because the characteristics of Web2py and Django are similar. However, Django is recommended because all of the available facilities are written explicitly and this library has the best documentation. Django uses Flup internally to cooperate with Lighttpd in current version of software; this configuration works correctly under all popular modern operating systems (Linux, Windows, iOS, etc.).

Bioweb provides two competitive solutions for client modules, where the first is based on JavaScript with HTML5, and the second uses Apache Flex and the Adobe Flash Player plugin. JavaScript with HTML5 web applications uses the Ajax techniques available on modern web browsers, mainly XMLHttpRequest objects, so client applications developed in JavaScript can send and retrieve data in the background. The data are interchanged using the JavaScript Object Notation (JSON), and the Python standard library module supports JSON encoders and decoders. The HTML5 standard includes scalable vector graphics support, which improves the graphical user interface. JavaScript is interpreted by a web browser and it conforms to international standard ISO/IEC 16262:2011. The current version uses Model View ViewModel (MVVM) client-side JavaScript framework AngularJS [8].

Apache Flex is a freely available set of software development tools, which support the construction of applications that use the Adobe Flash Player plugin. This plugin, which is available for most web browsers, allows the user to view multimedia, vector graphics, and animations. The Apache Flex application is loaded from web server and executed on the client side. Communication with the server uses the Action Message Format

(AMF), which is supported in Python by the pyAMF library. At present, this technology is being replaced by HTML5, which is supported directly by web browsers, so HTML5 and JavaScript are recommended for use in new applications.

Parallel Service Requests

The framework was designed to create the software that serves multiple users at the same time. The users communicate independently with the server via the Internet and the framework includes a component with the active object pattern [9] implementation to enhance concurrency and to exploit the server resources fully. This component, which is part of bioweb, is shown in Figure 3.

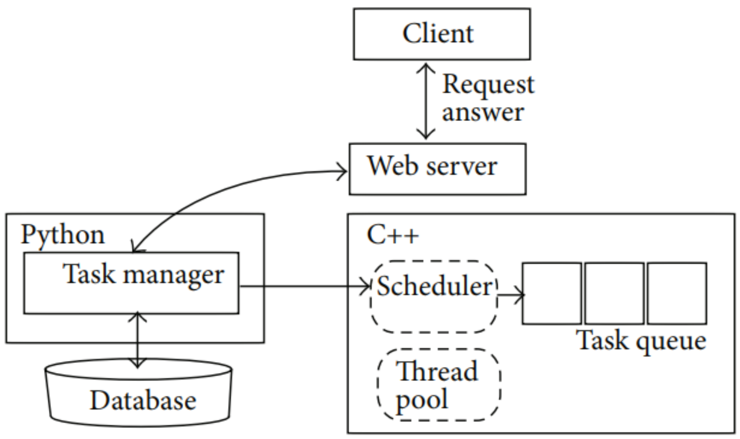


Figure 3: Active object implementation delivered by the framework. The client requests are transformed into commands automatically, which are executed by separate threads.

The execution of calculation tasks is decoupled from task invocation to enhance concurrency and to simplify multithread usage, as shown in Figure 4. Calculation requests sent from the client application are converted into C++ objects. These objects are commands (the command design pattern is used) which contain specific parameters as well as algorithm and synchronization mechanisms. Commands are stored in the task queue and executed by separate execution threads from the thread pool. The command handlers are accessible from Python, so the user can examine the current command state, that is, tasks that are awaiting execution in the queue, executed tasks, and completed tasks. This component uses an observer (from observer design pattern), to support the command progress notification. The active object module can be used independently of bioweb; it is supplied separately as a C++ library, whose sources are available at http://mt4cpp.sourceforge.net.

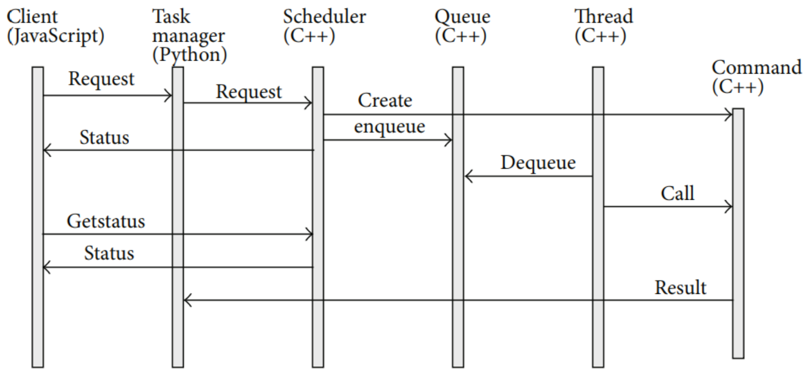


Figure 4: Cooperation among active object participants. The client request is converted into a command managed by the task manager on the Python side and by the scheduler in C++. The command is stored in the queue, and it is executed when an unoccupied thread is available. The client can request the current command status and the command progress.

Testing

Software testing is an integral part of the development process. Thus, testing techniques and libraries that support this process are specified in presented framework. Three types of tests are considered: unit tests, integration tests, and system tests. Unit testing checks individual functions, procedures, and classes in isolation. Integration tests examine the communication between modules, based on a consideration that they are created in different programming languages. System tests examine the functions of a computer program as a whole, without the knowledge of the internal structure of the software.

Unit testing uses Boost.Test [5] for C++ modules, the standard Python unittest package for Python code, and QUnit http://qunitjs.com for modules written in JavaScript. C++ unit testing is performed in both environments: g++ and msvc. Integration tests are implemented with the same tools and libraries as unit tests, but the features of C++ modules exported to Python by the Boost.Python library are tested in Python using unittest.

System testing uses the Python language and splinter http://splinter.cobrateam.info library. This tool automates browser actions such as visiting URLs, navigation, verifying page context, finding elements in the page, testing mouse and keyboard events, reading the text properties of elements, and other tasks. The system tests allow the automatic evaluation of test scenarios, without any requirement for manual testers, which reduces the time and the cost of the overall system examination.

The test quality measure is the source code coverage during unit, integration, and system testing. This measure provides numerical data related to the performance of test procedures, which helps to identify inadequately tested parts of the software. The analytic tools used to evaluate coverage in bioweb **are** gcov from the GNU Compiler Collection for C++ modules, Coverage.py from Python Package Index (PIP) for Python modules, and Blanket.js http://blanketjs.org/ **for JavaScript code.**

Tools

This section describes the programming tools used to create applications in bioweb. It is important that the latest versions of the tools described are used.

The C++ modules require a C++ compiler and it is recommended to use at least two different compilers, particularly the g++ compiler from the GNU Compiler Collection http://gcc.gnu.org and the Microsoft Visual C++ Compiler (msvc) http://msdn.microsoft.com. The use of different compilers increases the probability of capturing errors in the code and it ensures that the code is portable. The C++ modules use the standard C++ library and the Boost http://www.boost.org libraries. The server uses the Python interpreter, the Python standard library, and packages from the Python repository (PIP). The client uses the JavaScript interpreter built-in web and the AngularJS [8] framework, jQuery libraries http://jquery.com. The Bower [10] automatically manages client-side package dependencies. An alternative is to use the Apache Flex software developer's kit http://flex.apache.org. The Scons http://www.scons.org is used to create modules, for testing, and to consolidate the whole system, while Redmine http://www.redmine.org is used for project management, and mercurial http://mercurial.selenic.com is used as the version control system.

DISCUSSION

To speed up the creation of new software, the developer can use a specialized framework. The most popular, freely available frameworks are Bioconductor [11], MEGA tool [12], and OpenMS [13]. On the other hand, the programmer can use general-purpose programming language and specialized libraries, for example, C++ with NCBI C++ Toolkit [14], Python with BioPython [7], Java with BioJava [15], and BioWeka [16]. All these solutions impose limitations connected with the usage of only one programming language [17] and do not support the user interface in a web browser.

The polyglot environment is common among web software, that is, software accessible from a web browser, because the client-side software (JavaScript, HTML, and CSS) has different responsibilities compared to server side. The ubiquity of mobile applications and the advent of big data change the software development to use multiple languages [18]. Similar trends are evident in the bioinformatics software and the examples are GBrowse [19] or GEMBASSY [20]. Bioweb provides a framework for the construction of such applications. There are many application development frameworks that connect C++ with Python or Python with JavaScript. Presented solution is similar but combines three programming languages.

The bioweb is small, but it can be extended, and it can use specialized libraries. The heavyweight web-based genome analysis frameworks, such as Galaxy [21], have a lot of ready-made modules and meet most of the requirements for systems for the genetic data analysis. However, creating custom modules and algorithms is not trivial. Presented framework allows the user to create smaller and independent solutions, which are easier to manage and to customize. It could be easily extended to use GPU and/or computing clusters, which is required in production-scale analysis.

CONCLUSION

The bioweb framework is freely available from http://bioweb.sourceforge.net under GNU Library or Lesser General Public License version 3.0 (LGPLv3). All of the libraries and applications used in bioweb are available for free and they can be used in commercial software.

This framework was used to create several applications to analyze genetic data: DNASynth application for synthesizing artificial genes (i.e., completely synthetic double-stranded DNA molecules coding peptide), theDNAMarkers application for analyzing DNA mixtures, the CodonHmm application for protein back-translation, the WebOmicsViewer application for storing and analyzing genomes, the PETconn application to create scaffolds using paired-end tags, and the DNAAssembler for assembling DNA using next-generation sequencing data. The source code for these applications is available on the project website. This genetic data analysis software development project was performed in academia and it supports students who have a limited amount of time available and who also lack experience in design and programming. I found that agile methodologies [22] worked well in this project because they support the transfer of biological and medical knowledge from the users of the application. They let us avoid the duplication of information and allowed minimal documentation production, so a task could be completed relatively quickly by new users. In particular, the SCRUM [23] and the extreme programming (XP) [24] techniques were used, that is, SCRUM roles (product owner, development team, and scrum master), SCRUM iterations (sprint planning meeting, end meetings), SCRUM task management and prioritizing, XP test-driven development, and XP coding and documentation.

Presented framework is still being developed; the Guncorn [25] Python HTTP Server is added to the upcoming version. This cancels the Flup on Unix platforms and accelerates data transfer between client and server.

Availability and Requirements(i)project homepage: http://bioweb.sourceforge.net;(ii)operating systems(s): OS Portable;(iii)programming language: C++ and Python and JavaScript;(iv)license: GNU Library or Lesser General Public License version 3.0 (LGPLv3);(v)getting started: to build a "Hello World" application please download the latest version, extract the files from the archive, install additional software as described in README_EN (text file in main bioweb directory), and run scons command in the directory where you placed the bioweb. To start the client and server locally run scons r=1.

ACKNOWLEDGMENTS

This work was supported by the statutory research of Institute of Electronic Systems of Warsaw University of Technology. The author would like to thank the editor and anonymous reviewer for their constructive comments. The author is grateful to the students of the Faculty of Electronics and Information Technology of Warsaw University of Technology, who acted as the early users of this software, and to Hanna Markiewicz for the proofreading.

REFERENCES

1. R. Durbin, Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids, Cambridge University Press, Cambridge, UK, 1998.
2. N. C. Jones and P. Pevzner, An Introduction to Bioinformatics Algorithms, The MIT press, Cambridge, Mass, USA, 2004.
3. J. M. Osborne, M. O. Bernabeu, M. Bruna et al., "Ten simple rules for effective computational research," PLoS Biology, vol. 10, no. 3, Article ID e1003506, 2014.
4. G. Wilson, D. Aruliah, C. T. Brown et al., "Best practices for scientific computing," PLoS Biology, vol. 12, no. 1, Article ID e1001745, 2014.
5. R. Nowak and A. Pajak, C++ Language: mechanisms, design patterns, libraries, BTC, Le gionowo, Poland, 2010.
6. J. E. Stone, D. Gohara, and G. Shi, "OpenCL: a parallel programming standard for heterogeneous computing systems," Computing in Science and Engineering, vol. 12, no. 3, Article ID 5457293, pp. 66–72, 2010.
7. P. J. A. Cock, T. Antao, J. T. Chang et al., "Biopython: freely available python tools for computational molecular biology and bioinformatics," Bioinformatics, vol. 25, no. 11, pp. 1422–1423, 2009.
8. P. B. Darwin and P. Kozlowski, AngularJS Web Application Development, Packt Publishing, Birmingham, UK, 2013.
9. R. G. Lavender and D. C. Schmidt, "Active object—an object behavioral pattern for concurrent programming".
10. Bower, a package manager for the web, http://bower.io/.
11. R. C. Gentleman, V. J. Carey, D. M. Bates et al., "Bioconductor: open software development for computational biology and bioinformatics," Genome Biology, vol. 5, no. 10, p. R80, 2004.
12. S. Kumar, K. Tamura, and M. Nei, "MEGA3: integrated software for molecular evolutionary genetics analysis and sequence alignment," Briefings in Bioinformatics, vol. 5, no. 2, pp. 150–163, 2004.
13. M. Sturm, A. Bertsch, C. Gröpl et al., "OpenMS—an open-source software framework for mass spectrometry," BMC Bioinformatics, vol. 9, no. 1, article 163, 2008.

14. D. Vakatov, The NCBI C++ toolkit book, 2004.
15. R. C. G. Holland, T. A. Down, M. Pocock et al., "BioJava: an open-source framework for bioinformatics," Bioinformatics, vol. 24, no. 18, pp. 2096–2097, 2008.
16. J. E. Gewehr, M. Szugat, and R. Zimmer, "BioWeka: extending the Weka framework for bioinformatics," Bioinformatics, vol. 23, no. 5, pp. 651–653, 2007.
17. M. Fourment and M. R. Gillings, "A comparison of common programming languages used in bioinformatics," BMC Bioinformatics, vol. 9, article 82, 2008.
18. A. Binstock, The quiet revolution in programming. Dr. Dobb's , 2013, http://www.drdobbs.com/architecture-and-design/the-quiet-revolution-in-programming/240152206.
19. M. Wilkinson, "Gbrowse Moby: a Web-based browser for BioMoby services," Source Code for Biology and Medicine, vol. 1, no. 1, article 4, 2006.
20. H. Itaya, K. Oshita, K. Arakawa, and M. Tomita, "GEMBASSY: an EMBOSS associated software package for comprehensive genome analyses," Source Code for Biology and Medicine, vol. 8, article 17, 2013.
21. J. Goecks, A. Nekrutenko, J. Taylor et al., "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," Genome Biology, vol. 11, no. 8, article R86, 2010.
22. M. Fowler and J. Highsmith, "The agile manifesto," Software Development, vol. 9, no. 8, pp. 28–35, 2001.
23. K. Schwaber, Agile Project Management with SCRUM, O'Reilly Media, Sebastopol, Calif, USA, 2004.
24. K. Beck and C. Andres, Extreme Programming Explained: Embrace Change, Addison-Wesley Professional, Boston, Mass, USA, 2004.
25. Gunicorn, Python WSGI HTTP server for UNIX, http://gunicorn.org/.

CHAPTER
15

CLASSIFYING MULTIGRAPH MODELS OF SECONDARY RNA STRUCTURE USING GRAPH-THEORETIC DESCRIPTORS

Debra Knisley[1,2] , Jeff Knisley[1,2] , Chelsea Ross[2] , and Alissa Rockney[2]

[1]Institute for Quantitative Biology, East Tennessee State University, Johnson City, TN 37614-0663, USA

[2]Department of Mathematics and Statistics, East Tennessee State University, Johnson City, TN 37614-0663, USA

ABSTRACT

The prediction of secondary RNA folds from primary sequences continues to be an important area of research given the significance of RNA molecules in biological processes such as gene regulation. To facilitate this effort,

Citation: Debra Knisley, Jeff Knisley, Chelsea Ross, and Alissa Rockney, "Classifying Multigraph Models of Secondary RNA Structure Using Graph-Theoretic Descriptors," ISRN Bioinformatics, vol. 2012, Article ID 157135, 11 pages, 2012. doi:10.5402/2012/157135.

graph models of secondary structure have been developed to quantify and thereby characterize the topological properties of the secondary folds. In this work we utilize a multigraph representation of a secondary RNA structure to examine the ability of the existing graph-theoretic descriptors to classify all possible topologies as either RNA-like or not RNA-like. We use more than one hundred descriptors and several different machine learning approaches, including nearest neighbor algorithms, one-class classifiers, and several clustering techniques. We predict that many more topologies will be identified as those representing RNA secondary structures than currently predicted in the RAG (RNA-As-Graphs) database. The results also suggest which descriptors and which algorithms are more informative in classifying and exploring secondary RNA structures.

INTRODUCTION

The need for a more complete understanding of the structural characteristics of RNA is evidenced by the increasing awareness of the significance of RNA molecules in biological processes such as their role in gene regulatory networks which guide the overall expressions of genes. Consequently, the number of studies investigating the structure and function of RNA molecules continues to rise and the characterization of the structural properties of RNA remains a tremendous challenge in computational biology. RNA molecules are seemingly more sensitive to their environment and have greater degrees of backbone torsional freedom than proteins, resulting in even greater structural diversity [1]. Although the tertiary structure is of significant importance, it is much more difficult to predict than the tertiary structure of proteins. Advances in molecular modeling have resulted in accurate predictions of small RNAs. However, the structure prediction for large RNAs with complex topologies is beyond the reach of the current ab initio methods [2].

A coarse-grained model to refine tertiary RNA structure prediction was developed by Ding et al. [2] to produce useful candidate structures by integrating biochemical footprinting data with molecular dynamics. Although the focus is on tertiary folds, their method uses information about RNA base pairings from known secondary structures as a starting point. This, coupled with the understanding that the RNA folding mechanisms producing tertiary structure are believed to be hierarchical in nature, implies that much can be achieved by discovering all possible secondary structural RNA topologies.

Given the primary sequence of an RNA molecule, there are a number of algorithms and tools available to predict the most likely set of resulting secondary structures. The most widely used algorithms such as Zucker's Mfold [3] and Vienna RNAfold [4] typically base their predictions on the minimum free energy paradigm. While these algorithms have been highly beneficial, it is not always the case that the predicted structure with minimum free energy is the correct one and consequently some suggest that the actual RNA secondary structure may not have a minimum global free energy, only local ones [5]. Other means of characterizing the topology of secondary RNA structures are still an active avenue of pursuit.

The graph representations used in this work can be found in the database RAG: RNA-As-Graphs [6]. Secondary RNA structure is modeled by two graph-theoretic representations in the database resource RAG (see [6] for additional details on the differences between the two). In one of these representations, regions of the secondary structure that consist of unpaired bases such as junctions, hairpins, and bulges are represented by vertices. The connecting stems are represented quite naturally as connecting edges. The resulting graph is a connected, acyclic graph, that is, a tree. One advantage of this representation is the fact that trees have been highly studied in the graph theory thereby providing a wealth of information about the model. For instance it is known by the generating function developed by Harary and Prins [7] exactly how many distinct trees can be constructed for a given number of vertices. This allows the entire space, that is, all possible configurations, to be considered. Unfortunately, secondary RNA structures containing a pseudoknot cannot be represented as described above by the tree model. If, however, the model is reversed and stem regions are represented as vertices and connecting strings of unpaired bases as the edges, all secondary RNA structures can be now be modeled, including those that contain a pseudoknot. This representation is called the dual graph in the RAG database. The resulting dual graph however is no longer a simple graph; instead this method produces a multigraph. Unlike a simple graph, a multigraph can have more than one edge connecting a pair of vertices. And, unlike simple graphs, multigraphs have not been as highly studied in the theoretical setting. In previous work [8], the authors of this paper, together with Koessler et al., capitalize from the knowledge afforded by the graph theory and exploit the tree representation of the secondary RNA structure to build a predictive model that identifies whether a given tree structure is RNA-like or not RNA-like. In this work, we now consider the dual graph representation.

In particular, all possible dual graph representations of orders 2, 3, and 4 are given in the RAG database and the corresponding structures are classified as either (a) representing a known structure or (b) not representing a known structure. Those not representing a known structure are further classified as either likely to represent a structure in the future, that is, having the characteristics of RNA structure making it likely that such a structure will be identified at some point, or not RNA-like in structure. For the dual graphs of order 5, the database contains 18 structures that have been identified and states that there are 108 possible dual graphs of order 5. This number was determined by a graph growing algorithm. Eighteen of these 108 graphs are verified as representing existing RNA structures and the remaining 90 structures are classified as either RNA-like or not RNA-like in the most recent update for the database by Izzo et al. [9]. This update describes two methods by which the unverified structures are classified. The Laplacian eigenvalues for each structure were transformed using a linear regression to obtain two values for each structure and then these values were applied in two clustering algorithms, namely, a partitioning method called PAM and a k-nearest neighbor algorithm [9]. They state that 63 are RNA-like and 36 are not and that 45 are RNA-like and 45 are not RNA-like by the two methods, respectively. Since only 18 structures are provided in the database, our objectives were to (1) combinatorially analyze the structures of the 90 dual graphs of order 5 not in the RAGs database and (2) predict which of those 90 dual graphs of order 5 are RNA-like in structure via graph-theoretic information from chemical graph theory and mathematical graph theory.

Our findings differ significantly from those of Izzo et al. [9]. We find by using a combinatorial algorithm to construct all possible graphs with the given constraints that there are 118 instead of 108 possible dual graphs of order 5. Furthermore, we show that indeed almost all of the structures in the database with 5 vertices are RNA-like instead of approximately half as indicated in [9]. We feel that this is not too surprising. In the earlier version (2004) of the database, for instance, 8 of the 30 possible tree graphs were classified as not RNA-like, but in the updated version (2011), only 3 graphs are listed as not RNA-like. We expect that the remaining 3 topologies will be verified as RNA topologies as more RNA molecules are found. For example, genome-wide mapping of conserved RNA secondary structures reveals evidence for thousands of functional noncoding RNAs [10]. In the following sections, we discuss the dual graph representation and the graph-theoretic measures that we use. We then discuss the analysis and training together with the results.

The Dual Graph Representation of Secondary RNA Structure

Gan et al. [6] have used both tree graphs and the corresponding dual graphs which results in a multigraph representation of RNA secondary structures. Here, however, we will restrict our study to the multigraph representations of RNA secondary structures. As mentioned previously, the dual graphs can represent all types of RNA secondary structures, including the complex pseudoknot structures. When representing an RNA structure with a dual graphs, a vertex is used to represent stems (two or more complementary base pairs), and circular edges are used to represent the RNA motifs (hairpin loops, bulges, internal loops, and junctions). Dual graphs may contain multiple edges and loops; however, neither of these structures is required. Since a double-stranded RNA stem is connected to at most 2 strands on each side, every vertex v must have at most degree four. In fact, all vertices are of degree 4 except either (a) one of degree 2 or (b) two of degree 3. It follows that dual graphs of order n are of size 2n-1 [6]. Given these constraints, we use a constructive graph algorithm to enumerate the number of dual graphs of order five. These 118 graphs may be found in Figure 6.

Previous Results for the Dual Graph Model

The dual graph representation with 4 or fewer vertices was used in a previous work to train an artificial neural network (ANN) to recognize a dual graph as having the structural properties of secondary RNA [11]. In particular, we quantified the structures using graph invariants from graph theory and molecular descriptors from chemical graph theory and then used a multilayer perceptron artificial neural network to verify the findings in the RAG database regarding the classification of the dual graphs of order four. A set of ten structures that have been verified as RNA-like were chosen randomly from the set of 11 RNA-like graphs of order four. These ten graphs, in addition to the ten classified as not RNA-like, comprised the training set for the ANN. All graphs that were classified to be RNA-like in the database were predicted to be RNA-like by the neural network. However, one of the graphs whose structure represents a known topology was predicted with much lower probability than the other graphs in the set. Since this earlier work, the RAG database has been updated and a dual graph considered to be not RNA-like has since been changed to RNA verified [9]. This particular structure is similar to the structure that the neural network predicted to be RNA-like, but with lower probability. Given the updated information in the RAG database, we can now remove the incorrectly predicted structure from the training set and expect our results to confirm the new information. Thus, even with incorrect information in the training set, the graph-based measures were sufficient to characterize the topology of the RNA-like dual graphs of order 4.

We extend these findings to the dual graphs of order five. For this work we do not use the predicted classifications of the RAG database. We use only the verified structures in the database of which there are 18 of order 5 as well as 17 of order 4. We refer to these verified structures as RNA graphs. We consider the remaining 13 graphs of order 4 and 100 graphs of order 5 as unclassified structures.

GRAPH-THEORETIC MEASURES FOR THE DUAL GRAPHS

As stated previously, the dual graph representation method of the RAG database results in a multigraph. We began by writing a program in the computer language Python which generates the 30 multigraphs of order 4 and the 118 multigraphs of order 5. This program realized edgeless graphs as networkx [12] multigraph structures and then generated edges in accordance with the secondary RNA structural constraints. Several algorithms to calculate topological indices and graph invariants were also written in Python based on the networkx graph object.

In order to draw upon the wealth of graph-theoretic measures to quantify the topologies of the RNA model, we note that the majority of such measures is defined for simple graphs, and simple graphs do not have multiple edges nor do they have loops. Given that the dual graph representation has both, we therefore determined the line graph of each of the dual graphs and we use the line graph representation to determine the graphical measures of the topologies such as the clique number (both edge and vertex), independence number, and diameter and domination numbers. The line graph of a graph G is defined as the graph whose vertex set is the edge set of G and two vertices are adjacent in the line graph if the corresponding edges in G are incident. Thus the vertices in the line graph correspond to the regions in the RNA molecule with unpaired bases. Using the line graph of the dual graph allows quantification of the structural properties of the RNA molecule with graph-theoretic descriptors, even those containing pseudoknots. An algorithm for generating the line graph of a multigraph was also written in

Python, and this algorithm was used to generate the 30 + 118 line graphs of the multigraphs of orders 4 and 5. The multigraphs and line graphs were verified by the authors via a comparison to the RAG database and by manual inspection and reconstruction.

To calculate the graph-based measures, we used the GraphTheory package in Maple, the networkx package in Python, and the network analysis plugin in Cytoscape 2.8.2 [13]. Many invariants—such as diameter, radius, and clique numbers—were calculated either in all 3 or in 2 of the 3. This allowed us to verify the results of each software tool or to identify any variations in the graph invariant and/or topological index techniques. Most but not all of the measures we used can be found in at least one of the three tools mentioned above. In order to calculate a number of the measures, especially the topological indices, we need to determine the distance matrix of the graph. In a simple graph, the distance from a vertex u to itself is zero. However, with the presence of a loop, we considered three possibilities. One is the standard distance matrix with zeros down the diagonal. In the second case, we place either a zero or a one, depending on whether the vertex has a loop. In the third case, we not only modify the diagonal but also if the shortest path traversal includes a vertex with a loop, we include the loop in the edge count of edges encountered. Thus we are requiring any traversal to include a loop when encountered. We also modified the Balaban index, motivated by recent results using random walks on graphs. To find the distance between two vertices u and v in a dual graph, observe that if u is a vertex with a loop and if there are two edges between u and v, then the following options arise:(i)one of the edges from u to v is traversed;(ii)the other edge from u to v is traversed;(iii)the loop is traversed followed by a traversal of one of the edges.

There are four possibilities, so each traversal is assigned an equal weight of 1/4. The shortest route is the traversal of one edge which can happen in two ways. Thus the distance from u to v is 1/2.

We subsequently calculated approximately 100 invariants and indices of the multigraphs and line graphs using the 3 graph theoretic software tools mentioned above, some with slight modifications to account for the presence of loops and multiple edges. The invariants were normalized with respect to the values of the graphs that are verified as representing a known RNA secondary structure.

ASSESSING THE GRAPH-THEORETIC MEASURES AS DESCRIPTORS OF RNA TOPOLOGY

The total invariants were divided into 3 categories—topological indices, graph-theoretic invariants, and measures on line graphs. In order to compare the efficacy of an invariant or index in discriminating between the RNA graphs and the remaining graphs, the invariants were normalized with respect to the RNA graphs of orders 4 and 5, respectively. In particular, for each invariant or index, we calculated the mean and standard deviation of the RNA graphs of order 4, after which we used this mean and standard deviation to normalize all the values for graphs of order 4 of the given invariant or index according to the formula

$$I_{\text{normalized}} = \frac{I_{\text{observed}} - \text{Mean}_{\text{RNA graphs}}}{\text{StDev}_{\text{RNA graphs}}}. \tag{1}$$

Figure 1 shows the 10% percentile to 90% percentile of each normalized index/invariant in the topological indices collection as a rectangle. The mean is zero and the standard deviation is one for the given index across the RNA graphs of order 4. The values of the unverified graphs of order 4 are shown as points, so that a point inside the given rectangle is between the 10% and 90% percentiles for that index. The dotted lines correspond to the numbers of standard deviations from the mean. In general, if the values of the unverified graphs are close to the values of the verified graphs (i.e., if the dots are all on or inside the rectangle for a given invariant), then this invariant will not be useful as a factor in a machine learning classifier. For example, invariants 12–18 are poor predictors of RNA-like versus not RNA-like simply because there is not enough variation among the values for all the multigraphs of order 4. A support vector machine, a neural network, and logistic regression trained on the multigraphs of order 4 using invariants 12–18 were no better classifiers than was the uniformly random assignment to different classes, as evidence by the Receiver Operating Characteristic analysis in which the area under the curve for each method was approximately 0.5.

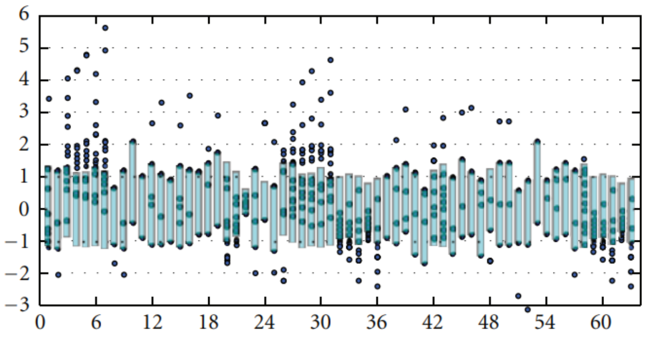


Figure 1: Topological invariants for RNA multigraphs of order 4.

In contrast, invariants 2 through 7 in Figure 2 are variations on the Balaban index for the graphs considered as simple graphs, and invariants 24–32 are variations on the Balaban index for the graphs considered as multigraphs. Like invariants 11–19, there is insufficient discrimination in each of the remaining topological indices, which includes eigenvalues of the Laplacian, the clustering coefficient, variations on the Weiner index, variations on the Randic index, variations on the Platt index, various measures of centrality, associativity, and connectivity, topological coefficients, and stress. Unfortunately, even though the Balaban indices and their variations have better discriminatory ability, they alone do not characterize between those graphs verified as RNA and those that are unclassified.

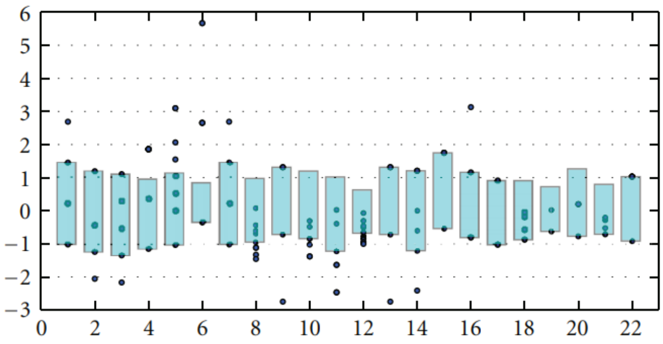


Figure 2: Variations on the Balaban index.

First, we find that variations on the clique number yield another factor with the ability to discriminate between the RNA graphs and the unclassified graphs. Observe invariants 4, 5, and 6 in Figure 2. Second, invariants and indices based on the line graphs retain more of the information contained in a multigraph than does a simple graph interpretation of a multigraph, while additionally allowing standard algorithms to calculate the invariants. For example, in Figure 3, invariants 7 through 12 are the chromatic index, the chromatic number, the circular chromatic index, the circular chromatic number, and the edge chromatic number, respectively, of line graphs of order 4.

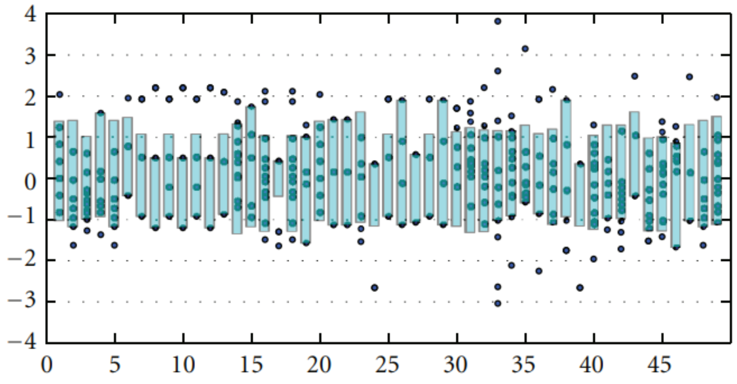


Figure 3: Line graph invariants.

Invariants 16 through 18 are variations on the clustering coefficient, and invariant 33 is the network centrality of the line graphs. Invariant 21 is the diameter, invariant 27 is the independence number, and invariant 28 is the maximum degree of the line graphs. It is interesting to note that the Balaban index of the line graphs, invariant 4, is not a good discriminator.

RESULTS

There are 18 multigraphs of order 5 that have been verified so far. The consensus across several techniques—including clustering, machine learning, and nearest neighbor analysis—and across several different combinations of invariants and indices indicate that most, if not all, of the unverified graphs are RNA-like.

For example, a simple machine learning scheme is that of choosing one unverified graph to be in class 0 while the 18 verified are in class 1. The neural network is then trained and the remaining unclassified RNA graphs are tested. Overwhelmingly, most if not all were classified as being in the same class as the 18 verified—that is, assuming only one non-RNA-like graph confirmed that all the graphs are RNA-like independent of which unverified graph was chosen to be RNA-like.

Regression, neural network, and support vector machine analysis similarly confirm the observation above. Nearly all the graphs of order 5 are predicted to be RNA-like in each run, and the ones that are predicted to be not RNA-like change from one run to the next.

Subsequently, we applied several different classifier/clustering techniques to graphs of order 5. Many different subsets of invariants and indices were used, but the invariant set suggested by the analysis above—as well as the one that produced the best results—was the following:(i) Four to eight variations of the Balaban index for multigraphs;(ii)Clique numbers;(iii)Chromatic numbers of the line graphs;(iv)Edge chromatic number of a line graphs;(v)Clique numbers of the line graphs;(vi)Diameters of the line graphs;(vii)Independence numbers of the line graphs;(viii) Maximum degrees of the line graphs.

Likewise, many different partitions of the total data were used, including the restriction to order 5 graphs known to be RNA-like. Results were consistent across these variations.

In particular, clustering tended to group all unverified graphs of order 5 with the 18 verified to be RNA-like (see Figure 4). To further investigate, we ranked the 100 unverified graphs using nearest neighbor analysis, and then we clustered in two groups—the 50 closest to and the 50 furthest from the 18 verified structures. The 50 closest to the 18 verified formed a single cluster with the 18 (using biclustering and hierarchical clustering in the statistical language R). The 50 furthest from those verified likewise clustered with the 18, but in a somewhat interesting manner. Having determined a 5-cluster scheme to be the best, we found that one cluster contained only one of the 18 verified graphs of order 5, and this graph (105 in our numbering) was both a large distance from the other 17 and had no more than an r=0.49645 correlation with any of the other verified graphs.

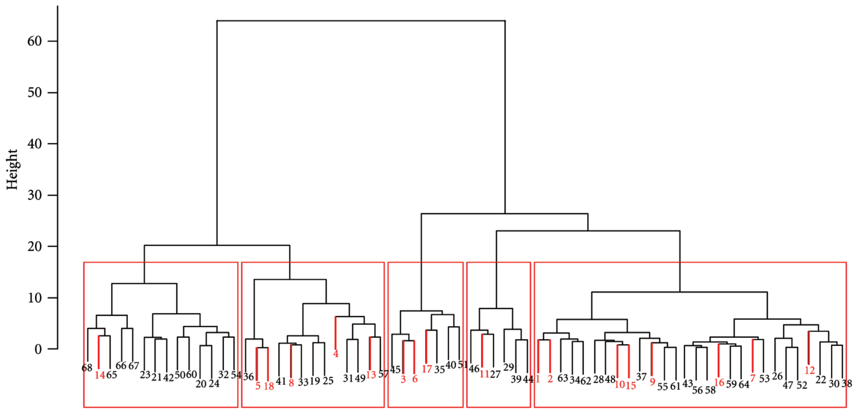


Figure 4: Clustering of the 50 graphs most distant from the 18 verified as RNA-like (in red).

Moreover, this was a rather large cluster containing 14 graphs of order 5, and, likely, if there were any graphs of order 5 that are eventually deemed to not be RNA-like, they would come from this cluster. However, the results seem to further support an interpretation of all the graphs of order 5 being RNA-like.

Data Domain Description

This interpretation motivated us to consider the problem to be a data domain description problem, also known as a one-class classification problem. In particular, rather than predict whether or not a graph is RNA-like, we instead explore the degree to which the 18 verified graphs typify the entire class of RNA-like graphs.

To do so, we use a "cognitive learning" approach in association with an artificial neural network [14]. While this is typically performed with a support vector machine [15, 16], our goal is to examine how the unverified RNA multigraphs of order 5 are distributed about the 18 verified multigraphs. In particular, the graded response of the neural network can be used to implement a genetic algorithm for successively refining the learning set of a neural network.

Suppose that we are given a training set P that contains examples from only one class of data along with a test set S of unclassified data that may or may not contain examples from another class. The method begins with a prior assumption: patterns that are many standard deviations away from any pattern in the training set form at least one other class of patterns. This assumption is used to generate an initial “negative example set”, N, of large σ patterns, after which the algorithm proceeds as follows.

(1) Train the neural network with P ∪ N.

(2) Classify the set S with the neural network. The classifications are numbers in [0, 1].

(3) Use the Receiver Operating Characteristic (ROC) or similar method to find the optimal threshold for distinguishing between patterns in N and patterns in P.

(4) Choose some number n of the highest scored patterns in S to be moved into P, being careful to stay above the threshold in step 3.

(5) Choose some number m of the lowest scored patterns in S to be moved into N, being careful to stay below the threshold in step 3.

(6) Move the q patterns in N and the r patterns in P not correctly classified into the set S.

(7) Eliminate the large sigma patterns (after the first iteration).

The algorithm proceeds either until S is empty, or through some set number of iterations. In practice, changes to N, P, and S are based on upper and lower thresholds based on the results of step 3.

Although the process is closely supervised in practice, the goal is to mimic the cognitive learning process of regrouping via reinforcement. Ideally, if there is more than one class in the initial P ∪ S set of patterns, then a two-class classifier will emerge in the process. If there is only one class in P ∪ S, then the algorithm will proceed until all (or in practice, most) of the patterns initially in S are in P and all the patterns in N are large sigma patterns. Moreover, the rate at which a pattern moves into P can be used as a measure of how close those patterns are to those in P itself.

The algorithm was tested on several standardized data sets from various sources and repositories. When there are two or more distinct classes, which is to say that S contains one or more classes distinct from P initially, then the algorithm stabilizes to a distinct non-P class containing N in each iteration. When there is only one class overall, then the set N is eventually empty. Within a domain description problem, the final set of patterns in N,by which we denote Nf , is significant in that it differs the most in some sense from the initial P class.

The latter was the case with the classification of the RNAlike multigraphs of order 5. In each of 10 trials, the set N became empty after a relatively few number of iterations. However, the final set N_f differed only slightly between trials and is accurately represented by 9 graphs. The graphs in N_f likewise were quite similar, in that each of the graphs contained a triangle with at least one vertex of degree 4.

Moreover, as N began to lose graphs in the algorithm above, the graphs that tended to remain the longest were those graphs containing triangles with at least one vertex having degree 3 or 4, as illustrated in Figure 5. Finally, the set N_f had no discernible relationship to the clustering or nearest neighbor results discussed earlier, further suggesting that all the multigraphs of order 5 are RNA-like.

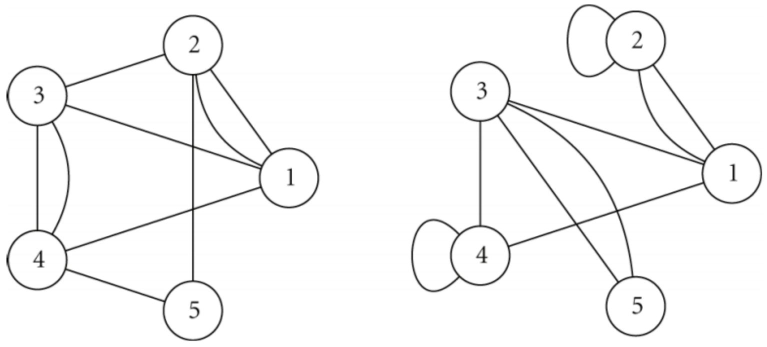
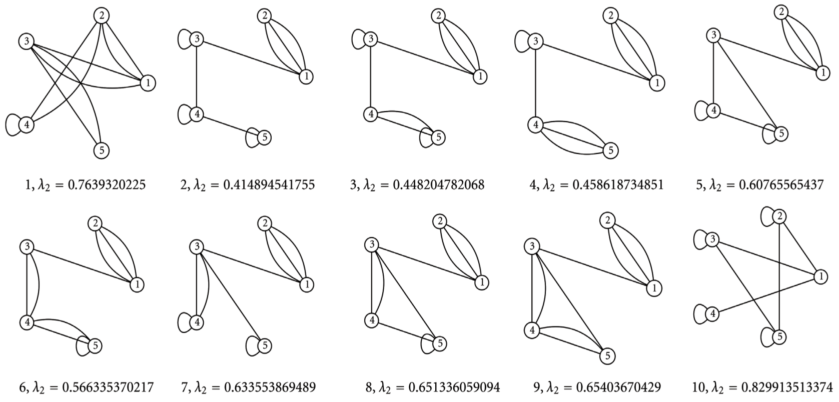
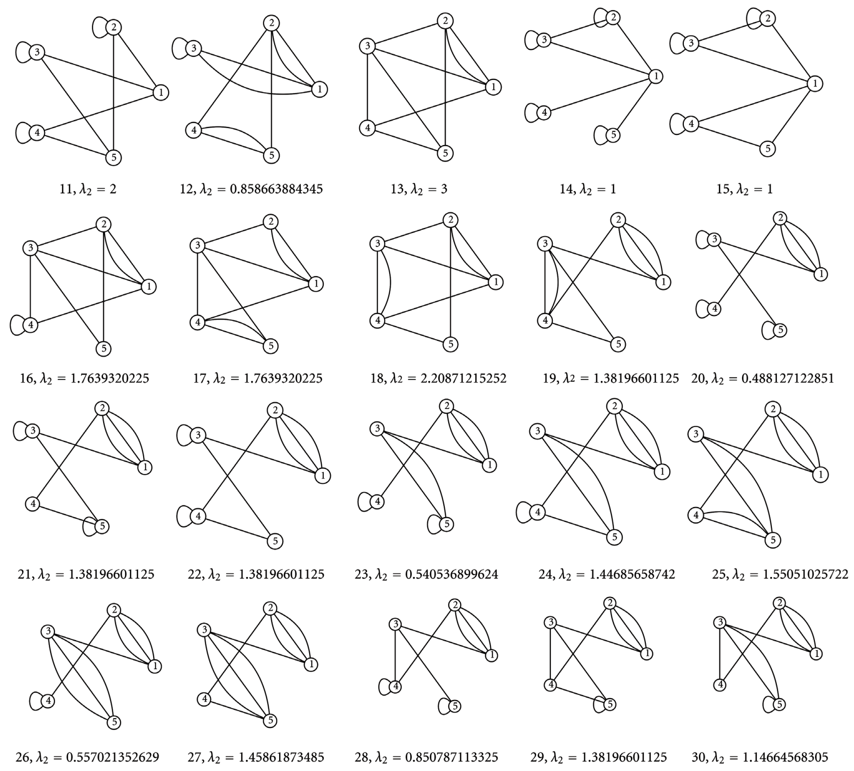
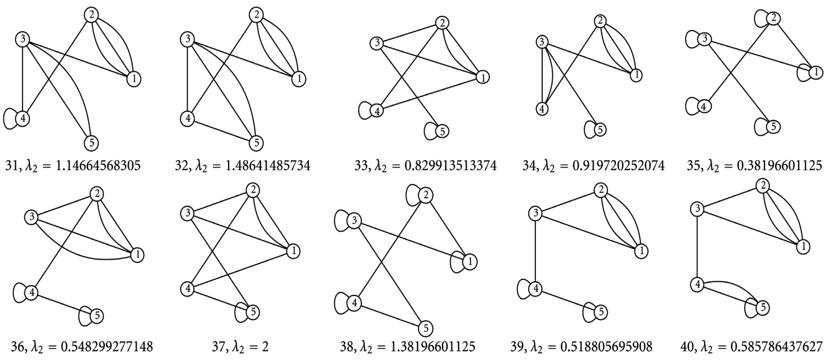


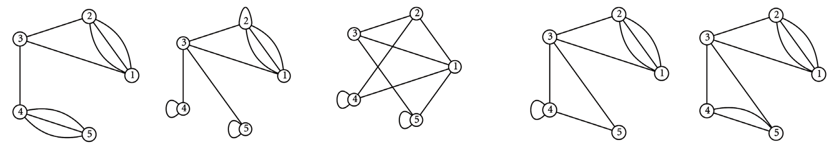
Figure 5: Two graphs from N_f.

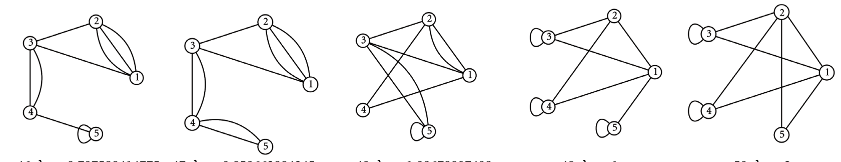

1, λ_2 = 0.7639320225
2, λ_2 = 0.414894541755
3, λ_2 = 0.448204782068
4, λ_2 = 0.458618734851
5, λ_2 = 0.60765565437
6, λ_2 = 0.566335370217
7, λ_2 = 0.633553869489
8, λ_2 = 0.651336059094
9, λ_2 = 0.65403670429
10, λ_2 = 0.829913513374

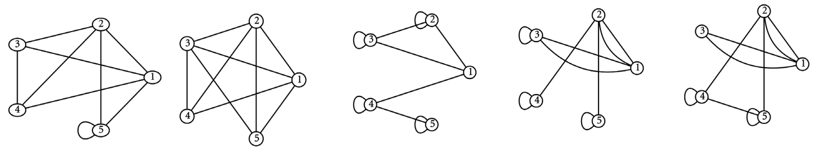

11, $\lambda_2 = 2$
12, $\lambda_2 = 0.858663884345$
13, $\lambda_2 = 3$
14, $\lambda_2 = 1$
15, $\lambda_2 = 1$
16, $\lambda_2 = 1.7639320225$
17, $\lambda_2 = 1.7639320225$
18, $\lambda_2 = 2.20871215252$
19, $\lambda_2 = 1.38196601125$
20, $\lambda_2 = 0.488127122851$
21, $\lambda_2 = 1.38196601125$
22, $\lambda_2 = 1.38196601125$
23, $\lambda_2 = 0.540536899624$
24, $\lambda_2 = 1.44685658742$
25, $\lambda_2 = 1.55051025722$
26, $\lambda_2 = 0.557021352629$
27, $\lambda_2 = 1.45861873485$
28, $\lambda_2 = 0.850787113325$
29, $\lambda_2 = 1.38196601125$
30, $\lambda_2 = 1.14664568305$

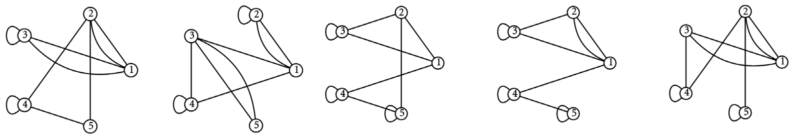
(a)

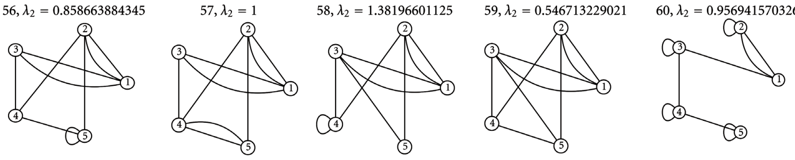

31, $\lambda_2 = 1.14664568305$
32, $\lambda_2 = 1.48641485734$
33, $\lambda_2 = 0.829913513374$
34, $\lambda_2 = 0.919720252074$
35, $\lambda_2 = 0.38196601125$
36, $\lambda_2 = 0.548299277148$
37, $\lambda_2 = 2$
38, $\lambda_2 = 1.38196601125$
39, $\lambda_2 = 0.518805695908$
40, $\lambda_2 = 0.585786437627$


41, λ = 0.60765565437 42, λ = 1 43, λ_2 = 1.58578643763 44, λ_2 = 1 45, λ_2 = 1

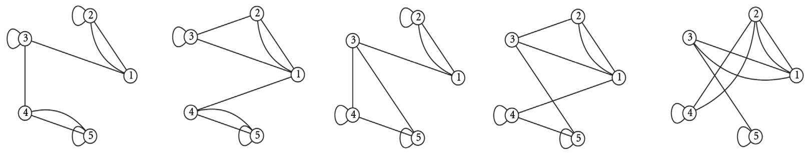

46, λ_2 = 0.707598414775 47, λ_2 = 0.858663884345 48, λ_2 = 1.09678807409 49, λ_2 = 1 50, λ_2 = 2

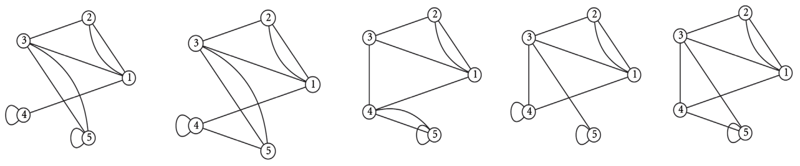

51, λ_2 = 2 52, λ_2 = 3 53, λ_2 = 0.518805695908 54, λ_2 = 0.858663884345 55, λ_2 = 0.858663884345

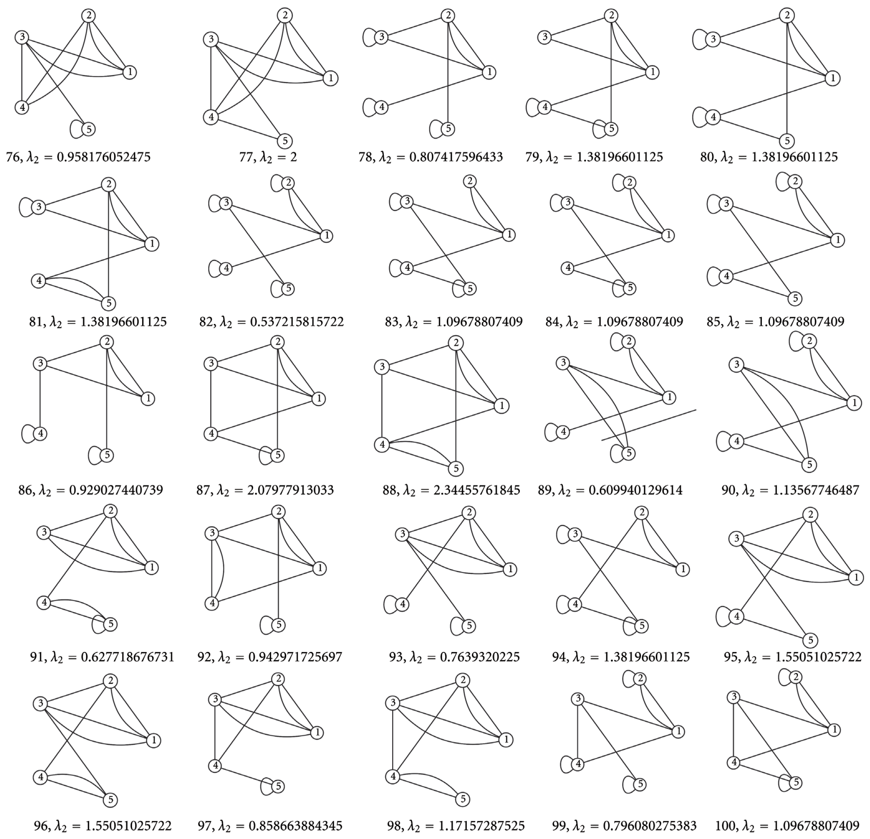

56, λ_2 = 0.858663884345 57, λ_2 = 1 58, λ_2 = 1.38196601125 59, λ_2 = 0.546713229021 60, λ_2 = 0.956941570326


61, λ_2 = 1.70759841478 62, λ_2 = 1.85926793352 63, λ_2 = 2 64, λ_2 = 2.7639320225 65, λ_2 = 0.407137631949

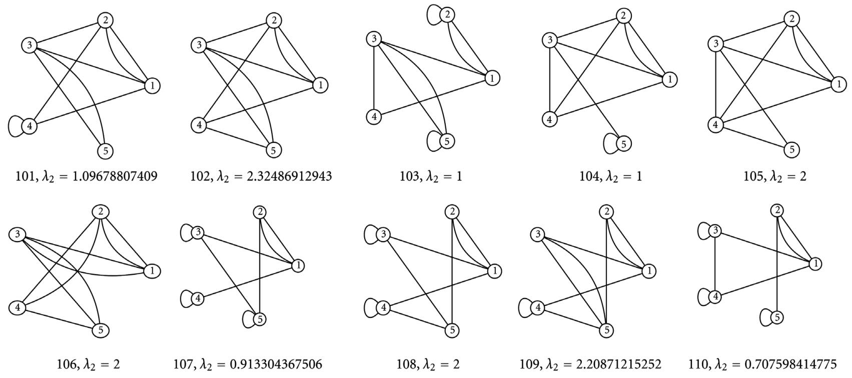
(b)


66, λ_2 = 0.438447187191 67, λ_2 = 0.625324415937 68, λ_2 = 0.585786437627 69, λ_2 = 1.48342018255 70, λ_2 = 0.65403670429


71, λ_2 = 0.839933268715 72, λ_2 = 1.85103885825 73, λ_2 = 1.14422749336 74, λ_2 = 1 75, λ_2 = 1.65707691722


76, $\lambda_2 = 0.958176052475$
77, $\lambda_2 = 2$
78, $\lambda_2 = 0.807417596433$
79, $\lambda_2 = 1.38196601125$
80, $\lambda_2 = 1.38196601125$
81, $\lambda_2 = 1.38196601125$
82, $\lambda_2 = 0.537215815722$
83, $\lambda_2 = 1.09678807409$
84, $\lambda_2 = 1.09678807409$
85, $\lambda_2 = 1.09678807409$
86, $\lambda_2 = 0.929027440739$
87, $\lambda_2 = 2.07977913033$
88, $\lambda_2 = 2.34455761845$
89, $\lambda_2 = 0.609940129614$
90, $\lambda_2 = 1.13567746487$
91, $\lambda_2 = 0.627718676731$
92, $\lambda_2 = 0.942971725697$
93, $\lambda_2 = 0.7639320225$
94, $\lambda_2 = 1.38196601125$
95, $\lambda_2 = 1.55051025722$
96, $\lambda_2 = 1.55051025722$
97, $\lambda_2 = 0.858663884345$
98, $\lambda_2 = 1.17157287525$
99, $\lambda_2 = 0.796080275383$
100, $\lambda_2 = 1.09678807409$

(c)


101, $\lambda_2 = 1.09678807409$
102, $\lambda_2 = 2.32486912943$
103, $\lambda_2 = 1$
104, $\lambda_2 = 1$
105, $\lambda_2 = 2$
106, $\lambda_2 = 2$
107, $\lambda_2 = 0.913304367506$
108, $\lambda_2 = 2$
109, $\lambda_2 = 2.20871215252$
110, $\lambda_2 = 0.707598414775$

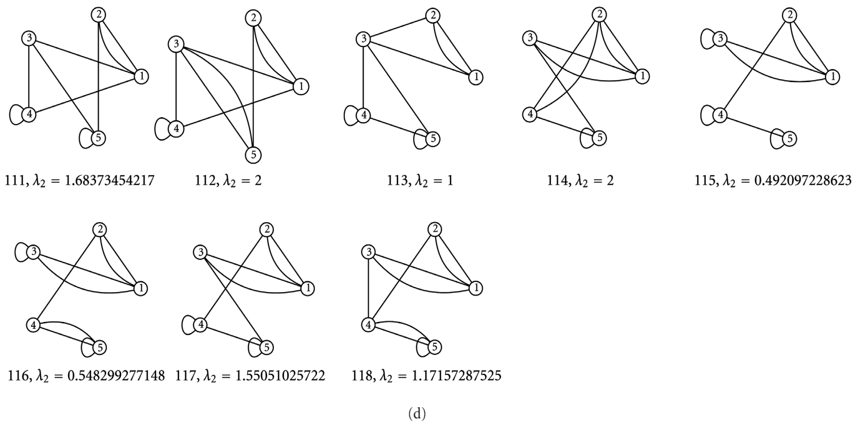

111, λ_2 = 1.68373454217
112, λ_2 = 2
113, λ_2 = 1
114, λ_2 = 2
115, λ_2 = 0.492097228623
116, λ_2 = 0.548299277148
117, λ_2 = 1.55051025722
118, λ_2 = 1.17157287525
(d)

Figure 6: The 118 multigraphs of order 5.

CONCLUSION

The most reasonable conclusion of this extensive analysis is that all the graphs of order 5 are likely to be verified as RNA structures. Indeed, across several variations of nearest neighbor analysis, machine learning, and clustering techniques using a variety of subsets of different graph invariants and topological indices, we consistently found that more than 90% of the unclassified graphs were closer to one of the 18 already verified as an RNA structure than the 18 were to each other.

This result is not surprising. Initial classification of the graph structures in the database RAG classified more than half of the dual graphs of order 4 as not RNA-like in structure. However, as more secondary RNA structures were identified, an update to the RAG database now predicts only a third to be not RNA-like in structure [9]. We predict that as the number of new motifs continues to increase, eventually almost all structures will be classified as RNA-like or verified as an RNA topology. Does this mean that the graph model in the database is too coarse to be of value and therefore should not be pursued as a model to characterize secondary RNA structure? No, not at all. It does suggest however that the model needs to contain more information in order to be discriminating. One way this can be achieved is by assigning weights to the vertices and edges based on the number of nucleotides, bases, and bonds in the respective stems and regions with unpaired bases. Karklin et al. [17] developed a labeled dual graph representation and defined a similarity measure using marginalized kernels. Using this measure they train support vector machine classifiers to identify known families of

RNAs from random RNAs with similar statistics. They achieved better than seventy percent accuracy using these biologically relevant vertex and edge labels. Efforts to synthesize RNA molecules for various purposes such as novel drug applications as well as efforts to develop efficient genome-wide screens for RNA molecules from existing families may be aided by the graph representation in the RAG database when coupled with vertex and edge weighting schemes. Indeed, the authors have successfully used vertex weighted graphs to characterize the residue structure of amino acids in order to build a predictive model of binding affinity levels resulting from single point mutations [18]. Future work naturally points to using vertex weighted graphs for the characterization of a secondary RNA structure. Information revealed by the labeled dual graph representation which shows that a secondary RNA structure is not consistent with those known to be found in nature can be considered a valuable resource for biotechnological applications, automated discovery of uncharacterized RNA molecules, and computationally efficient algorithms that can be used in conjunction with other methods for RNA structure identification.

REFERENCES

1. S. C. Flores and R. B. Altman, "Turning limited experimental information into 3D models of RNA," RNA, vol. 16, no. 9, pp. 1769–1778, 2010.
2. F. Ding, C. A. Lavender, and K. M. Weeks, "Three-dimensional RNA structure renement by hydroxyl radical probing," Nature Methods, vol. 9, pp. 603–608, 2012.
3. M. Zuker, "Mfold web server for nucleic acid folding and hybridization prediction," Nucleic Acids Research, vol. 31, no. 13, pp. 3406–3415, 2003.
4. I. L. Hofacker, "Vienna RNA secondary structure server," Nucleic Acids Research, vol. 31, no. 13, pp. 3429–3431, 2003.
5. J. P. Abrahams, M. van den Berg, E. van Batenburg, and C. Pleij, "Prediction of RNA secondary structure, including pseudoknotting, by computer simulation," Nucleic Acids Research, vol. 18, no. 10, pp. 3035–3044, 1990.
6. H. H. Gan, D. Fera, J. Zorn et al., "RAG: RNA-As-Graphs database—concepts, analysis, and features," Bioinformatics, vol. 20, no. 8, pp. 1285–1291, 2004.
7. F. Harary and G. Prins, "The number of homeomorphically irreducible trees, and other species," Acta Mathematica, vol. 101, no. 1-2, pp. 141–162, 1959.
8. D. R. Koessler, D. J. Knisley, J. Knisley, and T. Haynes, "A predictive model for secondary RNA structure using graph theory and a neural network," BMC Bioinformatics, vol. 11, no. 6, article 21, 2010.
9. J. A. Izzo, N. Kim, S. Elmetwaly, and T. Schlick, "RAG: an update to the RNA-As-Graphs resource," BMC Bioinformatics, vol. 12, article 219, 2011.
10. M. Guttman, I. Amit, M. Garber et al., "Chromatin signature reveals over a thousand highly conserved large non-coding RNAs in mammals," Nature, vol. 458, no. 7235, pp. 223–227, 2009.
11. A. Rockney, predictive model which uses descriptors of RNA secondary structures derived from graph theory [M.S. thesis], East Tennessee State University, 2011, http://libraries.etsu.edu/record=b2339936~S1a.
12. A. Hagberg, A. Schult, and P. Swart, "Exploring network structure, dynamics, and function using NetworkX," in Proceedings of the 7th Python in Science Conference (SciPy'08), G. Varoquaux, T. Vaught,

and J. Millman, Eds., p. 1115, Pasadena, Calif, USA, August 2008.

13. M. E. Smoot, K. Ono, J. Ruscheinski, P. L. Wang, and T. Ideker, "Cytoscape 2.8: new features for data integration and network visualization," Bioinformatics, vol. 27, no. 3, Article ID btq675, pp. 431–432, 2011.
14. O. Boehm, D. Hardoon, and L. Manevitz, "Classifying cognitive states of brain activity via one-class neural networks with feature selection by genetic algorithms," International Journal of Machine Learning and Cybernetics, vol. 2, no. 3, pp. 125–134, 2011.
15. D. Tax and R. Duin, "Data domain description using support vectors," in Proceedings of the European Symposium on Articial Neural Networks (ESANN'99), pp. 251–256, D-Facto Public, Brugge, Belgium, April 1999.
16. C. Elkan and K. Noto, "Learning classiers from only positive and unlabeled data," in Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'08), pp. 213–220, ACM, New York, NY, USA, 2008.
17. Y. Karklin, R. F. Meraz, and S. R. Holbrook, "Classification of non-coding RNA using graph representations of secondary structure," Pacific Symposium on Biocomputing, pp. 4–15, 2005.
18. D. Knisley and J. Knisley, "Predicting protein-protein interactions using graph invariants and a neural network," Computational Biology and Chemistry, vol. 35, no. 2, pp. 108–113, 2011.

INDEX