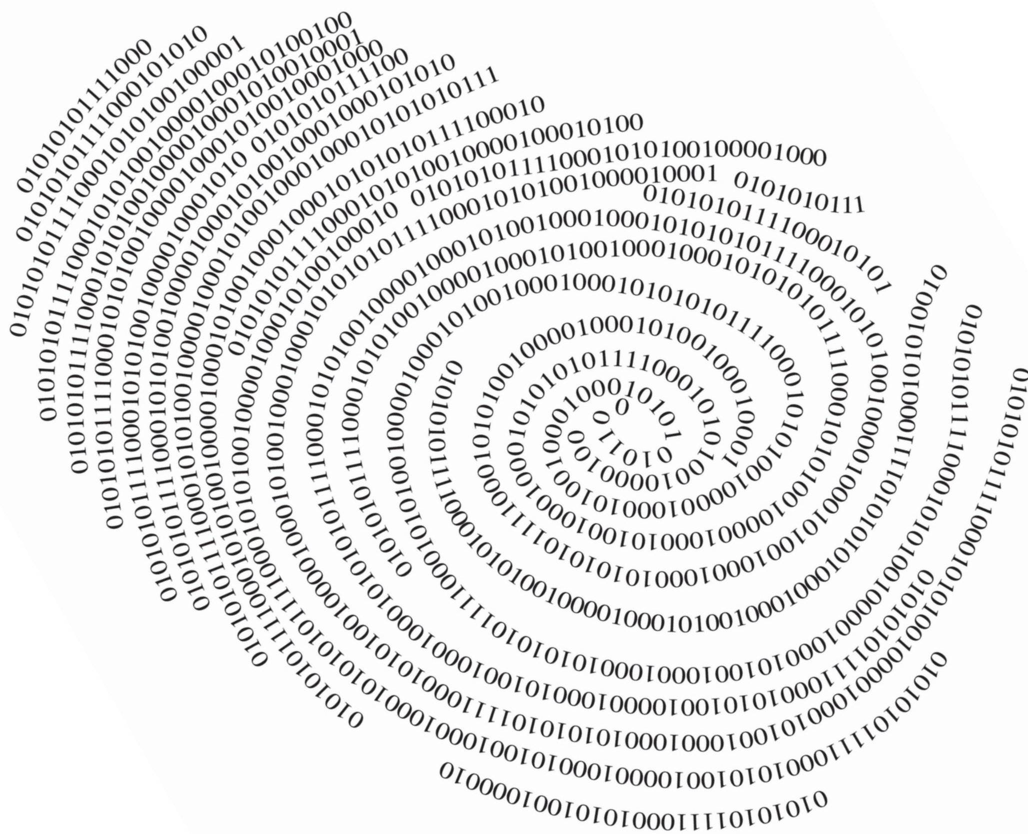
Эйял Вирсански



Генетические алгоритмы на Python

ДМК пресс
издательство

Эйял Вирсански

Генетические алгоритмы на Python

Hands-On Genetic Algorithms with Python

Applying genetic algorithms to solve real-world deep learning and artificial intelligence problems

Eyal Wirsansky

Packt>

BIRMINGHAM - MUMBAI

Генетические алгоритмы на Python

Применение генетических алгоритмов к решению задач глубокого обучения и искусственного интеллекта

Эйял Вирсански

Москва, 2020

УДК 004.421
ББК 32.811
В52

Вирсански Э.
В52 Генетические алгоритмы на Python / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2020. – 286 с.: ил.

ISBN 978-5-97060-857-9

Там, где традиционные алгоритмы бесполезны или не дают результата за обозримое время, на помощь могут прийти генетические алгоритмы. Они позволяют решить целый комплекс сложных задач, в том числе связанных с искусственным интеллектом, упростить оптимизацию непрерывных функций, выполнять реконструкцию изображений и многое другое.

Книга поможет программистам, специалистам по обработке данных и энтузиастам ИИ, интересующимся генетическими алгоритмами, подступиться к стоящим перед ними задачам, связанным с обучением, поиском и оптимизацией, а также повысить качество и точность результатов в уже имеющихся приложениях.

Для изучения материала книги требуются владение языком Python на рабочем уровне и базовые знания математики и информатики.

УДК 004.421
ББК 32.811

First published in the English language under the title 'Hands-On Genetic Algorithms with Python'. Russian language edition copyright © 2020 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-83855-774-4 (англ.)
ISBN 978-5-97060-857-9 (рус.)

Copyright © Packt Publishing, 2020
© Оформление, издание, перевод, ДМК Пресс, 2020

Моей любимой жене Джеки
за любовь, терпение и поддержку

Моим детям, Даниэлле и Лиарне,
чья творческая натура и артистические таланты
подвигли меня на написание этой книги

Содержание

Об авторе 13

О рецензенте 14

Предисловие 15

Часть I. ОСНОВЫ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ 19

Глава 1. Введение в генетические алгоритмы 20

Что такое генетические алгоритмы? 20
Дарвиновская эволюция 21
Аналогия с генетическими алгоритмами 21
Генотип 22
Популяция 22
Функция приспособленности 23
Отбор 23
Скрещивание 23
Мутация 24
Теоретические основы генетических алгоритмов 24
Теорема о схемах 25
Отличия от традиционных алгоритмов 26
Популяция как основа алгоритма 27
Генетическое представление 27
Функция приспособленности 27
Вероятностное поведение 27
Преимущества генетических алгоритмов 28
Глобальная оптимизация 28
Применимость к сложным задачам 29
Применимость к задачам, не имеющим математического представления 30
Устойчивость к шуму 30
Распараллеливание 30
Непрерывное обучение 31
Ограничения генетических алгоритмов 31
Специальные определения 31
Настройка гиперпараметров 31
Большой объем счетных операций 32
Преждевременная сходимость 32
Отсутствие гарантированного решения 32
Сценарии применения генетических алгоритмов 32
Резюме 33
Для дальнейшего чтения 33

Глава 2. Основные компоненты генетических алгоритмов 34
Базовая структура генетического алгоритма 34
Создание начальной популяции 36
Вычисление приспособленности 36
Применение операторов отбора, скрещивания и мутации 36
Проверка условий остановки 37
Методы отбора 37
Правило рулетки 38
Стохастическая универсальная выборка 39
Ранжированный отбор 40
Масштабирование приспособленности 41
Турнирный отбор 42
Методы скрещивания 43
Одноточечное скрещивание 43
Двухточечное и ***k***-точечное скрещивание 44
Равномерное скрещивание 45
Скрещивание для упорядоченных списков 45
Упорядоченное скрещивание 46
Методы мутации 47
Инвертирование бита 48
Мутация обменом 48
Мутация обращением 48
Мутация перетасовкой 49
Генетические алгоритмы с вещественным кодированием 49
Скрещивание смешением 50
Имитация двоичного скрещивания 51
Вещественная мутация 53
Элитизм 53
Образование ниш и разделение 54
Последовательное и параллельное образование ниш 56
Искусство решения задач с помощью генетических алгоритмов 57
Резюме 58
Для дальнейшего чтения 58

Часть II. РЕШЕНИЕ ЗАДАЧ С ПОМОЩЬЮ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ 59

Глава 3. Каркас DEAP 60
Технические требования 60
Введение в DEAP 61
Использование модуля creator 62
Создание класса Fitness 62
Определение стратегии приспособления 62
Хранение значения приспособленности 63
Создание класса Individual 64
Использование класса Toolbox 64

Создание генетических операторов........65
Создание популяции........66
Вычисление приспособленности........66
Задача OneMax........67
Решение задачи OneMax с помощью DEAP........67
Выбор хромосомы........67
Вычисление приспособленности........68
Выбор генетических операторов........68
Задание условия остановки........68
Реализация средствами DEAP........69
Подготовка........69
Эволюция решения........72
Выполнение программы........75
Использование встроенных алгоритмов........76
Объект Statistics........77
Алгоритм........77
Объект logbook........77
Выполнение программы........78
Зал славы........79
Эксперименты с параметрами алгоритма........81
Размер популяции и количество поколений........81
Оператор скрещивания........83
Оператор мутации........84
Оператор отбора........87
Размер турнира и его связь с вероятностью мутации........88
Отбор по правилу рулетки........92
Резюме........93
Для дальнейшего чтения........94

Глава 4. Комбинаторная оптимизация........95

Технические требования........95
Поисковые задачи и комбинаторная оптимизация........96
Решение задачи о рюкзаке........96
Задача о рюкзаке 0-1 с сайта Rosetta Code........97
Представление решения........98
Представление задачи на Python........98
Решение с помощью генетического алгоритма........99
Решение задачи коммивояжера........102
Файлы эталонных данных TSPLIB........103
Представление решения........104
Представление задачи на Python........104
Решение с помощью генетического алгоритма........106
Улучшение результатов благодаря дополнительному исследованию и элитизму........109
Решение задачи о маршрутизации транспорта........114
Представление решения........115
Представление задачи на Python........116

Решение с помощью генетического алгоритма 118
Резюме 122
Для дальнейшего чтения 123

Глава 5. Задачи с ограничениями 124
Технические требования 124
Соблюдение ограничений в поисковых задачах 125
Решение задачи об *N* ферзях 125
Представление решения 126
Представление задачи на Python 128
Решение с помощью генетического алгоритма 129
Решение задачи о составлении графика дежурств медсестер 133
Представление решения 133
Жесткие и мягкие ограничения 134
Представление задачи на Python 135
Решение на основе генетического алгоритма 137
Решение задачи о раскраске графа 140
Представление решения 142
Жесткие и мягкие ограничения в задаче о раскраске графа 143
Представление задачи на Python 143
Решение с помощью генетического алгоритма 145
Резюме 149
Для дальнейшего чтения 150

Глава 6. Оптимизация непрерывных функций 151
Технические требования 151
Хромосомы и генетические операторы для задач с вещественными числами 152
Использование DEAP совместно с непрерывными функциями 153
Оптимизация функции Eggholder 154
Оптимизация функции Eggholder с помощью генетического алгоритма 155
Повышение скорости сходимости посредством увеличения частоты мутаций 158
Оптимизация функции Химмельблау 159
Оптимизация функции Химмельблау с помощью генетического алгоритма 161
Использование ниш и разделения для отыскания нескольких решений 165
Функция Симионеску и условная оптимизация 168
Условная оптимизация с помощью генетического алгоритма 170
Оптимизация функции Симионеску с помощью генетического алгоритма 171
Использование ограничений для нахождения нескольких решений 172
Резюме 173
Для дальнейшего чтения 173

Часть III. ПРИЛОЖЕНИЯ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ В ИСКУССТВЕННОМ ИНТЕЛЛЕКТЕ 174

Глава 7. Дополнение моделей машинного обучения методами выделения признаков 175

Технические требования 176
Машинное обучение с учителем 176
Классификация 177
Регрессия 179
Алгоритмы обучения с учителем 180
Выделение признаков в обучении с учителем 180
Выделение признаков для задачи регрессии Фридмана-1 181
Представление решения 182
Представление решения на Python 182
Решение с помощью генетического алгоритма 184
Выделение признаков для классификации набора данных Zoo 186
Представление задачи на Python 187
Решение с помощью генетического алгоритма 189
Резюме 191
Для дальнейшего чтения 191

Глава 8. Настройка гиперпараметров моделей машинного обучения 192

Технические требования 193
Гиперпараметры в машинном обучении 193
Настройка гиперпараметров 194
Набор данных Wine 195
Классификатор на основе адаптивного усиления 195
Настройка гиперпараметров с помощью генетического поиска на сетке 196
Тестирование качества классификатора с параметрами по умолчанию 198
Результаты традиционного поиска на сетке 198
Результаты генетического поиска на сетке 198
Прямой генетический подход к настройке гиперпараметров 199
Представление гиперпараметров 200
Оценка верности классификатора 200
Настройка гиперпараметров с помощью генетического алгоритма 201
Резюме 204
Для дальнейшего чтения 204

Глава 9. Оптимизация архитектуры сетей глубокого обучения 205

Технические требования 205
Искусственные нейронные сети и глубокое обучение 206
Многослойный перцептрон 207
Глубокое обучение и сверточные нейронные сети 208

Оптимизация архитектуры классификатора на основе глубокой сети 209
Набор данных Iris 209
Представление конфигурации скрытого слоя 210
Оценка верности классификатора 211
Оптимизация архитектуры МСП с помощью генетического алгоритма 212
Объединение оптимизации архитектуры с настройкой гиперпараметров 215
Представление решения 215
Вычисление верности классификатора 216
Оптимизация объединенной конфигурации МСП с помощью генетического алгоритма 217
Резюме 218
Для дальнейшего чтения 218

Глава 10. Генетические алгоритмы и обучение с подкреплением 219
Технические требования 219
Обучение с подкреплением 220
Генетические алгоритмы и обучение с подкреплением 221
OpenAI Gym 221
Интерфейс env 222
Решение окружающей среды MountainCar 223
Представление решения 225
Оценивание решения 225
Представление задачи на Python 226
Решение с помощью генетического алгоритма 226
Решение окружающей среды CartPole 229
Управление средой CartPole с помощью нейронной сети 230
Представление и оценивание решения 231
Представление задачи на Python 232
Решение с помощью генетического алгоритма 233
Резюме 236
Для дальнейшего чтения 237

Часть IV. РОДСТВЕННЫЕ ТЕХНОЛОГИИ 238

Глава 11. Генетическая реконструкция изображений 239
Технические требования 239
Реконструкция изображений из многоугольников 240
Обработка изображений на Python 240
Библиотеки обработки изображений на Python 240
Библиотека Pillow 241
Библиотека scikit-image 241
Библиотека opencv-python 241
Рисование с помощью многоугольников 242

Измерение степени различия двух изображений........243
Попиксельная среднеквадратическая ошибка........243
Структурное сходство (SSIM)........244
Применение генетических алгоритмов для реконструкции изображений........244
Представление и оценивание решения........245
Представление задачи на Python........246
Реализация генетического алгоритма........246
Добавление функции обратного вызова в код генетического алгоритма........249
Результаты реконструкции изображения........250
Применение попиксельной среднеквадратической ошибки........251
Применение индекса структурного сходства........253
Другие эксперименты........256
Резюме........257
Для дальнейшего чтения........258

Глава 12. Другие эволюционные и бионические методы вычислений........259
Технические требования........259
Эволюционные и бионические вычисления........260
Генетическое программирование........260
Пример генетического программирования – контроль по четности........262
Реализация с помощью генетического программирования........264
Упрощение решения........269
Оптимизация методом роя частиц........271
Пример применения PSO – оптимизация функции........272
Реализация оптимизации методом роя частиц........273
Другие родственные методы........277
Эволюционные стратегии........277
Дифференциальная эволюция........278
Муравьиный алгоритм оптимизации........278
Искусственные иммунные системы........278
Искусственная жизнь........279
Резюме........279
Для дальнейшего чтения........280

Предметный указатель........281

Об авторе

Эйял Вирсански – старший инженер-программист, лидер технического сообщества, исследователь и энтузиаст искусственного интеллекта. Начал карьеру как один из первопроходцев в области передачи голоса по IP-сетям и теперь может похвастаться более чем 20-летним опытом создания самых разных высокопроизводительных корпоративных решений. Еще будучи студентом, проявил интерес к генетическим алгоритмам и нейронным сетям. Одним из результатов его исследований стал новый алгоритм обучения с учителем, объединяющий достоинства обоих подходов.

Эйял возглавляет группу пользователей Java в Джексонвилле, а также виртуальную группу пользователей по теме «Искусственный интеллект в корпоративных приложениях» и ведет блог по искусственному интеллекту ai4java, ориентированный на разработчиков.

Я хочу поблагодарить свою семью и близких друзей за терпение, поддержку и ободрение на протяжении длительного процесса написания книги. Отдельное спасибо группе пользователей Python в Джексонвилле (PyJax) за отзывы и поддержку.

О рецензенте

Лайза Бэнг окончила бакалавриат по биологии моря в Калифорнийском университете в Санта-Крус и магистратуру по биоинформатике в Сеульском университете Сонсиль под руководством д-ра Кван Хви Чо. Магистерская диссертация была посвящена методу создания воспроизводимых моделей QSAR (поиск количественных соотношений структура–свойство) с применением блокнота Jupyter, одним из компонентов которого был генетический алгоритм, позволяющий уменьшить пространство поиска. В настоящее время ведется работа по его переводу на каркас DEAP-VS для совместимости с Python 3. Она также работала в системе здравоохранения Гейсингера, входящей в состав Института биомедицинской и трансляционной информатики, где применяла данные следующего поколения о секвенировании и электронные медицинские карты пациентов для анализа исходов раковых и других заболеваний. Сейчас работает в компании Ultragenyx Pharmaceutical, где занимается доклиническими испытаниями и использует методы биоинформатики и хемоинформатики для анализа редких наследственных болезней.

Спасибо моей семье, учителям и наставникам!

Предисловие

Берущие начало в эволюционной теории Дарвина, *генетические алгоритмы* являются одним из самых удивительных методов решения задач поиска, оптимизации и обучения. Они могут привести к успеху там, где традиционные алгоритмы не способны дать адекватные результаты за приемлемое время.

Данная книга покажет вам путь к овладению этим чрезвычайно мощным и вместе с тем простым подходом к разнообразным задачам, венцом которых станут приложения ИИ.

Вы узнаете, как работают генетические алгоритмы и когда имеет смысл их использовать. А заодно получите практический опыт их применения в разных областях с помощью популярного языка программирования Python.

Предполагаемая аудитория

Эта книга призвана помочь программистам, специалистам по обработке данных и энтузиастам ИИ, интересующимся генетическими алгоритмами, подступиться к стоящим перед ними задачам, связанным с обучением, поиском и оптимизацией, а также повысить качество и точность результатов в уже имеющихся приложениях.

Книга адресована также всем, кто сталкивался в своей практике с трудными задачами, в которых традиционные алгоритмы вообще бесполезны или не дают результата за обозримое время. Показано, как использовать генетические алгоритмы в качестве эффективного и в то же время простого подхода к решению сложных задач.

Структура книги

В главе 1 «Введение в генетические алгоритмы» приводится краткое введение в теорию и принципы работы генетических алгоритмов. Рассматриваются также различия между генетическими алгоритмами и традиционными методами, и описываются сценарии, в которых имеет смысл применять генетические алгоритмы.

В главе 2 «Основные компоненты генетических алгоритмов» более глубоко описываются основные компоненты и детали реализации генетических алгоритмов. Познакомившись с потоком управления, вы затем узнаете о различных компонентах и их реализациях.

В главе 3 «Каркас DEAP» дается введение в DEAP – мощный и гибкий каркас эволюционных вычислений, позволяющий решать практические задачи с помощью генетических алгоритмов. Мы продемонстрируем его использо-

вание на примере программы на Python, которая решает задачу OneMax – что-то вроде «Hello World» в мире генетических алгоритмов.

В главе 4 «Комбинаторная оптимизация» рассматриваются такие проблемы, как упаковка рюкзака, задача коммивояжера и задача маршрутизации транспорта, а также программы на Python для их решения с помощью генетических алгоритмов и каркаса DEAP.

Глава 5 «Задачи с ограничениями» представляет собой введение в задачи с ограничениями, например: задача об *N* ферзях, составление графика дежурства медсестер, задача о раскраске графа. Объясняется, как решить их на Python с помощью генетических алгоритмов и каркаса DEAP.

В главе 6 «Оптимизация непрерывных функций» обсуждаются задачи непрерывной оптимизации и их решения с помощью генетических алгоритмов. В качестве примеров приводятся функция Eggholder, функции Химмельблау и Симионеску. Попутно мы изучим концепции образования ниш, разделения и обработки ограничений.

В главе 7 «Дополнение моделей машинного обучения методами выделения признаков» речь пойдет о моделях машинного обучения с учителем. Объясняется, как генетические алгоритмы позволяют повысить качество этих моделей за счет выделения наилучшего подмножества признаков из представленных входных данных.

В главе 8 «Настройка гиперпараметров моделей машинного обучения» показано, как использовать генетические алгоритмы для улучшения качества моделей машинного обучения с учителем путем применения поиска на сетке, управляемого генетическим алгоритмом, или прямого генетического поиска.

В главе 9 «Оптимизация архитектуры сетей глубокого обучения» рассматриваются искусственные нейронные сети и описывается, как генетические алгоритмы позволяют улучшить модель, основанную на нейронной сети, путем оптимизации архитектуры последней. Вы узнаете, как сочетать оптимизацию архитектуры сети с настройкой гиперпараметров.

Глава 10 «Генетические алгоритмы и обучение с подкреплением» посвящена обучению с подкреплением. Применение генетических алгоритмов иллюстрируется на примере двух тестовых окружающих сред – MountainCar (машина на горе) и CartPole (балансировка стержня) – из библиотеки OpenAI Gym.

В главе 11 «Генетическая реконструкция изображений» описываются эксперименты по реконструкции одного хорошо известного изображения с помощью множества полупрозрачных многоугольников на основе генетических алгоритмов. Попутно вы приобретете полезный опыт обработки изображений и научитесь работать с соответствующими библиотеками на Python.

Глава 12 «Другие эволюционные и бионические методы вычислений» расширяет горизонты и знакомит еще с несколькими пришедшими из биологии приемами решения задач. Два из них – генетическое программирование и метод роя частиц – будут реализованы с помощью Python и каркаса DEAP.

Что необходимо для чтения этой книги

Чтобы получить максимум пользы от чтения книги, необходимо владеть языком Python на рабочем уровне и иметь базовые знания по математике и информатике. Знакомство с фундаментальными понятиями машинного обучения желательно, но не обязательно, поскольку в книге приводится краткое изложение всего необходимого.

Для выполнения приведенных в книге примеров понадобится версия 3.7 или более поздняя, а также несколько Python-пакетов (все они здесь же и описываются). Рекомендуется, хотя это и необязательно, установить какую-нибудь интегрированную среду разработки на Python (IDE), например PyCharm или Visual Studio Code.

Скачивание исходного кода

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Обозначения и графические выделения

В этой книге применяется ряд соглашений о наборе текста.

`CodeInText`: код в тексте, имена таблиц базы данных, папок и файлов, расширения имен файлов, пути к файлам, URL-адреса, данные, вводимые пользователем, и адреса в Твиттере. Например: «Метод класса `__init__()` создает набор данных».

Отдельно стоящие фрагменты кода набраны так:

```
self.X, self.y = datasets.make_friedman1(n_samples=self.numSamples,
                                         n_features=self.numFeatures,
                                         noise=self.NOISE,
                                         random_state=self.randomSeed)
```

Желая привлечь внимание к какой-то части кода, мы выделяем ее полужирным шрифтом, например:

```
self.regressor = GradientBoostingRegressor(random_state=self.randomSeed)
```

Команды операционной системы и результаты их работы выделяются так:

```
pip install deap
```

Полужирный: новые термины и важные слова, а также части пользовательского интерфейса. Так выделяются команды меню и текст в диалоговых окнах, например: «Выберите команду **System info** на панели **Administration**».

Предупреждения и важные замечания выглядят так.

Советы и рекомендации выглядят так.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

ЧАСТЬ I

ОСНОВЫ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ

В этой части мы познакомимся с основными концепциями генетических алгоритмов и способами их применения

Глава 1

Введение в генетические алгоритмы

Позаимствовавший идею у теории эволюции Чарльза Дарвина, один из самых удивительных методов решения задач заслуженно получил название «**эволюционные вычисления**». Самыми известными и широко распространенными представителями этого семейства являются **генетические алгоритмы**. В этой главе мы начнем путешествие в мир этих чрезвычайно мощных и вместе с тем очень простых методов.

Мы познакомимся с **генетическими алгоритмами**, проведем аналогию между ними и дарвиновской эволюцией и опишем как теорию, так и базовые принципы их работы. Затем поговорим о различиях между генетическими и традиционными алгоритмами, раскроем их преимущества и ограничения, опишем области применения. И завершим примерами задач, в которых генетические алгоритмы могут оказаться полезными.

В этой вводной главе рассматриваются следующие вопросы:

- что такое генетические алгоритмы;
- теоретические основы генетических алгоритмов;
- различия между генетическими и традиционными алгоритмами;
- преимущества и ограничения генетических алгоритмов;
- когда имеет смысл использовать генетические алгоритмы.

Что такое генетические алгоритмы?

Генетические алгоритмы – это семейство поисковых алгоритмов, идеи которых подсказаны принципами эволюции в природе. Имитируя процессы естественного отбора и воспроизводства, генетические алгоритмы могут находить высококачественные решения задач, включающих поиск, оптимизацию и обучение. В то же время аналогия с естественным отбором позволяет этим алгоритмам преодолевать некоторые препятствия, встающие на пути тради-

ционных алгоритмов поиска и оптимизации, особенно в задачах с большим числом параметров и сложными математическими представлениями.

В этой части книги мы рассмотрим основные идеи генетических алгоритмов и их аналогию с эволюционными процессами в природе.

Дарвиновская эволюция

Генетические алгоритмы реализуют упрощенный вариант дарвиновской эволюции. Ниже перечислены принципы эволюционной теории Дарвина.

- **Изменчивость.** Признаки (атрибуты) отдельных особей, входящих в состав популяции, могут изменяться. Поэтому особи отличаются друг от друга, например по внешнему виду или поведению.
- **Наследственность.** Некоторые свойства устойчиво передаются от особи к ее потомкам. Поэтому потомки похожи на своих родителей больше, чем на других особей, не связанных с ними родством.
- **Естественный отбор.** Обычно популяции борются за ресурсы, имеющиеся в окружающей их среде. Особи, обладающие свойствами, лучше приспособленными к окружающей среде, более успешны в борьбе за выживание и привносят больше потомков в следующее поколение.

Иными словами, эволюция сохраняет популяцию особей, отличающихся друг от друга. Те, кто лучше приспособлен к окружающей среде, имеют больше шансов на выживание, размножение и передачу своих признаков следующему поколению. Так популяция от поколения к поколению становится все более приспособленной к окружающей среде и встающим на ее пути трудностям.

Важным механизмом эволюции является **скрещивание**, или **рекомбинация**, – когда потомок приобретает комбинацию признаков своих родителей. Скрещивание помогает поддерживать разнообразие популяции и со временем закреплять лучшие признаки. Кроме того, важную роль в эволюции играют **мутации** – случайные вариации признаков, – поскольку они вносят изменения, благодаря которым популяция время от времени совершает скачок в развитии.

Аналогия с генетическими алгоритмами

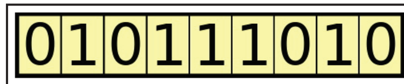
Цель генетических алгоритмов – найти оптимальное решение некоторой задачи. Если дарвиновская эволюция развивает популяцию отдельных особей, то генетические алгоритмы развивают популяцию потенциальных решений данной задачи, называемых **индивидуумами**. Эти решения итеративно оцениваются и используются для создания нового поколения решений. Те, что лучше проявили себя при решении задачи, имеют больше шансов пройти отбор и передать свои качества следующему поколению. Так постепенно потенциальные решения совершенствуются в решении поставленной задачи.

В следующих разделах описываются различные компоненты генетических алгоритмов, поддерживающие эту аналогию с дарвиновской эволюцией.

Генотип

В природе скрещивание, воспроизводство и мутация реализуются посредством **генотипа** – набора генов, сгруппированных в хромосомы. Когда две особи скрещиваются и производят потомство, каждая хромосома потомка несет комбинацию генов родителей.

В случае генетических алгоритмов каждому индивидууму соответствует хромосома, представляющая набор генов. Например, хромосому можно представить двоичной строкой, в которой каждый бит соответствует одному **гену**:

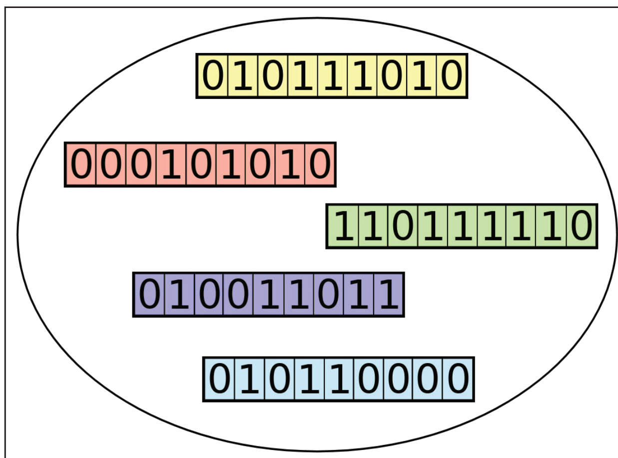


Простое двоичное кодирование хромосомы

На рисунке выше показан пример такой двоично-кодированной хромосомы, представляющей одного индивидуума.

Популяция

В любой момент времени генетический алгоритм хранит популяцию **индивидуумов** – набор потенциальных решений поставленной задачи. Поскольку каждый индивидуум представлен некоторой хромосомой, эту популяцию можно рассматривать как коллекцию хромосом:



Популяция индивидуумов,
представленных двоично-кодированными хромосомами

Популяция всегда представляет текущее поколение и эволюционирует со временем, когда текущее поколение заменяется новым.

Функция приспособленности

На каждой итерации алгоритма индивидуумы оцениваются с помощью **функции приспособленности** (или **целевой функции**). Это функция, которую мы стремимся оптимизировать, или задача, которую пытаемся решить.

Индивидуумы, для которых функция приспособленности дает наилучшую оценку, представляют лучшие решения и с большей вероятностью будут отобраны для воспроизводства и представлены в следующем поколении. Со временем качество решений повышается, значения функции приспособленности растут, а когда будет найдено удовлетворительное значение, процесс можно остановить.

Отбор

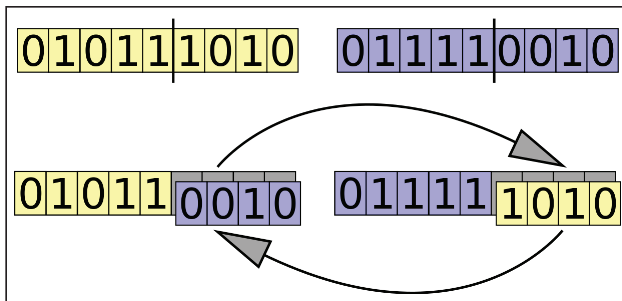
После того как вычислены приспособленности всех индивидуумов в популяции, начинается процесс отбора, который определяет, какие индивидуумы будут оставлены для воспроизводства, т. е. создания потомков, образующих следующее поколение.

Процесс отбора основан на оценке приспособленности индивидуумов. Те, чья оценка выше, имеют больше шансов передать свой генетический материал следующему поколению.

Плохо приспособленные индивидуумы все равно могут быть отобраны, но с меньшей вероятностью. Таким образом, их генетический материал не полностью исключен.

Скрещивание

Для создания пары новых индивидуумов родители обычно выбираются из текущего поколения, а части их хромосом меняются местами (скрещиваются), в результате чего создаются две новые хромосомы, представляющие потомков. Эта операция называется скрещиванием, или рекомбинацией.



Операция скрещивания двух двоично-кодированных хромосом

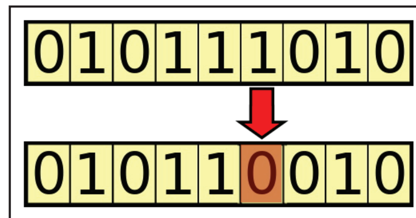
Источник: https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.One.Point.svg.Image, автор Yearofthedragon. Публикуется по лицензии Creative Commons CC BY-SA 3.0: https://creativecommons.org/licenses/by-sa/3.0/deed.en

На рисунке выше показана операция скрещивания с созданием двух потомков родителей.

Мутация

Цель оператора мутации – периодически случайным образом **обновлять** популяцию, т. е. вносить новые сочетания генов в хромосомы, стимулируя тем самым поиск в неисследованных областях пространства решений.

Мутация может проявляться как случайное изменение гена. Мутации реализуются с помощью внесения случайных изменений в значения хромосом, например инвертирования одного бита в двоичной строке.



Применение оператора мутации
к двоично-кодированной хромосоме

На рисунке выше приведен пример операции мутации.

Далее мы рассмотрим теорию, стоящую за генетическими алгоритмами.

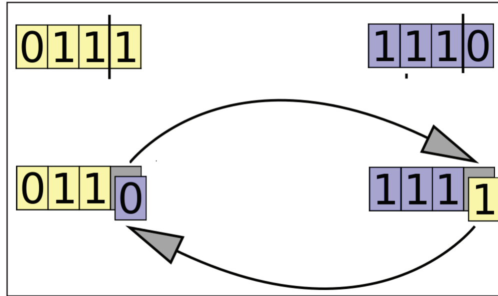
Теоретические основы генетических алгоритмов

Гипотеза структурных элементов, лежащая в основе генетических алгоритмов, заключается в том, что оптимальное решение задачи может быть собрано из небольших структурных элементов, и чем их больше, тем ближе мы подходим к оптимальному решению.

Индивидуумам, которые содержат некоторые из желательных структурных элементов, назначается более высокая оценка. Повторные операции отбора и скрещивания приводят к появлению все лучших индивидуумов, передающих эти структурные элементы следующему поколению, возможно, в сочетании с другими успешными структурными элементами. Тем самым создается генетическое давление, направляющее популяцию в сторону появления все большего числа индивидуумов, обладающих структурными элементами, образующими оптимальное решение.

В результате каждое поколение оказывается лучше предыдущего и содержит больше индивидуумов, близких к оптимальному решению.

Например, если имеется популяция четырехзначных двоичных строк и требуется найти строки с максимальной суммой цифр, то цифра 1 в любой из четырех позиций является хорошим структурным элементом. В процессе работы алгоритм будет определять решения, содержащие такие структурные элементы, и объединять их. В каждом поколении будет больше индивидуумов, содержащих 1 в различных позициях, а в конечном итоге получится строка 1111, объединяющая все четыре желательных структурных элемента. Это показано на рисунке ниже.



Демонстрация операции скрещивания –
структурные элементы оптимального решения собираются вместе

Здесь мы видим, как два индивидуума, представляющих хорошие решения задачи (в каждом три бита равны 1), порождают потомка, дающего наилучшее решение (четыре единичных бита), после того как операция скрещивания объединяет желательные элементы обоих родителей.

Теорема о схемах

Формальное выражение гипотезы о структурных элементах является содержанием **теоремы Холланда о схемах**, или **основной теоремы генетических алгоритмов**.

Эта теорема говорит о схемах, т. е. паттернах (или закономерностях), обнаруживаемых в хромосомах. Каждая схема представляет подмножество хромосом, обладающих определенным сходством.

Например, если набор хромосом представлен двоичными строками длины 4, то схема *1*01* представляет все хромосомы, у которых в крайней левой позиции находится 1, в двух крайних справа – 01, а во второй слева позиции может находиться как 1, так и 0, поскольку * – это **метасимвол**, сопоставляемый с любым конкретным символом.

Для каждой схемы можно определить две характеристики:

- **порядок**: количество фиксированных цифр (не метасимволов);
- **определяющая длина**: расстояние между крайними фиксированными цифрами.

В таблице ниже приведены примеры четырехзначных двоичных схем и их характеристик:

Схема	Порядок	Определяющая длина
1101	4	3
1*01	3	3
*101	3	2
*1*1	2	2
**01	2	1
1***	1	0
****	0	0

Каждая хромосома в популяции соответствует нескольким схемам – точно так же, как заданная строка соответствует разным регулярным выражениям. Например, хромосома 1101 соответствует любой из перечисленных в таблице схем. Если оценка этой хромосомы высока, то она с большей вероятностью успешно пройдет отбор вместе со всеми схемами, которые представляет. После скрещивания с другой хромосомой или мутации некоторые схемы выживут, а остальные исчезнут. Больше шансов на выживание у схем низкого порядка с малым определяющим расстоянием.

Теорема о схемах утверждает, что частота схем низкого порядка с малым определяющим расстоянием и приспособленностью выше средней экспоненциально возрастает в последующих поколениях. Иными словами, генетический алгоритм увеличивает частоту в популяции небольших и простых структурных элементов, представляющих атрибуты, благодаря которым решение становится лучше.

Отличия от традиционных алгоритмов

Между генетическими и традиционными алгоритмами поиска и оптимизации имеется несколько важных различий.

Перечислим основные характеристики генетических алгоритмов, отличающие их от традиционных:

- поддержание популяции решений:
- использование генетического представления решений;
- использование функции приспособленности;
- вероятностное поведение.

В следующих разделах эти факторы описываются более подробно.

Популяция как основа алгоритма

Целью генетического поиска является популяция потенциальных решений (индивидуумов), а не единственное решение. В любой точке поиска алгоритм сохраняет множество индивидуумов, образующих текущее поколение. На каждой итерации генетического алгоритма создается следующее поколение индивидуумов.

С другой стороны, в большинстве других алгоритмов поиска хранится единственное решение, которое итеративно улучшается. Например, алгоритм **градиентного спуска** итеративно сдвигает текущее решение в направлении наискорейшего спуска, которое определяется антиградиентом заданной функции.

Генетическое представление

Генетические алгоритмы работают не с самими потенциальными решениями, а с их кодированными представлениями, которые часто называют **хромосомами**. Простым примером хромосомы является двоичная строка фиксированной длины.

Хромосомы позволяют определить генетические операции скрещивания и мутации. Скрещивание реализуется обменом частей родительских хромосом, а мутация – изменением частей хромосом.

Побочный эффект генетического представления – отделение поиска от исходной предметной области. Генетические алгоритмы не знают, что именно представляют хромосомы, и не пытаются их интерпретировать.

Функция приспособленности

Функция приспособленности представляет проблему, которую мы пытаемся решить. Цель генетического алгоритма – найти индивидуумов, для которых оценка, вычисляемая функцией приспособленности, максимальна.

В отличие от традиционных алгоритмов поиска, генетические алгоритмы анализируют только значение, возвращенное функцией приспособленности, их не интересует ни производная, ни какая-либо другая информация. Поэтому они могут работать с функциями, которые трудно или невозможно продифференцировать.

Вероятностное поведение

Многие традиционные алгоритмы по природе своей детерминированы, тогда как правила, применяемые генетическими алгоритмами для перехода от предыдущего поколения к следующему, вероятностные.

Например, вероятность отбора индивидуума для создания следующего поколения тем выше, чем больше значение функции приспособленности, но элемент случайности все равно присутствует. Слабо приспособленные индивидуумы могут быть отобраны, хотя вероятность этого ниже.

Мутации тоже имеют вероятностный характер, обычно их вероятность мала, а изменению подвергаются случайные позиции в хромосоме.

Случайность присутствует и в операторе скрещивания. В некоторых генетических алгоритмах скрещивание происходит лишь с некоторой вероятностью. Если скрещивания не было, то оба родителя дублируются в следующем поколении вообще без изменений.

Несмотря на вероятностную природу процесса, поиск, основанный на генетическом алгоритме, нельзя назвать случайным; случайность используется, чтобы направить поиск в сторону тех областей пространства поиска, где выше шансы улучшить результаты. Теперь рассмотрим преимущества генетических алгоритмов.

Преимущества генетических алгоритмов

Особенности генетических алгоритмов, рассмотренные в предыдущих разделах, определяют их преимущества по сравнению с традиционными алгоритмами поиска.

Ниже перечислены основные преимущества генетических алгоритмов:

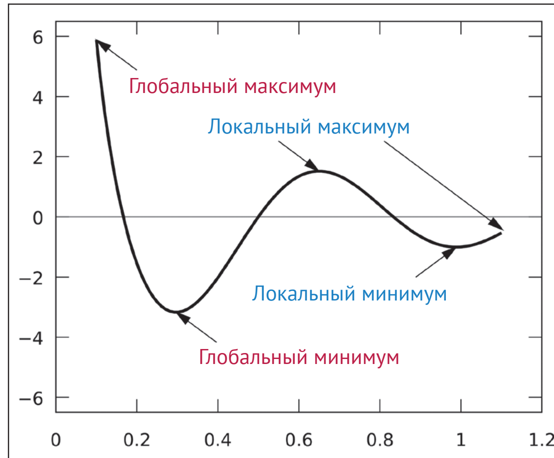
- способность выполнять глобальную оптимизацию;
- применимость к задачам со сложным математическим представлением;
- применимость к задачам, не имеющим математического представления;
- устойчивость к шуму;
- поддержка распараллеливания и распределенной обработки;
- пригодность к непрерывному обучению.

Мы рассмотрим их поочередно в следующих разделах.

Глобальная оптимизация

Во многих задачах оптимизации имеются точки локального максимума и минимума, которые представляют решения, лучшие, чем те, что находятся поблизости, но необязательно лучшие в глобальном смысле.

На рисунке ниже показаны различия между локальными и глобальными минимумами и максимумами.



Локальные и глобальные максимумы и минимумы функции

Источник: https://commons.wikimedia.org/wiki/File:Computational.science.
Genetic.algorithm.Crossover.One.Point.svg. Автор KSmrq.
Публикуется по лицензии Creative Commons CC BY-SA 3.0:
https://creativecommons.org/licenses/by-sa/3.0/

Большинство традиционных алгоритмов поиска и оптимизации, а особенно те, что основаны на вычислении градиента, могут застревать в локальном максимуме, вместо того чтобы найти глобальный. Это связано с тем, что в окрестности локального максимума всякое небольшое изменение решения ухудшает оценку.

Генетические алгоритмы менее подвержены этой напасти и имеют больше шансов отыскать глобальный максимум. Объясняется это тем, что используется популяция потенциальных решений, а не единственное решение, а операции скрещивания и мутации зачастую порождают решения, далеко отстоящие от ранее рассмотренных. Это остается справедливым при условии, что мы поддерживаем разнообразие популяции и избегаем **преждевременной сходимости**, о чем поговорим в следующем разделе.

Применимость к сложным задачам

Поскольку генетическим алгоритмам нужно знать только значение функции приспособленности каждого индивидуума, а все остальные ее свойства, в частности производные, несущественны, их можно применять к задачам со сложным математическим представлением, включающим функции, которые трудно или невозможно продифференцировать.

К сложным случаям, когда достоинства генетических алгоритмов раскрываются во всем блеске, относятся также задачи с большим числом параметров или со смешанными параметрами, например непрерывными и дискретными.

Применимость к задачам, не имеющим математического представления

Генетические алгоритмы применимы и к задачам, вообще не имеющим математического представления. Один из таких случаев, представляющий особый интерес, – когда оценка приспособленности основана на мнении человека. Пусть, например, требуется найти наиболее привлекательную цветовую палитру для веб-сайта. Мы можем попробовать разные комбинации цветов и попросить пользователей оценить привлекательность сайта. А затем применить генетический алгоритм, чтобы найти лучшую комбинацию, используя функцию приспособленности, основанную на оценках пользователей. Алгоритм будет работать, несмотря на то что никакого математического представления нет и невозможно вычислить оценку заданной комбинации непосредственно.

В следующей главе мы увидим, что генетические алгоритмы справляются даже со случаями, когда нельзя получить оценку каждого индивидуума, при условии что имеется возможность сравнить двух индивидуумов и определить, какой из них лучше. Примером может служить алгоритм машинного обучения, который управляет автомобилем в имитации гонок. Поиск на основе генетического алгоритма может оптимизировать и настроить алгоритм машинного обучения, организовав состязание различных вариантов алгоритма между собой, чтобы определить, какой лучше.

Устойчивость к шуму

Для некоторых задач характерно присутствие **шума**. Это означает, что даже при близких истинных значениях входных параметров результаты их измерений могут довольно сильно различаться. Например, так бывает, когда данные считываются с датчиков или когда оценка основана на мнении человека, как было описано в предыдущем разделе.

Подобное поведение может сделать непригодными многие традиционные алгоритмы поиска, но генетические алгоритмы в общем случае устойчивы к нему благодаря повторяющимся операциям сборки и оценивания индивидуумов.

Распараллеливание

Генетические алгоритмы хорошо поддаются распараллеливанию и распределенной обработке. Функция приспособленности независимо вычисляется для каждого индивидуума, а это значит, что все индивидуумы в популяции могут обрабатываться одновременно.

Кроме того, операции отбора, скрещивания и мутации могут одновременно выполняться для индивидуумов и пар индивидуумов.

Поэтому подход, основанный на генетических алгоритмах, естественно адаптируется к распределенным и облачным реализациям.

Непрерывное обучение

В природе эволюция никогда не прекращается. Если окружающие условия изменяются, популяция приспосабливается к ним. Так и генетические алгоритмы могут непрерывно работать в постоянно изменяющихся условиях, и мы всегда можем получить и использовать лучшее на данный момент решение.

Но это возможно, только если окружающая среда изменяется медленно по сравнению со скоростью смены поколений в генетическом алгоритме. Итак, преимущества мы обсудили, теперь перейдем к ограничениям генетических алгоритмов.

Ограничения генетических алгоритмов

Чтобы получить максимум пользы от генетических алгоритмов, мы должны знать об их ограничениях и потенциальных подвохах.

Ниже перечислены ограничения генетических алгоритмов:

- необходимы специальные определения;
- необходима настройка гиперпараметров;
- большой объем счетных операций;
- опасность преждевременной сходимости;
- отсутствие гарантированного решения.

Специальные определения

Пытаясь применить генетические алгоритмы к некоторой задаче, мы должны создать подходящее представление – определить функцию приспособленности и структуру хромосом, а также операторы отбора, скрещивания и мутации. Зачастую это совсем не просто и занимает много времени.

По счастью, генетические алгоритмы уже бессчетное число раз применялись к самым разным задачам, так что многие определения стандартизованы. В этой книге рассматриваются многочисленные примеры реальных задач и способы их решения с помощью генетических алгоритмов. Можете использовать ее как руководство, когда столкнетесь с новой задачей.

Настройка гиперпараметров

Поведение генетических алгоритмов контролируется набором гиперпараметров, например размером популяции и скоростью мутации. Точных правил для выбора значений гиперпараметров не существует.

Однако так обстоит дело практически со всеми алгоритмами поиска и оптимизации. Проработав примеры в книге и поэкспериментировав самостоятельно, вы научитесь подбирать разумные значения гиперпараметров.

Большой объем счетных операций

Работа с потенциально большими популяциями и итеративный характер генетических алгоритмов обусловливают большой объем вычислений, поэтому на получение приемлемого результата может уйти много времени.

Проблему можно сгладить за счет хорошего выбора гиперпараметров, распараллеливания и в некоторых случаях кеширования промежуточных результатов.

Преждевременная сходимость

Если приспособленность какого-то индивидуума гораздо больше, чем у всей остальной популяции, то не исключено, что он продублируется так много раз, что в конечном счете, кроме него, в популяции ничего не останется. В результате генетический алгоритм может застрять в локальном максимуме и не найдет глобального.

Чтобы предотвратить такое развитие событий, важно поддерживать разнообразие популяции. В следующей главе мы рассмотрим различные способы достижения этой цели.

Отсутствие гарантированного решения

Использование генетических алгоритмов не гарантирует нахождения глобального максимума.

Однако это типично для всех алгоритмов поиска и оптимизации, если только у задачи не существует аналитического решения.

Но, вообще говоря, при правильном применении генетические алгоритмы находят хорошие решения на разумное время. Далее мы рассмотрим несколько ситуаций, в которых применение генетических алгоритмов оправдано.

Сценарии применения генетических алгоритмов

Резюмируя изложенное в предыдущих разделах, можно сказать, что генетические алгоритмы лучше применять для решения следующих задач.

- **Задачи со сложным математическим представлением.** Поскольку генетическим алгоритмам нужно знать только значение функции приспособленности, их можно использовать для решения задач, в которых целевую функцию трудно или невозможно продифференцировать, задач с большим количеством параметров и задач с параметрами разных типов.
- **Задачи, не имеющие математического представления.** Генетические алгоритмы не требуют математического представления задачи,

коль скоро можно получить значение оценки или существует метод сравнения двух решений.

- **Задачи с зашумленной окружающей средой.** Генетические алгоритмы устойчивы к зашумленным данным, например прочитанным с датчика или основанным на оценках, сделанных человеком.
- **Задачи, в которых окружающая среда изменяется во времени.** Генетические алгоритмы могут адаптироваться к медленным изменениям окружающей среды, поскольку постоянно создают новые поколения, приспосабливающиеся к изменениям.

С другой стороны, если для задачи известен специализированный способ решения традиционным или аналитическим методом, то вполне вероятно, что он окажется эффективнее.

Резюме

Мы начали эту главу со знакомства с генетическими алгоритмами, с аналогии между ними и дарвиновской эволюцией и с основных принципов их работы: использования популяции, генотипа, функции приспособленности и операторов отбора, скрещивания и мутации.

Затем рассмотрели теорию, лежащую в основе генетических алгоритмов, – гипотезу структурных элементов и теорему о схемах – и показали, как генетические алгоритмы работают, собирая наилучшие решения из небольших структурных элементов, обладающих превосходными качествами.

Далее мы разобрали различия между генетическими и традиционными алгоритмами, в т. ч. поддержание популяции решений и использование генетического представления решений.

После этого мы обсудили сильные стороны генетических алгоритмов, включая возможность глобальной оптимизации, применимость к задачам со сложным математическим представлением или вообще без такового и устойчивость к шуму. Мы также поговорили о недостатках: необходимости специальных определений и настройки гиперпараметров, опасности преждевременной сходимости.

В заключение перечислили ситуации, когда применение генетических алгоритмов может дать выигрыш: математически сложные задачи, оптимизация в зашумленной или постоянно изменяющейся среде.

В главе 2 мы более подробно рассмотрим ключевые компоненты и детали реализации генетических алгоритмов. Это станет подготовкой к последующим главам, где мы приведем код решения различных задач.

Для дальнейшего чтения

Дополнительный материал, относящийся к этой главе, можно найти в главе «Введение в генетические алгоритмы» книги Amita Kapoor «Hands-On Artificial Intelligence for IoT» (январь 2019), опубликованной по адресу https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788836067.

Глава 2

Основные компоненты генетических алгоритмов

В этой главе мы подробнее рассмотрим основные компоненты и детали реализации генетических алгоритмов. Это станет подготовкой к чтению последующих глав, посвященных использованию таких алгоритмов для решения различных задач.

Прежде всего опишем базовую структуру генетического алгоритма, а затем выделим в ней отдельные компоненты и продемонстрируем различные реализации методов отбора, скрещивания и мутации. Затем мы познакомимся с генетическими алгоритмами с вещественными хромосомами для организации поиска в непрерывном пространстве параметров. За этим последует увлекательное введение в вопросы элитизма, образования ниш и разделения в генетических алгоритмах. Наконец, мы займемся искусством решения задач с помощью генетических алгоритмов.

Прочитав эту главу, вы будете:

- знать об основных компонентах генетических алгоритмов;
- понимать, из каких этапов состоит поток управления в генетическом алгоритме;
- знать, что такое генетические операторы и какие у них есть варианты;
- знать о различных видах условий остановки;
- понимать, как нужно модифицировать базовый генетический алгоритм, чтобы он мог работать с вещественными числами;
- понимать механизм элитизма;
- понимать идею и реализацию механизмов образования ниш и разделения;
- знать, с чего начинать при решении новой задачи.

Базовая структура генетического алгоритма

На следующей блок-схеме показаны основные этапы типичного генетического алгоритма.

Базовая структура генетического алгоритма

В следующих разделах эти этапы описаны более подробно.

Создание начальной популяции

Начальная популяция состоит из случайным образом выбранных потенциальных решений (индивидуумов). Поскольку в генетических алгоритмах индивидуумы представлены хромосомами, начальная популяция – это, по сути дела, набор хромосом. Формат хромосом должен соответствовать принятым для решаемой задачи правилам, например это могут быть двоичные строки определенной длины.

Вычисление приспособленности

Для каждого индивидуума вычисляется функция приспособленности. Это делается один раз для начальной популяции, а затем для каждого нового поколения после применения операторов отбора, скрещивания и мутации. Поскольку приспособленность любого индивидуума не зависит от всех остальных, эти вычисления можно производить параллельно.

Так как на этапе отбора, следующем за вычислением приспособленности, более приспособленные индивидуумы обычно считаются лучшими решениями, генетические алгоритмы естественно «заточены» под нахождение максимумов функции приспособленности. Если в какой-то задаче нужен минимум, то при вычислении приспособленности следует инвертировать найденное значение, например умножив его на –1.

Применение операторов отбора, скрещивания и мутации

Применение генетических операторов к популяции приводит к созданию новой популяции, основанной на лучших индивидуумах из текущей.

Оператор **отбора** отвечает за отбор индивидуумов из текущей популяции таким образом, что предпочтение отдается лучшим. Примеры операторов отбора приведены в разделе «Методы отбора».

Оператор **скрещивания** (или рекомбинации) создает потомка выбранных индивидуумов. Обычно для этого берутся два индивидуума, и части их хромосом меняются местами, в результате чего создаются две новые хромосомы, представляющие двух потомков. Примеры операторов скрещивания приведены в разделе «Методы скрещивания».

Оператор **мутации** вносит случайные изменения в один или несколько генов хромосомы вновь созданного индивидуума. Обычно вероятность мутации очень мала. Примеры операторов мутации приведены в разделе «Методы мутации».

Проверка условий остановки

Может существовать несколько условий, при выполнении которых процесс останавливается. Сначала отметим два самых распространенных:

- достигнуто максимальное количество поколений. Это условие заодно позволяет ограничить время работы алгоритма и потребление им ресурсов системы;
- на протяжении нескольких последних поколений не наблюдается заметных улучшений. Это можно реализовать путем запоминания наилучшей приспособленности, достигнутой в каждом поколении, и сравнения наилучшего текущего значения со значениями в нескольких предыдущих поколениях. Если разница меньше заранее заданного порога, то алгоритм можно останавливать.

Перечислим также другие возможные условия:

- с момента начала прошло заранее определенное время;
- превышен некоторый лимит затрат, например процессорного времени или памяти;
- наилучшее решение заняло часть популяции, большую заранее заданного порога.

Резюмируем. Генетический алгоритм начинается с популяции случайно выбранных потенциальных решений (индивидуумов), для которых вычисляется функция приспособленности. Алгоритм выполняет цикл, в котором последовательно применяются операторы отбора, скрещивания и мутации, после чего приспособленность индивидуумов пересчитывается. Цикл продолжается, пока не выполнено условие остановки, после чего лучший индивидуум в текущей популяции считается решением. Теперь рассмотрим методы отбора.

Методы отбора

Отбор выполняется в начале каждой итерации цикла генетического алгоритма, чтобы выбрать из текущей популяции тех индивидуумов, которые станут родителями индивидуумов в следующем поколении. Отбор носит вероятностный характер, причем вероятность выбора индивидуума зависит от его приспособленности, так что у более приспособленных индивидуумов шансы отобраться выше.

Ниже описаны некоторые из распространенных методов отбора и их характеристики.

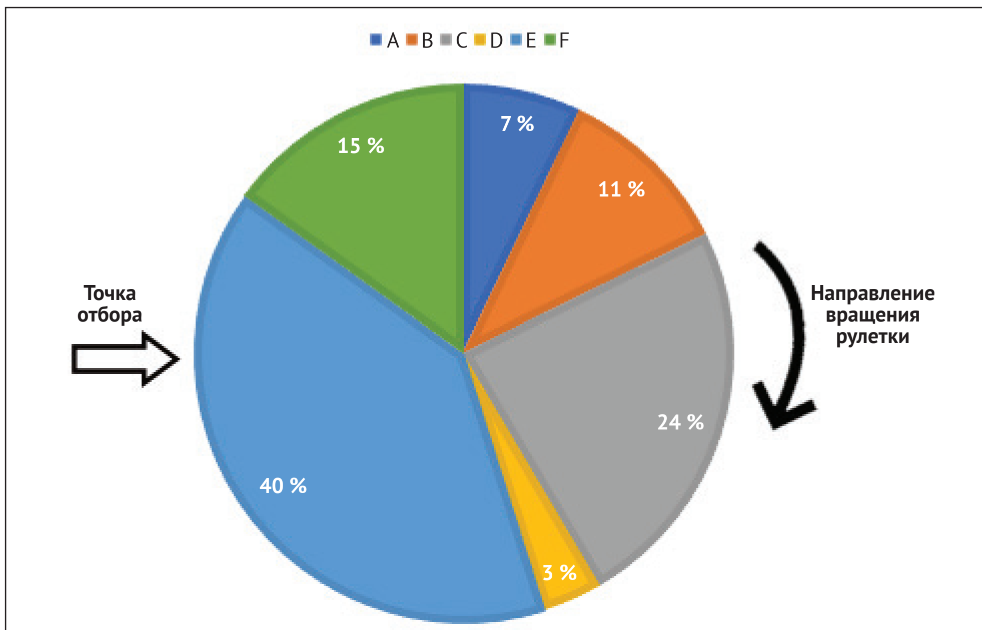
Правило рулетки

Метод отбора по правилу рулетки, или **отбор пропорционально приспособленности** (fitness proportionate selection – **FPS**), устроен так, что вероятность отбора индивидуума прямо пропорциональна его приспособленности. Тут можно провести аналогию с вращением колеса рулетки, где каждому индивидууму соответствует сектор, стоимость которого равна приспособленности индивидуума. Шансы, что шарик остановится в секторе индивидуума, пропорциональны размеру этого сектора.

Пусть, например, имеется популяция из шести индивидуумов с такими значениями приспособленности, как в таблице ниже. По этим значениям вычисляются доли, занимаемые секторами каждого индивидуума.

Индивидуум	Приспособленность	Относительная доля
A	8	7 %
B	12	11 %
C	27	24 %
D	4	3 %
E	45	40 %
F	17	15 %

На рисунке ниже изображена соответствующая рулетка.

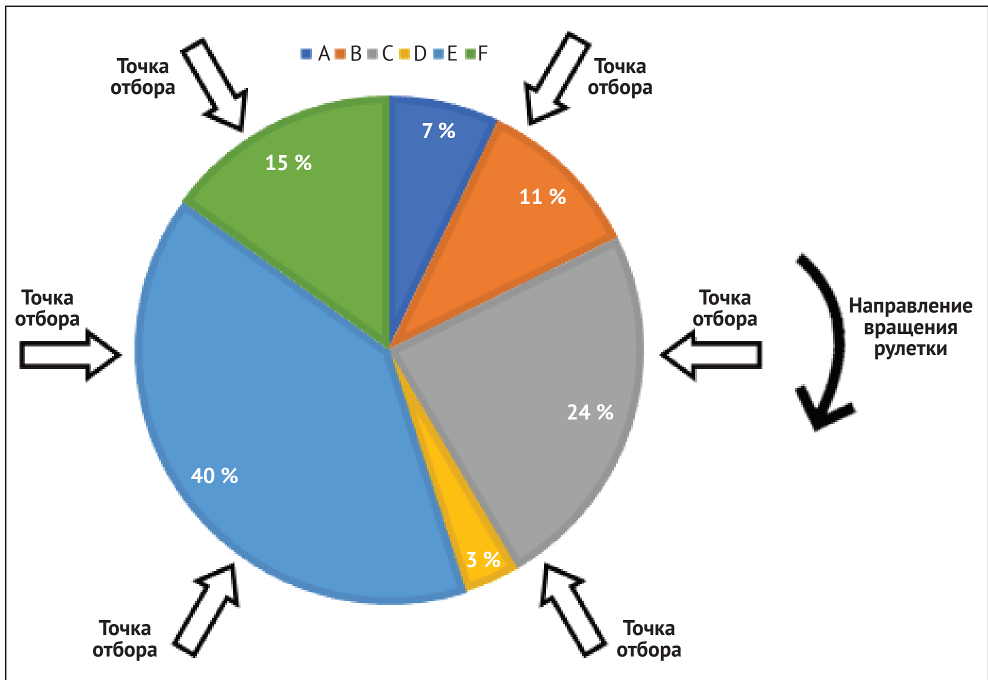


Пример отбора по правилу рулетки

После каждого запуска рулетки отбор индивидуума из популяции производится в точке отбора. Затем рулетка запускается еще раз для выбора следующего индивидуума, и так до тех пор, пока не наберется достаточно индивидуумов для образования следующего поколения. В результате один и тот же индивидуум может быть выбран несколько раз.

Стохастическая универсальная выборка

Стохастическая универсальная выборка (stochastic universal sampling – SUS) – немного модифицированный вариант правила рулетки. Используется та же рулетка с такими же секторами, но вместо одной точки отбора и многократного запуска рулетки мы вращаем колесо только один раз, а отбор индивидуумов производим в нескольких точках, равномерно расставленных по окружности. Тем самым все индивидуумы выбираются одновременно, как показано на рисунке ниже.



Пример стохастической универсальной выборки

Этот метод отбора не дает индивидуумам с особенно высокой приспособленностью заполнить все следующее поколение в результате повторного выбора. Поэтому более слабым индивидуумам предоставляется шанс, а несправедливость чистого правила рулетки в какой-то мере сглаживается.

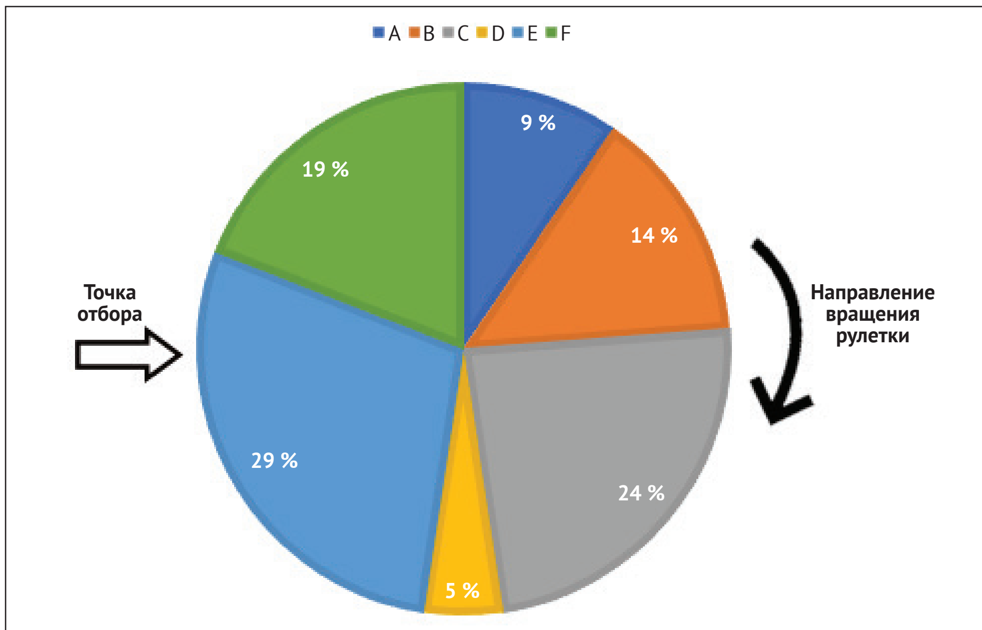
Ранжированный отбор

Метод ранжированного отбора похож на правило рулетки, но значения приспособленности используются не для вычисления вероятностей выбора, а просто для сортировки индивидуумов. После сортировки каждому индивидууму назначается ранг, соответствующий его позиции в списке, а вероятности секторов рулетки вычисляются на основе этих рангов.

Возьмем ту же самую популяцию из шести индивидуумов, что и раньше. И добавим в таблицу столбец с рангом индивидуума. Поскольку размер популяции равен 6, наивысший возможный ранг тоже равен 6, следующий по порядку – 5 и т. д. Каждому индивидууму сопоставляется сектор рулетки, вычисленный по этим рангам, а не по значениям функции приспособленности.

Индивидуум	Приспособленность	Ранг	Относительная доля
A	8	2	9 %
B	12	3	14 %
C	27	5	24 %
D	4	1	5 %
E	45	6	29 %
F	17	4	19 %

На рисунке ниже изображена соответствующая рулетка.



Пример ранжированного отбора

Ранжированный отбор полезен, когда есть несколько индивидуумов, гораздо лучше приспособленных, чем все остальные. Использование ранга вместо самой приспособленности мешает этим индивидуумам захватить всю популяцию в следующем поколении, поскольку ранжирование сглаживает значительные различия.

Кроме того, когда все индивидуумы обладают почти одинаковой приспособленностью, ранжированный отбор позволяет разделить их, отдавая преимущество лучшим, даже когда различия малы.

Масштабирование приспособленности

Если при ранжированном отборе приспособленности заменяются рангами, то в случае масштабирования приспособленности к исходным значениям приспособленности применяется линейное преобразование, которое переводит их в желательный диапазон:

масштабированная приспособленность = $a \times$ (*исходная приспособленность*) + b,

где a и b – постоянные, выбираемые по нашему усмотрению.

Например, в предыдущих примерах значения приспособленности лежали в диапазоне от 4 (у индивидуума D) до 45 (у индивидуума E). Допустим, мы хотим перевести их в диапазон от 50 до 100. Тогда a и b вычисляются из уравнений

- $50 = a \times 4 + b$ (наименьшая приспособленность);
- $100 = a \times 45 + b$ (наибольшая приспособленность).

Решая эту систему двух линейных уравнений, получаем следующие параметры масштабирования:

$a = 1.22$, $b = 45.12$.

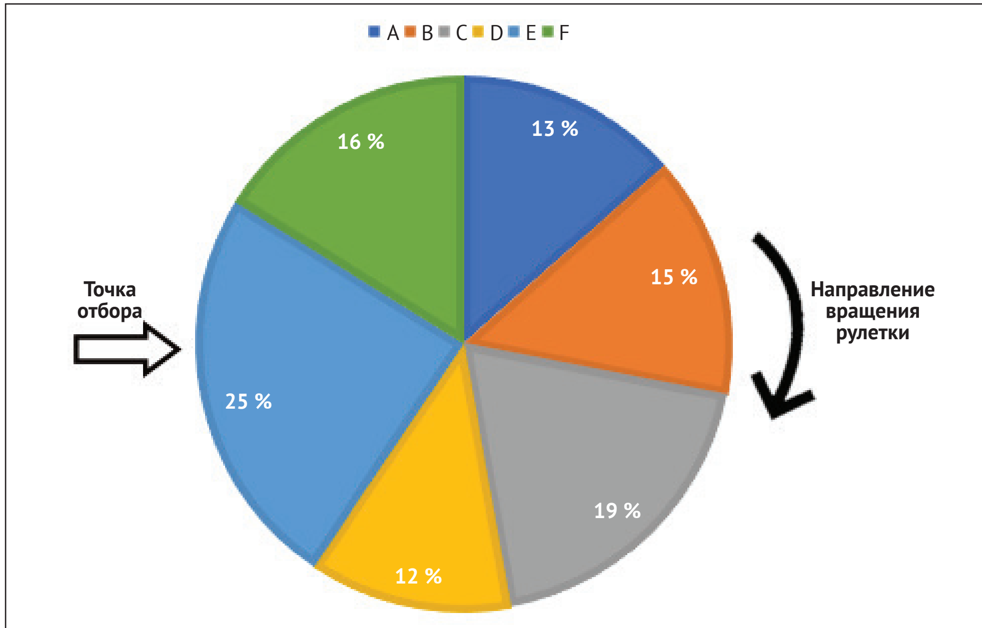
Таким образом, масштабированные значения приспособленности вычисляются по формуле

масштабированная приспособленность = 1.22 × (*исходная приспособленность*) + 45.12.

Добавив в таблицу столбец, содержащий значения масштабированной приспособленности, убеждаемся, что все они действительно попадают в диапазон от 50 до 100.

Индивидуум	Приспособленность	Масштабированная приспособленность	Относительная доля
A	8	55	13 %
B	12	60	15 %
C	27	78	19 %
D	4	50	12 %
E	45	100	25 %
F	17	66	16 %

На рисунке ниже изображена соответствующая рулетка.



Пример колеса рулетки после масштабирования приспособленности

Как видно по рисунку, переход к новому диапазону дает гораздо более равномерное разбиение рулетки на сектора по сравнению с исходным. Вероятность отбора лучшего индивидуума (с масштабированной приспособленностью 100) теперь всего в два раза выше, чем худшего (с масштабированной приспособленностью 50), а не в 11 раз, как первоначально.

Турнирный отбор

В каждом раунде турнирного отбора из популяции выбираются два или более индивидуумов, и тот, у кого приспособленность больше, выигрывает и отбирается в следующее поколение.

Рассмотрим тех же индивидуумов с такими же приспособленностями, что и выше. На следующем рисунке показан результат случайного выбора трех из них (A, B и F) с последующим объявлением F победителем, поскольку у него приспособленность максимальная из трех (17).

Индивидуум	Приспособленность
A	8
B	12
C	27
D	4
E	45
F	17

Пример турнирного отбора на турнире с тремя участниками

Количество индивидуумов, участвующих в каждом раунде турнирного отбора (в нашем примере – три), называется *размером турнира*. Чем больше размер турнира, тем выше шансы, что в раундах будут участвовать лучшие индивидуумы, и тем меньше шансов у слабых участников победить в турнире и отобраться.

У этого метода отбора есть интересная особенность: если мы умеем сравнивать любых двух индивидуумов и определять, какой из них лучше, то сами значения функции приспособленности и не нужны.

Методы скрещивания

Оператор скрещивания (или рекомбинации) соответствует биологическому скрещиванию при половом размножении. Он используется для комбинирования генетической информации двух индивидуумов, выступающих в роли родителей, в процессе порождения потомков (обычно двух).

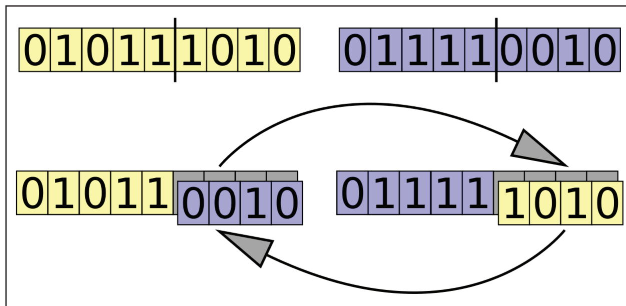
Как правило, оператор скрещивания применяется не всегда, а с некоторой (высокой) вероятностью. Если скрещивание *не* применяется, то копии обоих родителей переходят в следующее поколение без изменения.

В следующих разделах описываются некоторые методы скрещивания и типичные примеры их применения. Впрочем, в некоторых ситуациях приходится придумывать специальный метод, соответствующий конкретной задаче.

Одноточечное скрещивание

В этом случае позиция в хромосомах обоих родителей выбирается случайным образом. Эта позиция называется *точкой скрещивания*, или точкой разреза. Гены одной хромосомы, расположенные справа от этой точки, обмениваются с точно так же расположенными генами другой хромосомы. В результате мы получаем двух потомков, несущих генетическую информацию обоих родителей.

На рисунке ниже показано одноточечное скрещивание пары двоичных хромосом, когда точка скрещивания находится между пятым и шестым геном.



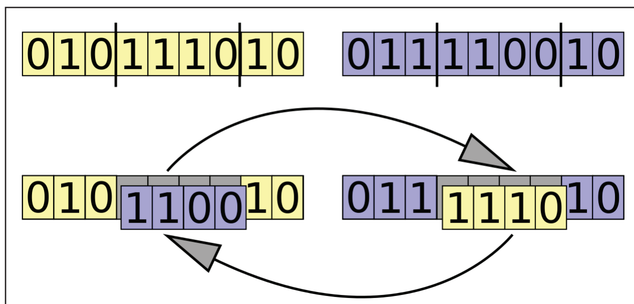
Пример одноточечного скрещивания

Источник: https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.One.Point.svg. Автор: Yearofthedragon.
Публикуется по лицензии Creative Commons CC BY-SA 3.0:
https://creativecommons.org/licenses/by-sa/3.0/deed.en

Двухточечное и *k*-точечное скрещивание

При двухточечном скрещивании случайным образом выбираются по две точки скрещивания в каждой хромосоме. Гены одной хромосомы, расположенные между этими точками, обмениваются с точно так же расположенными генами другой хромосомы.

На рисунке ниже показано двухточечное скрещивание пары двоичных хромосом, когда первая точка скрещивания находится между третьим и четвертым геном, а вторая – между седьмым и восьмым.



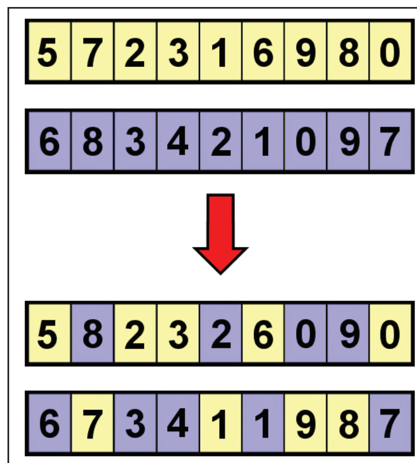
Пример двухточечного скрещивания

Источник: https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.Two.Point.svg. Автор: Yearofthedragon.
Публикуется по лицензии Creative Commons CC BY-SA 3.0:
https://creativecommons.org/licenses/by-sa/3.0/deed.en

Метод двухточечного скрещивания можно реализовать с помощью двух одноточечных скрещиваний с разными точками скрещивания. Его обобщением является метод k-точечного скрещивания, где k – целое положительное число.

Равномерное скрещивание

При равномерном скрещивании каждый ген обоих родителей определяется независимо путем случайного выбора с равномерным распределением. Когда выбирается 50 % генов, оба родителя имеют одинаковые шансы повлиять на потомков.



Пример равномерного скрещивания

Заметим, что в этом примере гены обоих потомков меняются местами, но в принципе потомков можно создавать и независимо.

В этом примере использовались целочисленные хромосомы, но метод точно так же работает для двоичных.

Поскольку в этом методе не производится обмен целых участков хромосом, потенциально он может повысить разнообразие потомков.

Скрещивание для упорядоченных списков

В предыдущем примере мы видели результаты операции скрещивания двух целочисленных хромосом. В обоих родителях каждая цифра от 0 до 9 встречалась ровно один раз, но в потомках некоторые цифры встречаются более одного раза (например, 2 в верхнем потомке и 1 в нижнем), тогда как другие отсутствуют вовсе (например, 4 в верхнем потомке и 5 в нижнем).

Но бывают задачи, в которых целочисленные хромосомы представляют индексы упорядоченного списка. Пусть, например, имеется несколько городов, известны попарные расстояния между ними и требуется найти кратчайший путь, проходящий через все города. Это так называемая задача коммивояжера, которую мы подробно рассмотрим в последующих главах.

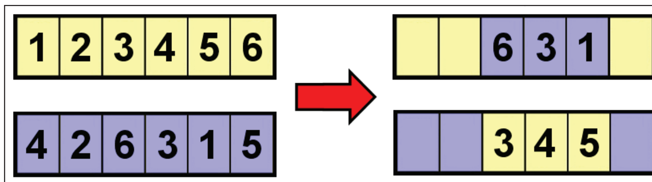
Если городов четыре, то возможное решение задачи удобно представить четырехзначной целочисленной хромосомой, описывающей порядок посещения городов, например (1,2,3,4) или (3,4,2,1). Хромосома, в которой некоторые значения повторяются или, наоборот, отсутствуют, не является допустимым решением.

Для таких случаев были придуманы альтернативные методы скрещивания, гарантирующие, что всегда создается допустимый потомок. Один из таких методов, *упорядоченное скрещивание*, рассматривается в следующем разделе.

Упорядоченное скрещивание

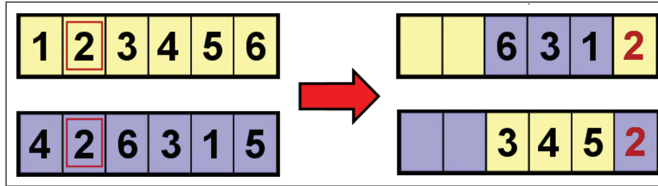
Метод **упорядоченного скрещивания** (**OX1**) стремится по возможности сохранить относительный порядок родительских генов. Мы продемонстрируем его на примере хромосом длины шесть.

На первом шаге мы выполняем двухточечное скрещивание со случайно выбранными точками разреза, как показано на рисунке ниже (родители изображены слева):



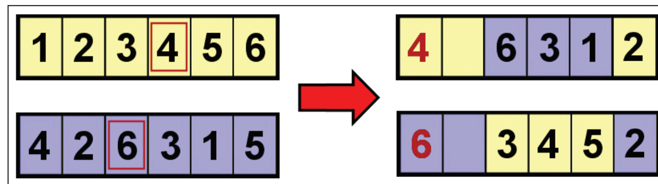
Пример упорядоченного скрещивания – шаг 1

Затем начинаем заполнять оставшиеся гены в каждом потомке, для чего обходим гены родителя в их исходном порядке, начиная со следующего за второй точкой разреза. В первом родителе в этой позиции находится 6, но это число уже присутствует в потомке, поэтому мы переходим к следующей позиции (с возвратом в начало), в которой находится 1. Это число также уже имеется. Следующим является число 2. Поскольку его еще нет в потомке, добавляем его, как показано на рисунке ниже. Для второй пары родитель–потомок мы начинаем с позиции, в которой находится 5. Поскольку это число уже присутствует в потомке, переходим к числу 4. Оно тоже присутствует, поэтому идем дальше и заканчиваем в позиции, где находится число 2, которое мы добавляем в потомка. Результат также показан на рисунке.



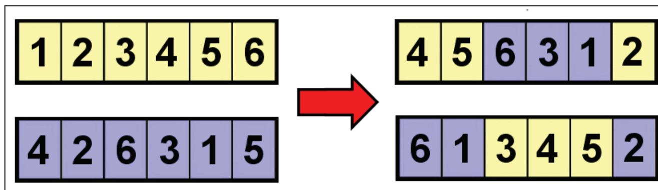
Пример упорядоченного скрещивания – шаг 2

Следующим числом для первого родителя будет 3 (оно уже присутствует в потомке), а за ним 4, которое добавляется к потомку. Для второго родителя следующим идет ген 6. Поскольку его еще нет в потомке, добавляем. Результат показан на рисунке ниже.



Пример упорядоченного скрещивания – шаг 3

Продолжая таким же манером, мы заполним все свободные позиции в потомках и придем к окончательным хромосомам-потомкам:



Пример упорядоченного скрещивания – шаг 4

Существуют и другие методы скрещивания, некоторые из них встретятся нам позже. Но благодаря гибкости генетических алгоритмов вы можете придумать и свои методы. А мы перейдем к методам мутации.

Методы мутации

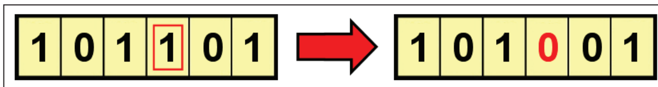
Мутация – последний генетический оператор, применяемый при создании нового поколения. Он применяется к потомку, созданному в результате операций отбора и скрещивания.

Операция мутации вероятностная, обычно она выполняется изредка, с очень низкой вероятностью, поскольку может ухудшить качество индивидуума, к которому применена. В некоторых вариантах генетических алгоритмов вероятность мутации постепенно увеличивается, чтобы предотвратить стагнацию и повысить разнообразие популяции. С другой стороны, если частота мутации слишком велика, то генетический алгоритм выродится в случайный поиск.

В следующих разделах описываются некоторые методы мутации и типичные примеры их применения. Но не забывайте, что вы всегда можете придумать свой собственный метод, отвечающий специфике конкретной задачи.

Инвертирование бита

Для двоичной хромосомы мы можем случайным образом выбрать ген и инвертировать его (взять двоичное дополнение), как показано на рисунке ниже.

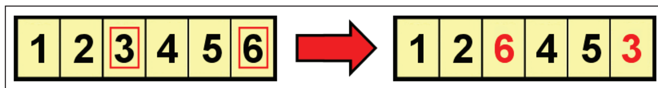


Пример мутации инвертированием бита

Можно пойти дальше и инвертировать не один, а сразу несколько генов.

Мутация обменом

Этот метод применим как к двоичным, так и к целочисленным хромосомам: случайно выбираются два гена, и их значения меняются местами, как показано на рисунке ниже.

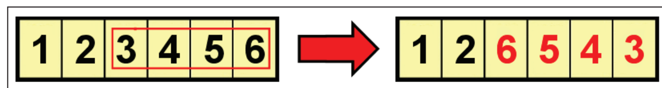


Пример мутации обменом

Такая операция мутации годится для хромосом, представляющих упорядоченные списки, поскольку набор генов в новой хромосоме такой же, как в исходной.

Мутация обращением

При применении этого метода к двоичной или целочисленной хромосоме выбирается случайная последовательность генов, и порядок генов в ней меняется на противоположный, как показано на рисунке ниже.

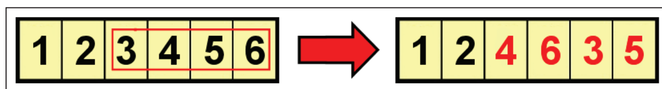


Пример мутации обращением

Как и мутация обменом, мутация обращением годится для хромосом, представляющих упорядоченные списки.

Мутация перетасовкой

Еще один оператор мутации, подходящий для хромосом, представляющих упорядоченные списки, – мутация перетасовкой. В этом случае выбирается случайная последовательность генов, и порядок генов в ней изменяется случайным образом (тасуется), как показано на рисунке ниже.



Пример мутации перетасовкой

В следующем разделе мы рассмотрим другие типы специализированных операторов, встречающиеся в генетических алгоритмах с вещественным кодированием.

Генетические алгоритмы с вещественным кодированием

До сих пор мы рассматривали хромосомы, представляющие двоичные или целочисленные параметры. И генетические операторы были приспособлены для работы с такими хромосомами. Но зачастую встречаются задачи с непрерывным пространством решений, когда индивидуумы описываются вещественными числами с плавающей точкой.

Исторически в генетических алгоритмах применялись двоичные строки для представления целых чисел, но для вещественных чисел это не годится. Точность вещественного числа, представленного двоичной строкой, ограничена длиной строки (количеством бит). Поскольку эту длину нужно задать заранее, может получиться, что строки слишком короткие, поэтому точность недостаточна, или, наоборот, чересчур длинные.

Кроме того, в двоичной строке значение бита зависит от его позиции – наиболее значимые биты находятся слева. Это может стать причиной дисбаланса схем – паттернов, встречающихся в хромосомах; например, схема 1**** (все пятизначные двоичные строки, начинающиеся с 1) и схема ****1

(все пятизначные двоичные строки, заканчивающиеся на 1) имеет порядок 1 и определяющую длину 0, однако первая гораздо более значима, чем вторая.

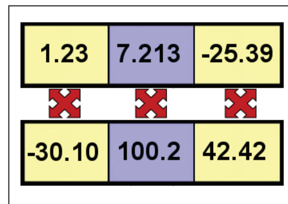
Поэтому вместо двоичных строк используются массивы вещественных чисел – это проще и удобнее. Так, в задаче с тремя вещественными параметрами хромосомы будут иметь вид $[x_1, x_2, x_3]$, где x_1, x_2, x_3 – вещественные числа, например: [1.23, 7.2134, –25.309] или [–30.10, 100.2, 42.424].

Все методы отбора, описанные выше в этой главе, будут работать точно так же для вещественных хромосом, поскольку они зависят только от приспособленности индивидуумов, а не от их представления.

Однако рассмотренные выше методы скрещивания и мутации для вещественных хромосом не годятся, поэтому нужны специальные методы. Важно помнить, что эти операции скрещивания и мутации применяются по отдельности к каждой позиции массивов, представляющих вещественные хромосомы. Например, если родителями в операции скрещивания являются хромосомы [1.23, 7.213, –25.39] и [–30.10, 100.2, 42.42], то скрещивание производится отдельно для следующих пар:

- 1.23 и –30.10 (первая позиция);
- 7.213 и 100.2 (вторая позиция);
- –25.39 и 42.42 (третья позиция).

Это показано на рисунке ниже.



Пример скрещивания вещественных хромосом

Все сказанное относится и к оператору мутации вещественных хромосом.

Ниже описано несколько методов скрещивания и мутации в вещественном случае. А в главе 6 мы увидим, как они применяются на практике.

Скрещивание смешением

В случае **скрещивания смешением** (blend crossover – **BLX**) каждый потомок случайным образом выбирается из следующего интервала, созданного родителями $parent_1$ и $parent_2$:

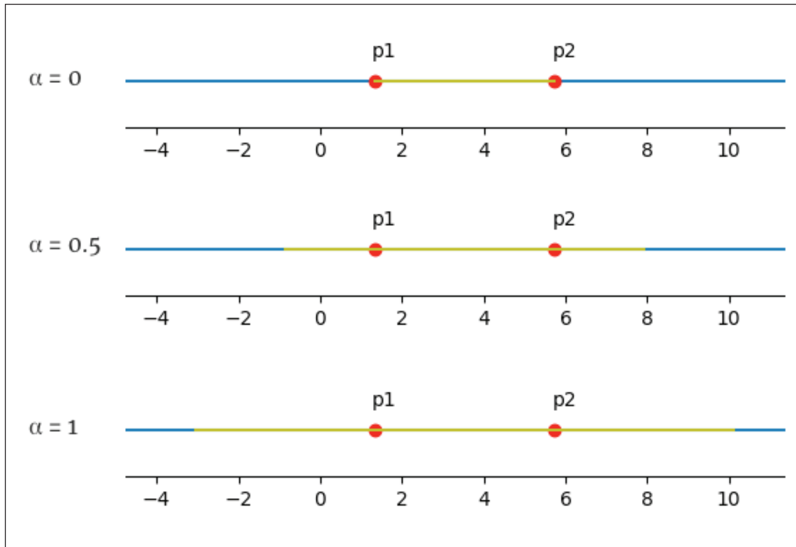
$$[parent_1 - \alpha(parent_2 - parent_1), parent_2 + \alpha(parent_2 - parent_1)].$$

Параметр α – постоянная от 0 до 1. Чем больше α, тем шире интервал. Например, если значения родителей равны 1.33 и 5.72, то:

- при α = 0 получается интервал [1.33, 5.72] (такой же, как между родителями);

- при α = 0.5 получается интервал [–0.865, 7.915] (в два раза шире, чем между родителями);
- при α = 1.0 получается интервал [–3.06, 10.11] (в три раза шире, чем между родителями).

Эти примеры показаны на рисунке ниже, где родители обозначены p1 и p2, а интервал скрещивания изображен желтым цветом:



Пример скрещивания смешением

При использовании этого метода скрещивания обычно полагают α = 0.5.

Имитация двоичного скрещивания

Идея **имитации двоичного скрещивания** (simulated binary crossover – **SBX**) – имитировать свойства одноточечного скрещивания, часто применяемого для двоичных хромосом. Одно из свойств заключается в том, что среднее значений родителей равно среднему значений потомков.

В случае метода SBX два потомка создаются из родителей по следующей формуле:

$$offspring_1 = \tfrac{1}{2}\,[(1 + \beta]parent_1 + (1 - \beta)parent_2]);$$
$$offspring_2 = \tfrac{1}{2}\,[(1 - \beta]parent_1 + (1 + \beta)parent_2]),$$

где β – случайное число, называемое *коэффициентом распределения*.

Эта формула обладает следующими свойствами:

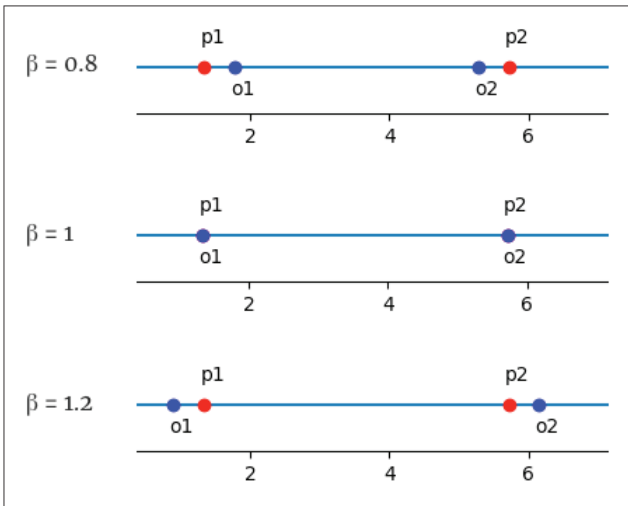
- среднее потомков равно среднему родителей независимо от значений β;
- если β = 1, потомки являются копиями родителей;

- когда $\beta < 1$, потомки располагаются ближе друг к другу, чем родители;
- когда $\beta > 1$, потомки располагаются дальше друг от друга, чем родители.

Например, если значения родителей равны 1.33 и 5.72, то имеем:

- при $\beta = 0.8$ получаются потомки 1.769 и 5.281;
- при $\beta = 1.0$ получаются потомки 1.33 и 5.72;
- при $\beta = 1.2$ получаются потомки 0.891 и 6.159.

Все три случая показаны на следующем рисунке, где родители обозначены p1 и p2, а потомки – o1 и o2.



Пример имитации двоичного скрещивания

В каждом из этих случаев среднее значение потомков равно 3.525, т. е. совпадает со средним родителей.

У двоичного одноточечного скрещивания есть еще одно свойство: мы стремимся сохранить сходство между потомком и родителями. Чтобы имитировать его, нужно выбирать β из случайного распределения. Плотность вероятности β должна быть гораздо выше в окрестности 1, где потомки похожи на родителей. Для этого β вычисляется с помощью еще одной случайной величины, обозначаемой u, которая равномерно распределена в интервале [0, 1]. Ели значение u выбрано, то β вычисляется следующим образом:

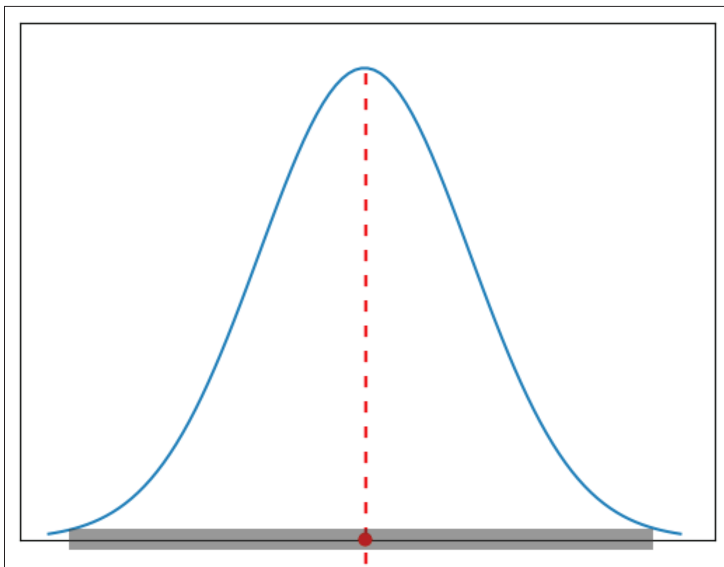
- если $u \leqslant 0.5$: $\beta = (2u)^{\frac{1}{\eta+1}}$,
- иначе: $\beta = \left(\frac{1}{2}(1-u)\right)^{\frac{1}{\eta+1}}$.

Параметр η в этих формулах – постоянная, называемая **индексом распределения**. Чем больше значение η, тем сильнее потомок похож на своих родителей. Обычно выбирают η равным 10 или 20.

Вещественная мутация

Один из способов применения мутации в генетических алгоритмах с вещественным кодированием – заменить любое вещественное значение совершенно новым, случайно сгенерированным. Однако при этом может оказаться, что мутировавший индивидуум не имеет ничего общего с исходным.

Другой подход – сгенерировать случайное вещественное число, находящееся рядом с исходным индивидуумом. Примером может служить **нормально распределенная** (или **гауссова**) **мутация**: случайное число выбирается из нормального распределения со средним 0 и каким-то заранее заданным стандартным отклонением, как показано на рисунке ниже.



Пример гауссовой мутации

Далее мы рассмотрим еще два вопроса: элитизм и образование ниш.

Элитизм

Хотя средняя приспособленность популяции в генетическом алгоритме, вообще говоря, возрастает от поколения к поколению, в любой момент может случиться так, что лучшие индивидуумы в текущем поколении исчезнут. Это связано с тем, что операторы отбора, скрещивания и мутации изменяют индивидуумов в процессе создания следующего поколения. Во многих случаях потеря временная, поскольку эти (или даже лучшие) индивидуумы снова появятся в будущем поколении.

Но если мы хотим гарантировать, что лучшие индивидуумы обязательно переходят в следующее поколение, то можем применить факультативную стратегию элитизма. Это означает, что n лучших индивидуумов (n – небольшое, заранее заданное число) копируются в следующее поколение, до того как все места будут заняты потомками, полученными в результате отбора, скрещивания и мутации. Скопированные элитные индивидуумы по-прежнему могут использоваться как родители новых индивидуумов.

Иногда элитизм оказывает заметный положительный эффект на качество алгоритма, поскольку не нужно тратить времени на повторное открытие хороших решений, потерянных в результате эволюции.

Еще один интересный способ улучшить результаты генетического алгоритма – воспользоваться образованием ниш, как описано в следующем разделе.

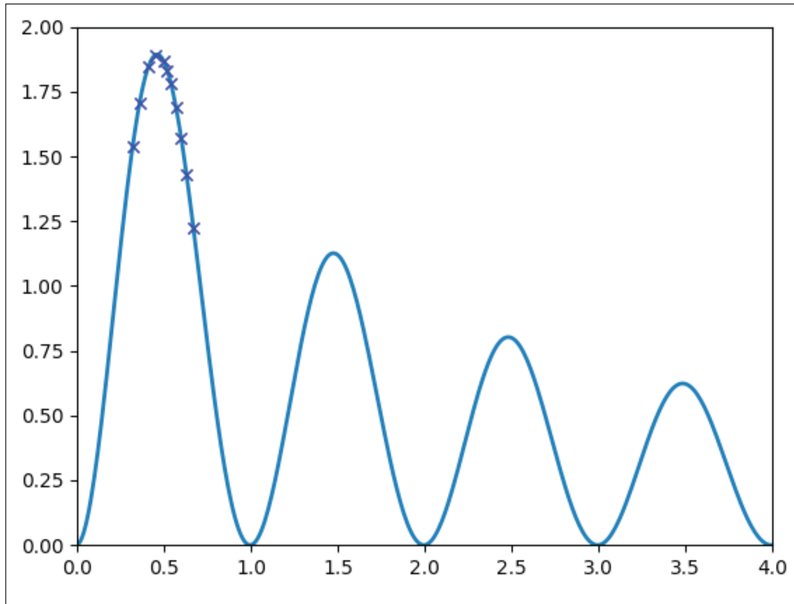
Образование ниш и разделение

В природе любая окружающая среда разбивается на несколько меньших сред, или ниш, занятых различными видами, которые обращают себе на пользу уникальные ресурсы ниши, например пищу и укрытие. Так, в лесу можно выделить верхушки деревьев, подлесок, лесную подстилку, корни деревьев и т. д. В каждой такой нише обитают разные виды, приспособленные к существованию именно в ней.

Если в одной нише обитают несколько разных видов, то они конкурируют за одни и те же ресурсы, поэтому наблюдается тенденция к поиску новых свободных ниш и их заселению.

В генетических алгоритмах феномен ниш можно использовать для поддержания разнообразия популяции, а также для поиска нескольких оптимальных решений, каждое из которых считается нишей.

Пусть, например, наш генетический алгоритм должен максимизировать функцию приспособленности, имеющую несколько пиков разной высоты, как на графике ниже.

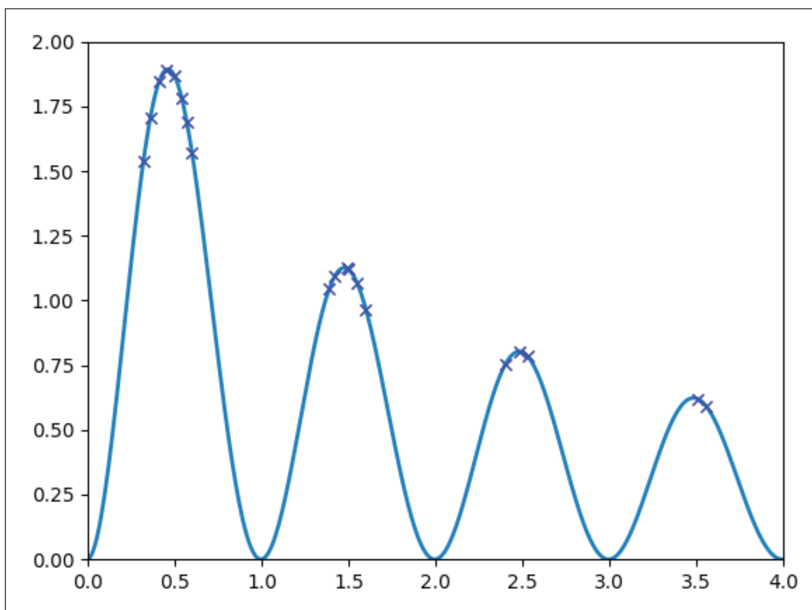


Ожидаемые результаты генетического алгоритма без образования ниш

Поскольку генетический алгоритм обычно стремится найти глобальный максимум, можно ожидать, что через некоторое время большая часть популяции сосредоточится в районе верхнего пика. На рисунке это показано крестиками на графике функции, представляющими индивидуумов в текущем поколении.

Однако иногда нам нужно найти не только глобальный максимум, но и некоторые (или все) другие пики. Для этого мы должны рассматривать каждый пик как нишу, предлагающую ресурсы в объеме, пропорциональном высоте. Затем мы ищем способ разделить эти ресурсы между индивидуумами, занимающими нишу. В идеале популяция при этом должна распределиться соответственно, так что самый высокий пик будет привлекать больше всего индивидуумов, поскольку предлагает наибольшее вознаграждение, а прочие пики будут заселены не так плотно, потому что вознаграждение в них меньше.

Эта идеальная ситуация изображена на рисунке ниже.



Результаты идеального генетического алгоритма с образованием ниш

Проблема теперь в том, как реализовать этот механизм разделения. Один из вариантов – разделить исходное значение приспособленности каждого индивидуума на число, равное какой-то функции от комбинации расстояний до всех остальных индивидуумов. Другой вариант – разделить исходную приспособленность каждого индивидуума на число других индивидуумов, отстоящих от него не более чем на некоторое пороговое расстояние.

Последовательное и параллельное образование ниш

К сожалению, описанную выше идею ниш трудно реализовать, потому что при этом возрастает сложность вычисления приспособленности. На практике понадобится также умножить исходный размер популяции на количество ожидаемых пиков (которое в общем случае неизвестно).

Один из способов преодолеть эти проблемы – искать пики по одному (последовательное образование ниш), а не пытаться найти все сразу (параллельное образование ниш). Для реализации последовательного образования ниш мы используем генетический алгоритм как обычно и находим лучшее решение. Затем функция приспособленности изменяется таким образом, чтобы «сплющить» окрестность точки максимума, после чего генетический алгоритм выполняется еще раз.

В идеале мы найдем следующий по высоте пик, поскольку исходного пика больше не существует. Этот процесс можно повторять итеративно, отыскивая на каждой итерации очередной пик.

Искусство решения задач с помощью генетических алгоритмов

Генетические алгоритмы дают нам мощный и гибкий инструмент, позволяющий решать широкий спектр задач. Приступая к новой задаче, мы должны подогнать этот инструмент к ее особенностям. Для этого есть несколько подходов, описанных ниже.

Прежде всего необходимо определить функцию приспособленности. С ее помощью оцениваются все индивидуумы: чем больше приспособленность, тем лучше индивидуум. Функция необязательно должна быть выражена в виде математической формулы. Она может быть представлена алгоритмом, или обращением к внешней службе, или даже являться результатом игры – и это далеко не все возможности. Главное, чтобы мы могли программно получить приспособленность любого решения (индивидуума).

Затем нужно выбрать подходящий способ **кодирования хромосом**. Он основан на параметрах, передаваемых функции приспособленности. До сих пор нам встречались такие кодировки: двоичная, целочисленная, упорядоченный список, вещественная. Но бывают задачи со смешанными типами параметров, а иногда даже приходится создавать собственные кодировки.

Далее нужно определиться с методом отбора. Большинство методов работают для хромосом любого типа. Если функция приспособленности как таковая недоступна, но мы все-таки можем сказать, какой из двух кандидатов лучше, то можно воспользоваться турнирным методом отбора.

В предыдущих разделах мы видели, что выбор операторов **скрещивания** и **мутации** зависит от кодировки хромосом. Для двоичных хромосом применяются не такие схемы скрещивания и мутации, как для вещественных. И в этом случае также можно придумать собственные методы скрещивания и мутации, отвечающие специфике задачи.

Наконец, следует помнить о гиперпараметрах алгоритма. Вот некоторые из наиболее распространенных:

- размер популяции;
- частота скрещивания;
- частота мутаций;
- максимальное количество поколений;
- другие условия остановки;
- элитизм (использовать или нет, какого размера).

Мы можем начать со значений, кажущихся разумными, а затем настроить их, как поступаем с гиперпараметрами любого другого алгоритма оптимизации и обучения.

Если все это кажется вам неподъемным, не паникуйте! В следующих главах мы снова и снова будем проделывать все эти действия для разного рода задач. А дочитав книгу до конца, вы сможете самостоятельно принимать такие решения.

Резюме

В этой главе мы познакомились с общей структурой генетического алгоритма. Затем мы рассмотрели детали: создание популяции, вычисление функции приспособленности, применение генетических операторов и проверку условий остановки.

Далее обсудили различные методы отбора, в т. ч. правило рулетки, стохастическую универсальную выборку, ранжированный отбор, масштабирование приспособленности и турнирный отбор, и рассказали, чем они отличаются друг от друга.

После этого мы рассмотрели несколько методов скрещивания: одноточечное, двухточечное и *k*-точечное, а также упорядоченное и смешением.

Мы также познакомились с различными методами мутации: инвертирование бита, обмен, обращение и перетасовка.

Затем были представлены вещественные генетические алгоритмы со свойственными им специальными способами кодирования хромосом и генетическими операторами скрещивания и мутации.

Мы также поговорили о концепциях элитизма, образования ниш и разделения ресурсов в генетических алгоритмах.

И напоследок перечислили, что нужно сделать, чтобы применить к задаче генетический алгоритм, эту процедуру мы будем повторять в книге многократно.

В следующей главе начнется интересное – программирование на Python! Мы познакомимся с каркасом эволюционных вычислений DEAP – эффективным инструментом применения генетических алгоритмов к широкому кругу задач. DEAP будет использоваться нами при разработке Python-программ для решения различных проблем.

Для дальнейшего чтения

За дополнительными сведениями обращайтесь к главе 8 «Генетические алгоритмы» книги Prateek Joshi «Artificial Intelligence with Python», доступной по адресу https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781786464392/8.

ЧАСТЬ II

РЕШЕНИЕ ЗАДАЧ С ПОМОЩЬЮ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ

В этой части мы займемся применением генетических алгоритмов на Python к решению различных практических задач

Глава 3

Каркас DEAP

В этой главе, как и было обещано, начнется интересное! Мы познакомимся с DEAP – мощным и гибким **каркасом эволюционных вычислений** для решения практических задач с помощью генетических алгоритмов. После краткого введения мы представим два главных модуля – `creator` и `toolbox` – и научимся создавать различные компоненты генетического алгоритма. Затем напишем программу для решения задачи OneMax – своего рода «Hello World» в мире генетических алгоритмов. После этого мы покажем более короткий вариант той же программы, в котором используются встроенные в каркас алгоритмы. А под конец главы мы припасли самое увлекательное – эксперименты с различными настройками созданных генетических алгоритмов и выяснение того, на что они влияют.

К концу главы вы:

- познакомитесь с каркасом DEAP и имеющимися в нем модулями для генетических алгоритмов;
- будете понимать, как работать с модулями `creator` и `toolbox` из каркаса DEAP;
- сможете перевести простую задачу на язык генетического алгоритма;
- научитесь строить решение в виде генетического алгоритма с помощью каркаса DEAP;
- будете знать, как использовать встроенные в DEAP алгоритмы для создания лаконичного кода;
- решите задачу OneMax с помощью генетического алгоритма и каркаса DEAP;
- научитесь экспериментировать с параметрами генетических алгоритмов и интерпретировать результаты.

Технические требования

Последние версии DEAP можно использовать с Python 2 и 3. В этой книге применяется Python 3.7. Дистрибутив самого языка Python можно скачать с сайта Python Software Foundation по адресу https://www.python.org/downloads/. Дополнительные полезные материалы можно найти по адресу https://realpython.com/installing-python/.

Каркас DEAP рекомендуется устанавливать с помощью программ `easy_install` или `pip`, например:

```
pip install deap
```

Дополнительные сведения смотрите в документации по DEAP по адресу https://deap.readthedocs.io/en/master/installation.html. Дистрибутив для Conda имеется по адресу https://anaconda.org/conda-forge/deap.

В книге используются и другие Python-пакеты. В частности, в этой главе понадобятся пакеты:

- NumPy: http://www.numpy.org/;
- Matplotlib: https://matplotlib.org/;
- Seaborn: https://seaborn.pydata.org/.

Итак, мы готовы к работе с DEAP. В следующих двух разделах рассматриваются самые полезные инструменты и утилиты. Но сначала немного познакомимся с DEAP и разберемся, почему мы выбрали именно этот каркас для изучения генетических алгоритмов.

Код приведенных в этой главе программ можно найти в репозитории книги на GitHub по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter03. Чтобы увидеть код в действии, посмотрите видео по адресу http://bit.ly/3aTym1p.

Введение в DEAP

В предыдущих главах мы видели, что основные идеи генетических алгоритмов и их структура довольно просты, как и большинство генетических операторов. Поэтому вполне посильно разработать программу, реализующую генетический алгоритм решения конкретной задачи, с нуля.

Однако, как часто бывает при разработке программного обеспечения, использование проверенной специализированной библиотеки может облегчить жизнь. Она помогает создавать решения быстрее и с меньшим числом ошибок, а также предлагает на выбор много отработанных заготовок, вместо того чтобы изобретать велосипед.

Для работы с генетическими алгоритмами создан целый ряд каркасов на Python, например GAFT, Pyevolve и PyGMO. Но, рассмотрев различные варианты, мы решили остановиться на каркасе DEAP, поскольку он прост в использовании и предлагает широкий набор функций, поддерживает расширяемость и может похвастаться подробной документацией.

DEAP (сокращение от Distributed Evolutionary Algorithms in Python – распределенные эволюционные алгоритмы на Python) поддерживает быструю разработку решений с применением генетических алгоритмов и других методов эволюционных вычислений. DEAP предлагает различные структуры данных и инструменты, необходимые для реализации самых разных решений на основе генетических алгоритмов.

Каркас DEAP был разработан в канадском университете Лаваля в 2009 г. и предлагается на условиях лицензии GNU Lesser General Public License (LGPL). Исходный код DEAP доступен по адресу https://github.com/DEAP/deap, а документация размещена по адресу https://deap.readthedocs.io/en/master/.

Использование модуля creator

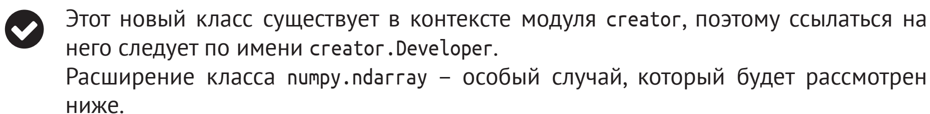
Мы начнем с рассмотрения модуля DEAP `creator`. Он используется как метафабрика и позволяет расширять существующие классы, добавляя в них новые атрибуты. Пусть, например, имеется класс `Employee`. С помощью модуля `creator` мы можем создать из него класс `Developer` следующим образом:

```
from deap import creator
creator.create("Developer", Employee, position = "Developer",
programmingLanguages = set)
```

Первым аргументом функции `create()` передается имя нового класса, вторым – существующий класс, подлежащий расширению. Все последующие аргументы определяют атрибуты нового класса. Если значением аргумента является класс (например, `dict` или `set`), то он будет добавлен в новый класс как атрибут экземпляра, инициализируемый в конструкторе. Если же это не класс (а, например, литерал), то он добавляется как атрибут класса (статический).

Таким образом, созданный класс `Developer` расширяет класс `Employee` и имеет атрибут класса `position`, равный строке `Developer`, и атрибут экземпляра `programmingLanguages` типа `set`, который инициализируется в конструкторе. Следовательно, новый класс эквивалентен такому:

```
class Developer(Employee):
    position = "Developer"

    def __init__(self):
        self.programmingLanguages = set()
```



Этот новый класс существует в контексте модуля `creator`, поэтому ссылаться на него следует по имени `creator.Developer`.
Расширение класса `numpy.ndarray` – особый случай, который будет рассмотрен ниже.

При работе с DEAP модуль `creator` обычно служит для создания классов `Fitness` и `Individual`, используемых в генетических алгоритмах.

Создание класса Fitness

При работе с DEAP значения приспособленности инкапсулированы в классе `Fitness`. DEAP позволяет распределять приспособленность по нескольким компонентам (называемым целями), у каждого из которых есть свой вес. Комбинация весов определяет поведение, или стратегию приспособления в конкретной задаче.

Определение стратегии приспособления

Для определения стратегии в состав DEAP входит абстрактный класс `base.Fitness`, который содержит кортеж `weights`. Этому кортежу необходимо при-

своить значения, чтобы определить стратегию и сделать класс пригодным для использования. Для этого мы расширяем базовый класс `Fitness` с помощью модуля `creator` так же, как делали это ранее с классом `Developer`:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

Получается класс `creator.FitnessMax`, расширяющий класс `base.Fitness`, в котором атрибут класса `weights` инициализирован значением `(1.0,)`.

Обратите внимание на запятую в конце определения `weights`, которая присутствует, хотя задан всего один вес. Она необходима, потому что `weights` – кортеж.

Стратегия этого класса `FitnessMax` – максимизировать приспособленность индивидуумов с единственной целью. Если бы нам нужно было минимизировать приспособленность в задаче с одной целью, то для задания соответствующей стратегии можно было бы определить такой класс:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

Можно также определить класс для оптимизации сразу нескольких целей различной важности:

```
creator.create("FitnessCompound", base.Fitness, weights=(1.0, 0.2, -0.5))
```

В классе `creator.FitnessCompound` используется три компоненты приспособленности с весами 1.0, 0.2 и –0.5. Это означает, что первая и вторая компоненты (цели) максимизируются, а третья минимизируется, причем первая компонента самая важная, следующей по важности является третья, а последней – вторая.

Хранение значения приспособленности

Если кортеж `weights` определяет стратегию приспособления, то кортеж `values` используется для хранения самих значений функции приспособленности в базовом классе `base.Fitness`. Эти значения дает отдельно определяемая функция, которую обычно называют `evaluate()`; мы опишем ее ниже в этой главе. Как и `weights`, кортеж `values` содержит по одному значению для каждой компоненты приспособленности (цели).

Третий кортеж, `wvalues`, содержит взвешенные значения, полученные перемножением элементов кортежа `values` и соответственных элементов кортежа `weights`. Всякий раз, как устанавливаются значения приспособленности, соответствующие взвешенные значения вычисляются и сохраняются в `wvalues`. Они используются при сравнении индивидуумов.

Взвешенные приспособленности можно сравнивать лексикографически с помощью следующих операторов:

```
>, <, >=, <=, ==, !=
```

Созданный класс `Fitness` мы будем использовать в определении класса `Individual`, как описано в следующем разделе.

Создание класса Individual

Второе типичное применение модуля `creator` – определение индивидуумов, образующих популяцию в генетическом алгоритме. В предыдущих главах мы видели, что индивидуумы представлены хромосомами, которыми можно манипулировать с помощью генетических операторов. В DEAP класс `Individual` создается путем расширения базового класса, представляющего хромосому. Кроме того, каждый экземпляр класса `Individual` должен содержать функцию приспособленности в качестве атрибута.

Чтобы удовлетворить обоим требованиям, мы воспользуемся модулем `creator` для создания класса `creator.Individual`:

```
creator.create("Individual", list, fitness=creator.FitnessMax)
```

Результат работы этой строки двоякий:

- созданный класс `Individual` расширяет встроенный класс Python `list`. Это означает, что все хромосомы имеют тип `list`;
- в каждом экземпляре класса `Individual` имеется атрибут `fitness` созданного ранее класса `FitnessMax`.

В следующем разделе мы рассмотрим класс `Toolbox`.

Использование класса Toolbox

Второй механизм, предлагаемый каркасом DEAP, – класс `base.Toolbox`. Он используется как контейнер для функций (или операторов) и позволяет создавать новые операторы путем назначения псевдонимов или настройки существующих функций.

Пусть, например, имеется следующая функция `sumOfTwo()`:

```
def sumOfTwo(a, b):
    return a + b
```

Воспользовавшись модулем `toolbox`, мы сможем создать новый оператор `incrementByFive()` на основе функции `sumOfTwo()`:

```
from deap import base
toolbox= base.Toolbox()
toolbox.register("incrementByFive", sumOfTwo, b=5)
```

Первым аргументом функции `register()` передается имя нового оператора (или псевдоним существующего), вторым – настраиваемая функция. Все остальные (необязательные) аргументы передаются этой функции при вызове нового оператора. Рассмотрим, к примеру, следующее определение:

```
toolbox.incrementByFive(10)
```

Этот вызов эквивалентен такому:

```
sumOfTwo(10, 5)
```

поскольку аргументу `b` было присвоено фиксированное значение 5 в определении оператора `incrementByFive`.

Создание генетических операторов

Во многих случаях класс `Toolbox` используется для настройки существующих функций из модуля `tools`, который содержит ряд полезных функций, относящихся к генетическим операциям отбора, скрещивания и мутации, а также утилиты для инициализации.

Например, в коде ниже определены три псевдонима, которые впоследствии будут использованы как генетические операторы:

```
from deap import tools
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.02)
```

Опишем, что здесь было сделано.

- Имя `select` зарегистрировано как псевдоним существующей в модуле `tools` функции `selTournament()` с аргументом `tournsize = 3`. В результате создается оператор `toolbox.select`, который выполняет турнирный отбор с размером турнира 3.
- Имя `mate` зарегистрировано как псевдоним существующей в модуле `tools` функции `cxTwoPoint()`. В результате создается оператор `toolbox.mate`, который выполняет двухточечное скрещивание.
- Имя `mutate` зарегистрировано как псевдоним существующей в модуле `tools` функции `mutFlipBit` с аргументом `indpb = 0.02`. В результате создается оператор `toolbox.mutate`, который выполняет мутацию инвертированием бита с вероятностью `0.02`.

Модуль `tools` предоставляет реализации различных генетических операторов, в т. ч. упомянутых в предыдущей главе.

Функции отбора находятся в файле `selection.py`. Перечислим некоторые из них:

- `selRoulette()` – отбор по правилу рулетки;
- `selStochasticUniversalSampling()` – стохастическая универсальная выборка;
- `selTournament()` – турнирный отбор.

Функции скрещивания находятся в файле `crossover.py`:

- `cxOnePoint()` – одноточечное скрещивание;
- `cxUniform()` – равномерное скрещивание;
- `cxOrdered()` – упорядоченное скрещивание (OX1);
- `cxPartialyMatched()` – скрещивание с частичным сопоставлением (partially matched crossover – PMX).

В файле `mutation.py` находятся две функции мутации:

- `mutFlipBit()` – мутация инвертированием бита;
- `mutGaussian()` – нормально распределенная мутация.

Создание популяции

Файл init.py модуля tools содержит несколько функций, полезных для создания и инициализации популяции. Особенно полезна функция initRepeat(), принимающая три аргумента:

- тип контейнера результирующих объектов;
- функция, генерирующая объекты, которые помещаются в контейнер;
- сколько объектов генерировать.

Например, следующая строка создает список из 30 случайных чисел от 0 до 1:

```
randomList = tools.initRepeat(list, random.random, 30)
```

В этом примере list – тип, выступающий в роли заполняемого контейнера, random.random – порождающая функция, а 30 – количество вызовов этой функции, необходимых для заполнения контейнера.

Что, если мы захотим заполнить список случайными целыми числами, равными 0 или 1? Можно было бы создать функцию, которая вызывает random.radint() для генерации одного случайного числа, равного 0 или 1, а затем использовать ее в качестве порождающей функции в initRepeat(), как показано ниже:

```
def zeroOrOne():
    return random.randint(0, 1)

randomList = tools.initRepeat(list, zeroOrOne, 30)
```

Или можно воспользоваться модулем toolbox:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
randomList = tools.initRepeat(list, toolbox.zeroOrOne, 30)
```

Здесь вместо того чтобы явно определять функцию zeroOrOne(), мы создали оператор (или псевдоним) zeroOrOne, который вызывает random.radint() с фиксированными параметрами 0 и 1.

Вычисление приспособленности

Как было отмечено выше, в классе Fitness задаются веса, определяющие стратегию приспособления (например, максимизация или минимизация), а сами значения приспособленности возвращает отдельно определенная функция. Эта функция обычно регистрируется в модуле toolbox под псевдонимом evaluate, как показано ниже:

```
def someFitnessCalculationFunction(individual):
    return _some_calculation_of_the_fitness

toolbox.register("evaluate",someFitnessCalculationFunction)
```

В данном примере функция `someFitnessCalculationFunction()` вычисляет приспособленность заданного индивидуума, а имя `evaluate` зарегистрировано в качестве ее псевдонима.

Вот теперь мы готовы воспользоваться полученными знаниями и решить первую задачу, применив DEAP для реализации генетического алгоритма.

Задача OneMax

OneMax (или One-Max) – это простая задача оптимизации, которую часто приводят в пример как аналог программы «Hello World» в мире генетических алгоритмов. В этой главе мы будем использовать ее для демонстрации возможностей каркаса DEAP.

Задача OneMax состоит в том, чтобы найти двоичную строку заданной длины, для которой сумма составляющих ее цифр максимальна. Например, при решении задачи OneMax длины 5 будут рассматриваться такие кандидаты:

- 10010 (сумма цифр = 2);
- 01110 (сумма цифр = 3);
- 11111 (сумма цифр = 5).

Очевидно (нам), что решением всегда является строка, состоящая из одних единиц. Но генетический алгоритм не обладает таким знанием, поэтому должен слепо искать решение, пользуясь генетическими операторами. Если алгоритм справится с работой, то найдет решение (или приближение к нему) за разумное время.

В документации по каркасу DEAP задача OneMax приведена в качестве вводного примера (https://github.com/DEAP/deap/blob/master/examples/ga/onemax.py). В следующих разделах мы опишем наш подход к ее решению.

Решение задачи OneMax с помощью DEAP

В предыдущей главе мы перечислили несколько действий, которые нужно предпринять для решения задачи с помощью генетического алгоритма. Эти действия ниже будут оформлены в виде шагов. В последующих главах те же шаги встретятся нам в процессе применения генетических алгоритмов к решению различных задач.

Выбор хромосомы

Поскольку в задаче OneMax мы имеем дело с двоичными строками, с выбором хромосом проблем не возникает – каждый индивидуум представлен двоичной строкой, которая непосредственно соответствует потенциальному решению. На языке Python это реализуется в виде списка, содержащего числа 0 или 1. Длина хромосомы совпадает с размером задачи OneMax. Например,

в задаче OneMax размера 5 индивидуум 10010 будет представлен списком `[1, 0, 0, 1, 0]`.

Вычисление приспособленности

Поскольку мы хотим найти индивидуума с **наибольшей** суммой цифр, следует использовать стратегию `FitnessMax`. А поскольку каждый индивидуум представлен списком целых чисел, равных 0 или 1, то приспособленность вычисляется просто как сумма элементов списка, например: `sum([1, 0, 0, 1, 0]) = 2`.

Выбор генетических операторов

Теперь нужно решить, какие реализации генетических операторов отбора, скрещивания и мутации использовать. В предыдущей главе мы рассмотрели несколько таких реализаций. Выбор конкретного оператора – не точная наука, обычно приходится экспериментировать. Но если операторы отбора обычно могут работать с хромосомами любого типа, то операторы скрещивания и мутации должны соответствовать типу хромосомы, иначе будут получаться недопустимые хромосомы.

В качестве оператора отбора можно для начала взять турнир, потому что его реализация проста и эффективна. Позже можно будет поэкспериментировать с другими стратегиями, например правилом рулетки и SUS.

Что касается скрещивания, то подойдет одноточечный или двухточечный оператор, поскольку в результате скрещивания двух двоичных строк этими методами получается допустимая двоичная строка.

В качестве операции мутации можно взять простое инвертирование бита, этот метод хорошо работает для двоичных строк.

Задание условия остановки

Всегда имеет смысл ограничивать количество поколений, чтобы алгоритм не работал вечно. Тем самым мы получаем одно условие остановки.

Кроме того, так уж получилось, что мы знаем наилучшее решение в задаче OneMax – двоичная строка из одних единиц со значением приспособленности, равным количеству единиц, – поэтому можно взять это в качестве второго условия остановки.

Если хотя бы одно условие выполнено – достигнут предел количества поколений или найдено наилучшее решение, то алгоритм останавливается.

Реализация средствами DEAP

Теперь можно, наконец, приступить к написанию кода решения задачи OneMax с помощью каркаса DEAP.

Полный код программы имеется по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter03/01-OneMax-long.py.

Подготовка

Прежде чем запускать главный цикл алгоритма, нужно все подготовить, и в DEAP это делается вполне определенным образом.

1. Сначала импортируем необходимые модули из DEAP и еще несколько вспомогательных библиотек:

```
from deap import base
from deap import creator
from deap import tools
import random
import matplotlib.pyplot as plt
```

2. Затем объявляем несколько констант, содержащих значения параметров самой задачи и генетического алгоритма:

```
# константы задачи
ONE_MAX_LENGTH = 100    # длина подлежащей оптимизации битовой строки

# константы генетического алгоритма
POPULATION_SIZE = 200   # количество индивидуумов в популяции
P_CROSSOVER = 0.9       # вероятность скрещивания
P_MUTATION = 0.1        # вероятность мутации индивидуума
MAX_GENERATIONS = 50    # максимальное количество поколений
```

3. Важный аспект генетического алгоритма – его вероятностный характер, поэтому в алгоритм нужно внести элемент случайности. Однако на этапе экспериментирования требуется, чтобы результаты были воспроизводимы. Для этого мы задаем какое-нибудь фиксированное начальное значение генератора случайных чисел:

```
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

Впоследствии эти строки нужно будет удалить, чтобы при разных прогонах получались разные результаты.

4. Выше мы видели, что одним из основных компонентов каркаса DEAP является класс `Toolbox`, который позволяет регистрировать новые функции (или операторы), настраивая поведение существующих функций. В данном случае мы воспользуемся им, чтобы определить оператор `zeroOrOne` путем специализации функции `random.randomint(a, b)`. Эта

функция возвращает случайное целое число `N` такое, что `a ≤ N ≤ b`. Если задать в качестве `a` и `b` фиксированные значения 0 и 1, то оператор `zeroOrOne` будет случайным образом возвращать 0 или 1. Во фрагменте ниже мы определяем переменную `toolbox`, а затем используем ее для регистрации оператора `zeroOrOne`:

```
toolbox = base.Toolbox()
toolbox.register("zeroOrOne", random.randint, 0, 1)
```

5. Далее следует создать класс `Fitness`. Поскольку у нас всего одна цель – сумма цифр, а наша задача – максимизировать ее, то выбираем стратегию `FitnessMax`, задав в кортеже `weights` всего один положительный вес:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

6. По соглашению, в DEAP для представления индивидуумов используется класс с именем `Individual`, для создания которого применяется модуль `creator`. В нашем случае базовым классом является `list`, т. е. хромосома представляется списком. Дополнительно в класс добавляется атрибут `fitness`, инициализируемый экземпляром определенного ранее класса `FitnessMax`:

```
creator.create("Individual", list, fitness=creator.FitnessMax)
```

7. Следующий номер нашей программы – регистрация оператора `individualCreator`, который создает экземпляр класса `Individual`, заполненный случайными значениями 0 или 1. Для этого мы настроим ранее определенный оператор `zeroOrOne`. В качестве базового класса используется вышеупомянутый оператор `initRepeat`, специализированный следующими аргументами:
   - класс `Individual` в качестве типа контейнера, в который помещаются созданные объекты;
   - оператор `zeroOrOne` в качестве функции генерации объектов;
   - константа `ONE_MAX_LENGTH` в качестве количества генерируемых объектов (сейчас она равна 100).

   Поскольку оператор `zeroOrOne` создает объекты, принимающие случайное значение 0 или 1, то получающийся в результате оператор `individualCreator` заполняет экземпляр `Individual` 100 случайными значениями 0 или 1:

```
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.zeroOrOne, ONE_MAX_LENGTH)
```

8. Наконец, регистрируем оператор `populationCreator`, создающий список индивидуумов. В его определении также используется оператор `initRepeat` со следующими аргументами:
   - класс `list` в качестве типа контейнера;
   - оператор `individualCreator`, определенный ранее в качестве функции, генерирующей объекты в списке.

Последний аргумент initRepeat – количество генерируемых объектов – здесь не задан. Это означает, что при использовании оператора populationCreator мы должны будем указать этот аргумент, т. е. задать размер популяции:

```
toolbox.register("populationCreator", tools.initRepeat,
list, toolbox.individualCreator)
```

9. Для вычисления приспособленности мы сначала определим свободную функцию, которая принимает экземпляр класса Individual и возвращает его приспособленность. В данном случае мы назвали функцию, вычисляющую количество единиц в индивидууме, oneMaxFitness. Поскольку индивидуум представляет собой не что иное, как список значений 0 и 1, то на поставленный вопрос в точности отвечает встроенная функция Python sum():

```
def oneMaxFitness(individual):
    return sum(individual),    # вернуть кортеж
```

Как уже было сказано, значения приспособленности в DEAP представлены кортежами, поэтому если возвращается всего одно значение, то после него нужно поставить запятую.

10. Теперь определим оператор evaluate – псевдоним только что определенной функции oneMaxfitness(). Ниже мы узнаем, что использование псевдонима evaluate для вычисления приспособленности – принятое в DEAP соглашение:

```
toolbox.register("evaluate", oneMaxFitness)
```

11. В предыдущем разделе мы говорили, что генетические операторы обычно создаются как псевдонимы существующих функций из модуля tools с конкретными значениями аргументов. В данном случае аргументы будут такими:
    - турнирный отбор с размером турнира 3;
    - одноточечное скрещивание;
    - мутация инвертированием бита.

    Обратите внимание на параметр indpb функции mutFlipBit. Эта функция обходит все атрибуты индивидуума – в нашем случае список значений 0 и 1 – и для каждого атрибута использует значение данного аргумента как вероятность инвертирования (применения логического оператора НЕ) значения атрибута. Это значение не зависит от вероятности мутации, которая задается константой P_MUTATION, – мы определили ее выше, но пока не использовали. Вероятность мутации нужна при решении о том, вызывать ли функцию mutFlipBit для данного индивидуума в популяции:

```
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", tools.cxOnePoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/ONE_MAX_LENGTH)
```

Итак, подготовка завершена, все операторы определены, и мы готовы реализовать генетический алгоритм.

Эволюция решения

Генетический алгоритм реализован в функции `main()`, его шаги описаны ниже.

1. Создаем начальную популяцию оператором `populationCreator`, задавая размер популяции `POPULATION_SIZE`. Также инициализируем переменную `generationCounter`, которая понадобится нам позже:

   ```
   population = toolbox.populationCreator(n=POPULATION_SIZE)
   generationCounter = 0
   ```

2. Для вычисления приспособленности каждого индивидуума в начальной популяции воспользуемся функцией Python `map()`, которая применяет оператор `evaluate` к каждому элементу популяции. Поскольку оператор `evaluate` – это псевдоним функции `oneMaxFitness()`, получающийся итерируемый объект содержит вычисленные значения приспособленности каждого индивидуума. Затем мы преобразуем его в список кортежей:

   ```
   fitnessValues = list(map(toolbox.evaluate, population))
   ```

3. Поскольку элементы списка `fitnessValues` взаимно однозначно соответствуют элементам популяции (представляющей собой список индивидуумов), мы можем воспользоваться функцией `zip()`, чтобы объединить их попарно, сопоставив каждому индивидууму его приспособленность:

   ```
   for individual, fitnessValue in zip(population, fitnessValues):
       individual.fitness.values = fitnessValue
   ```

4. Далее, так как в нашем случае имеет место приспособляемость всего с одной целью, то извлекаем первое значение из каждого кортежа приспособленности для сбора статистики:

   ```
   fitnessValues = [individual.fitness.values[0] for individual in population]
   ```

5. В качестве статистики мы собираем максимальное и среднее значение приспособленности в каждом поколении. Для этого нам понадобятся два списка, создадим их:

   ```
   maxFitnessValues = []
   meanFitnessValues = []
   ```

6. Теперь мы готовы написать главный цикл алгоритма. В самом начале цикла проверяются условия остановки. Одно из них – ограничение на количество поколений, второе – проверка на лучшее возможное решение (двоичная строка из одних единиц):

```
while max(fitnessValues) < ONE_MAX_LENGTH and generationCounter
< MAX_GENERATIONS:
```

7. Затем обновляется счетчик поколений. Он используется в условии остановки и в последующих предложениях печати:

```
generationCounter = generationCounter + 1
```

8. Сердце алгоритма – генетические операторы, которые применяются на следующем шаге. Сначала – оператор отбора `toolbox.select`, который мы выше определили как турнирный отбор. Поскольку размер турнира был задан в определении оператора, сейчас нам осталось передать только популяцию и ее размер:

```
offspring = toolbox.select(population, len(population))
```

9. Далее отобранные индивидуумы, которые находятся в списке `offspring`, клонируются, чтобы можно было применить к ним следующие генетические операторы, не затрагивая исходную популяцию:

```
offspring = list(map(toolbox.clone, offspring))
```

Заметим, что, несмотря на имя `offspring`, это пока еще клоны индивидуумов из предыдущего поколения, и к ним еще только предстоит применить оператор скрещивания для создания потомков.

10. Следующий генетический оператор – скрещивание. Ранее мы определили его в атрибуте `toolbox.mate` как псевдоним одноточечного скрещивания. Мы воспользуемся встроенной в Python операцией среза, чтобы объединить в пары каждый элемент списка `offspring` с четным индексом со следующим за ним элементом с нечетным индексом. Затем с помощью функции `random()` мы «подбросим монету» с вероятностью, заданной константой `P_CROSSOVER`, и тем самым решим, применять к паре индивидуумов скрещивание или оставить их как есть. И наконец, удалим значения приспособленности потомков, потому что они были модифицированы и старые значения уже не актуальны:

```
for child1, child2 in zip(offspring[::2], offspring[1::2]):
    if random.random() < P_CROSSOVER:
        toolbox.mate(child1, child2)
        del child1.fitness.values
        del child2.fitness.values
```

Отметим, что функция `mate` принимает двух индивидуумов и модифицирует их на месте, т. е. присваивать им новые значения не нужно.

11. Последний генетический оператор – мутация, ранее мы определили его в атрибуте `toolbox.mutate` как псевдоним инвертирования бита. Мы должны обойти всех потомков и применить оператор мутации с вероятностью `P_MUTATION`. Если индивидуум подвергся мутации, то нужно удалить значение его приспособленности (если оно существует), по-

скольку оно могло быть перенесено из предыдущего поколения, а после мутации уже не актуально:

```
for mutant in offspring:
    if random.random() < P_MUTATION:
        toolbox.mutate(mutant)
        del mutant.fitness.values
```

12. Те индивидуумы, к которым не применялось ни скрещивание, ни мутация, остались неизменными, поэтому их приспособленности, вычисленные в предыдущем поколении, не нужно заново пересчитывать. В остальных индивидуумах значение приспособленности будет пустым. Мы находим этих индивидуумов, проверяя свойство `valid` класса `Fitness`, после чего вычисляем новое значение приспособленности так же, как делали это ранее:

```
freshIndividuals = [ind for ind in offspring if not ind.fitness.valid]
freshFitnessValues = list(map(toolbox.evaluate, freshIndividuals))
for individual, fitnessValue in zip(freshIndividuals, freshFitnessValues):
    individual.fitness.values = fitnessValue
```

13. После того как все генетические операторы применены, нужно заменить старую популяцию новой:

```
population[:] = offspring
```

14. Прежде чем переходить к следующей итерации, учтем в статистике текущие значения приспособленности. Поскольку приспособленность представлена кортежем (из одного элемента), необходимо указать индекс [0]:

```
fitnessValues = [ind.fitness.values[0] for ind in population]
```

15. Далее мы вычисляем максимальное и среднее значения, помещаем их в накопители и печатаем сводную информацию:

```
maxFitness = max(fitnessValues)
meanFitness = sum(fitnessValues) / len(population)
maxFitnessValues.append(maxFitness)
meanFitnessValues.append(meanFitness)
print("- Поколение {}: Макс приспособ. = {}, Средняя приспособ. = {}"
    .format(generationCounter, maxFitness, meanFitness))
```

16. Дополнительно мы находим индекс (первого) лучшего индивидуума, пользуясь только что найденным значением приспособленности, и распечатываем этого индивидуума:

```
t_index = fitnessValues.index(max(fitnessValues))
print("Лучший индивидуум = ", *population[best_index], "\n")
```

17. После срабатывания условий остановки накопители статистики можно использовать для построения двух графиков с помощью библиотеки `matplotlib`. Во фрагменте кода ниже рисуется график изменения лучшей и средней приспособленности:

```
plt.plot(maxFitnessValues, color='red')
plt.plot(meanFitnessValues, color='green')
plt.xlabel(Поколение')
plt.ylabel('Макс/средняя приспособленность')
plt.title('Зависимость максимальной и средней приспособленности от поколения')
plt.show()
```

Теперь мы можем испытать свой первый генетический алгоритм – запустим его и найдем решение задачи OneMax.

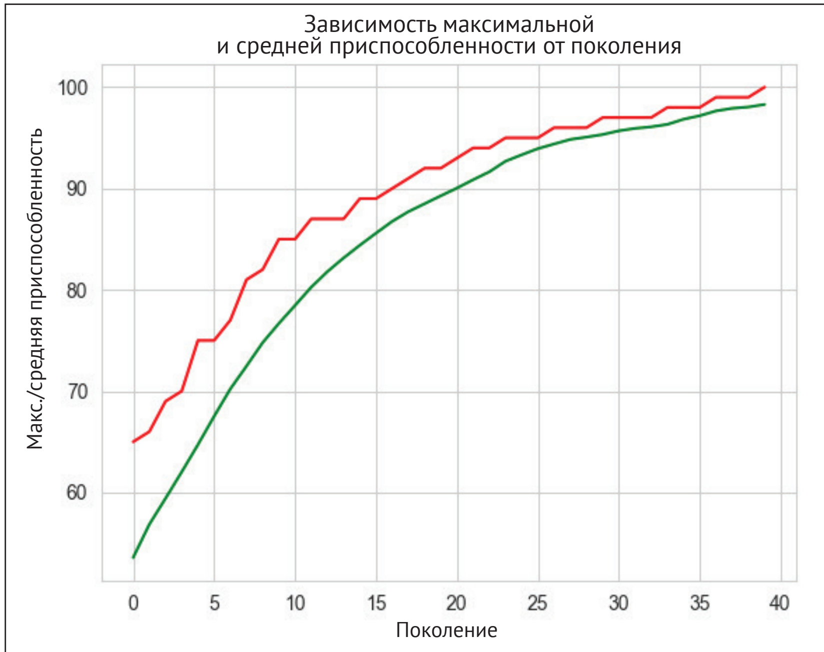
Выполнение программы

В результате выполнения написанной выше программы получается такая распечатка:

```
- Поколение 1: Макс приспособ. = 65.0, Средняя приспособ. = 53.575
Лучший индивидуум = 1 1 0 1 0 1 0 0 1 0 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 1 1 1
1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 0 0
0 1 0 1 0 1 1 1 0 1 1 0 0 0 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
...
- Поколение 40: Макс приспособ. = 100.0, Средняя приспособ. = 98.29
Лучший индивидуум = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Как видим, после смены 40 поколений было найдено решение, содержащее только единицы, для которого приспособленность равна 100. После этого алгоритм остановился. Начальное значение средней приспособленности было равно примерно 53, конечное – 100.

Ниже показан график, построенный с помощью `matplotlib`.



Статистика программы для решения задачи OneMax

Как видно из графика, максимальная приспособленность возрастает скачкоообразно, а средняя – плавно.

Итак, мы решили задачу OneMax с помощью каркаса DEAP, а теперь посмотрим, как можно сделать код более лаконичным.

Использование встроенных алгоритмов

Каркас DEAP включает несколько встроенных эволюционных алгоритмов, находящихся в модуле `algorithms`. Один из них, `eaSimple`, реализует общую структуру генетического алгоритма и может заменить большую часть написанного нами кода в функции `main`. Для сбора и печати статистики можно использовать другие полезные объекты DEAP, `Statistics` и `logbook`.

Описанная в этом разделе программа реализует то же решение задачи OneMax, что и программа из предыдущего раздела, но с меньшим объемом кода. Отличается только метод `main`. Ниже мы опишем различия, а полный код имеется по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter03/02-OneMax-short.py.

Объект Statistics

Первое изменение относится к способу сбора статистики. Для этой цели мы воспользуемся классом `tools.Statistics`, предоставляемым DEAP. Он позволяет собирать статистику, задавая функцию, применяемую к данным, для которых вычисляется статистика.

1. Поскольку в нашем случае данными является популяция, зададим функцию, которая извлекает приспособленность каждого индивидуума:

   ```
   stats = tools.Statistics(lambda ind: ind.fitness.values)
   ```

2. Теперь можно зарегистрировать функции, применяемые к этим значениям на каждом шаге. В нашем примере это функции NumPy `max` и `mean`, но можно регистрировать и другие функции (например, `min` и `std`):

   ```
   stats.register("max", numpy.max)
   stats.register("avg", numpy.mean)
   ```

Как мы скоро увидим, собранная статистика возвращается в объекте `logbook` в конце работы программы.

Алгоритм

Теперь можно приступить к основной работе. Для этого нужно всего одно обращение к методу `algorithms.eaSimple`, одному из встроенных в DEAP эволюционных алгоритмов. Этому методу передаются популяция, `toolbox`, объект статистики и другие параметры:

```
population, logbook = algorithms.eaSimple(population, toolbox,
                                          cxpb=P_CROSSOVER,
                                          mutpb=P_MUTATION,
                                          ngen=MAX_GENERATIONS,
                                          stats=stats, verbose=True)
```

Метод `algorithms.eaSimple` предполагает, что в `toolbox` уже зарегистрированы операторы `evaluate`, `select`, `mate` и `mutate`, – и мы действительно сделали это в первоначальной версии программы. Условие остановки задается с помощью параметра `ngen` – максимального количества поколений.

Объект logbook

Метод `algorithms.eaSimple` возвращает два объекта – конечную популяцию и объект `logbook`, содержащий собранную статистику. Интересующую нас статистику можно извлечь методом `select()` и использовать для построения графиков, как и раньше:

```
maxFitnessValues, meanFitnessValues = logbook.select("max", "avg")
```

Теперь можно запустить эту сильно похудевшую версию программы.

Выполнение программы

При запуске программы с теми же параметрами, что и раньше, выдается такая распечатка:

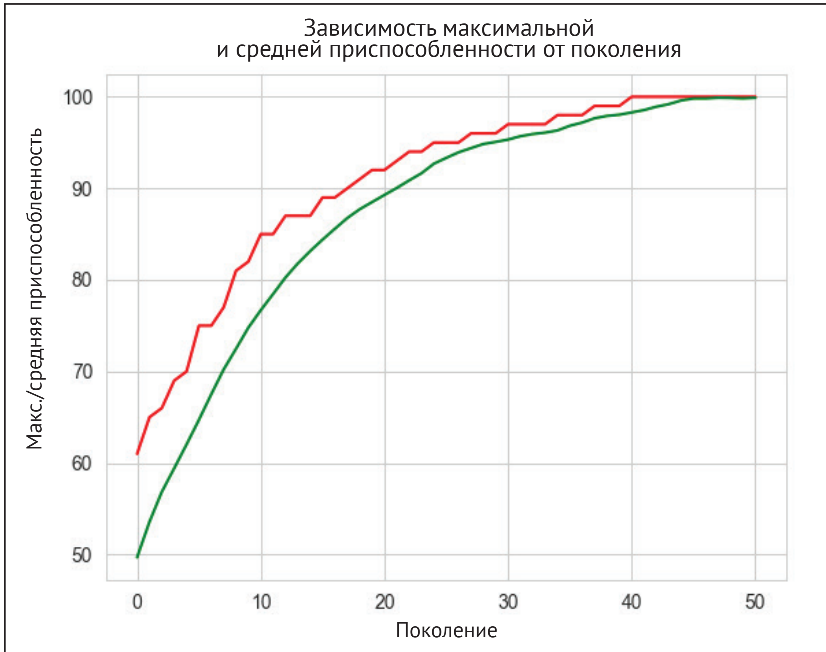
```
gen  nevals   max  avg
0    200      61   49.695
1    193      65   53.575

...

39   192      99   98.04
40   173      100  98.29
...
49   187      100  99.83
50   184      100  99.89
```

Она автоматически генерируется методом `algorithms.eaSimple` в соответствии с переданным ему объектом статистики, если аргумент `verbose` равен `True`.

Численно результаты похожи на те, что печатались в предыдущей версии программы, с двумя отличиями:

- в распечатке имеется строка для поколения 0, раньше мы ее не включали;
- алгоритм работает на протяжении всех 50 поколений, поскольку это было единственное условие остановки, тогда как в первоначальной программе имелось дополнительное условие – обнаружение наилучшего решения (известного заранее), – благодаря которому цикл остановился на 40-м поколении.

На графике наблюдается то же поведение, что и раньше. От предыдущего графика новый отличается тем, что продолжается до 50-го поколения, хотя наилучший результат получен уже в 40-м поколении.



Статистика программы для решения задачи OneMax
с помощью встроенного алгоритма

Начиная с 40-го поколения максимальное значение приспособленности перестает изменяться, а среднее продолжает расти, пока в конце концов не станет почти равным максимальному. Это означает, что в конце прогона почти все индивидуумы стали равны лучшему.

Зал славы

У встроенного метода `algorithms.eaSimple` есть еще одна возможность – **зал славы** (hall of fame, сокращенно **hof**). Класс `HallOfFame`, находящийся в модуле `tools`, позволяет сохранить лучших индивидуумов, встретившихся в процессе эволюции, даже если вследствие отбора, скрещивания и мутации они были в какой-то момент утрачены. Зал славы поддерживается в отсортированном состоянии, так что первым элементом всегда является индивидуум с наилучшим встретившимся значением приспособленности.

Полный код описанной в этом разделе программы находится по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter03/03-OneMax-short-hof.py.

Чтобы добавить зал славы, в написанный выше код нужно внести следующие изменения.

1. Определим константу, равную количеству индивидуумов, которых мы хотим хранить в зале славы:

   ```
   HALL_OF_FAME_SIZE = 10
   ```

2. Прежде чем вызывать алгоритм `eaSimple`, создадим объект `HallOfFame` такого размера:

   ```
   hof = tools.HallOfFame(HALL_OF_FAME_SIZE)
   ```

3. Объект `HallOfFame` передается алгоритму `eaSimple`, который самостоятельно обновляет его в процессе выполнения генетического алгоритма:

   ```
   population, logbook = algorithms.eaSimple(population, toolbox,
   cxpb=P_CROSSOVER, mutpb=P_MUTATION, ngen=MAX_GENERATIONS,
   stats=stats, halloffame=hof, verbose=True)
   ```

4. По завершении алгоритма атрибут `items` объекта `HallOfFame` можно использовать для доступа к списку помещенных в зал славы индивидуумов:

   ```
   print("Индивидуумы в зале славы  = ", *hof.items, sep="\n")
   print("Лучший индивидуум = ", hof.items[0])
   ```

Ниже приведены напечатанные результаты – лучший индивидуум, содержащий только единицы, а после него несколько индивидуумов, содержащих 0 в разных позициях:

```
Индивидуумы в зале славы =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
...
```

Лучшим является все тот же индивидуум, содержащий одни единицы:

```
Лучший индивидуум = [1, 1, 1, 1, 1, ...
```

Далее мы во всех программах будем использовать эти средства: объекты `Statistics` и `logbook`, встроенный алгоритм и зал славы.

Эксперименты с параметрами алгоритма

Теперь мы можем поэкспериментировать с различными параметрами и определениями и понаблюдать за изменениями поведения и результатов.

В каждом из последующих подразделов мы будем вносить в исходные параметры программы одно или несколько изменений. Призываем вас поэкспериментировать самостоятельно и попробовать подвергнуть программу сразу нескольким модификациям.

Имейте в виду, что последствия изменений могут зависеть от конкретной программы, и то, что мы видим в простой задаче OneMax, может не наблюдаться в других задачах.

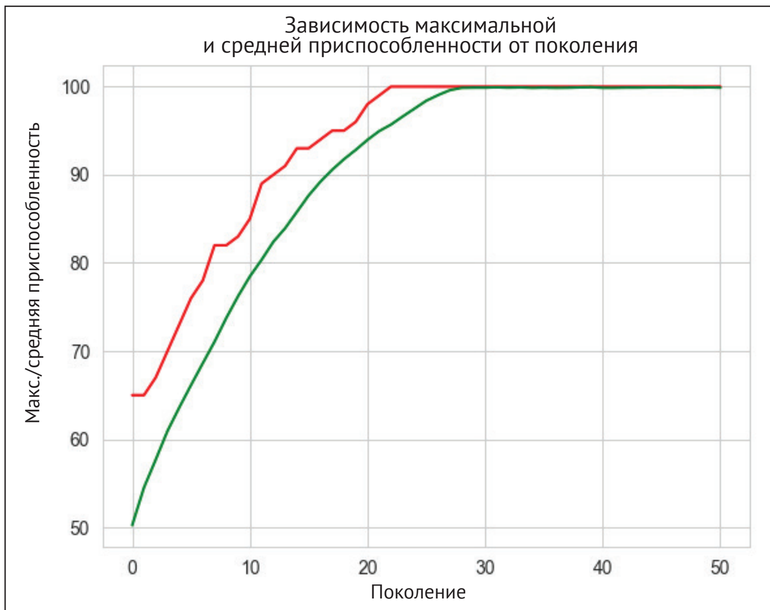
Размер популяции и количество поколений

Для начала изменим размер популяции и количество поколений в генетическом алгоритме.

1. Размер популяции определяется константой `POPULATION_SIZE`. Увеличим ее с 200 до 400:

```
POPULATION_SIZE = 400
```

При этом алгоритм работает быстрее, лучшее решение обнаруживается после 22 поколений, как видно из графика.

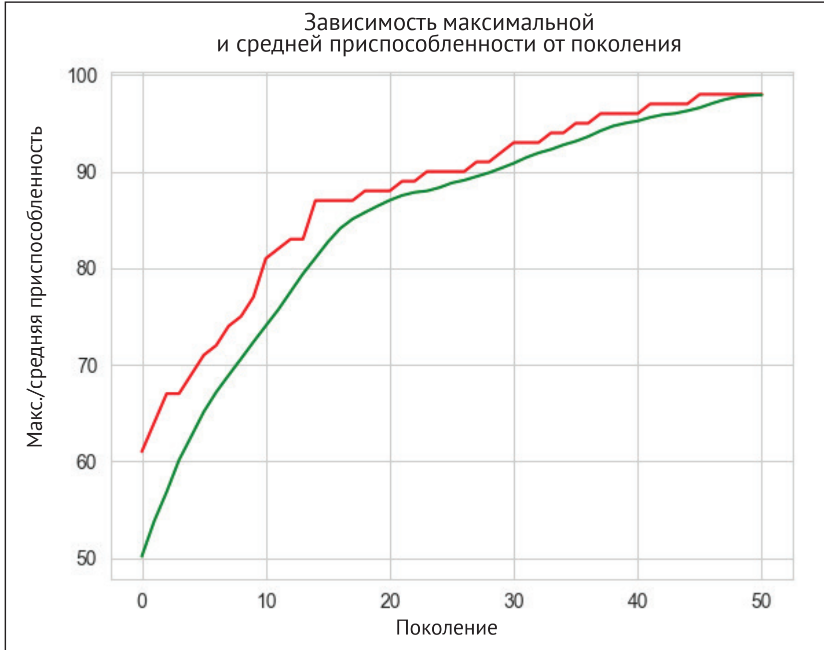


Статистика программы для решения задачи OneMax
после увеличения размера популяции до 400

2. Теперь попробуем уменьшить размер популяции до 100:

```
POPULATION_SIZE = 100
```

Сходимость алгоритма замедляется, на протяжении 50 поколений наилучшее решение так и не найдено.

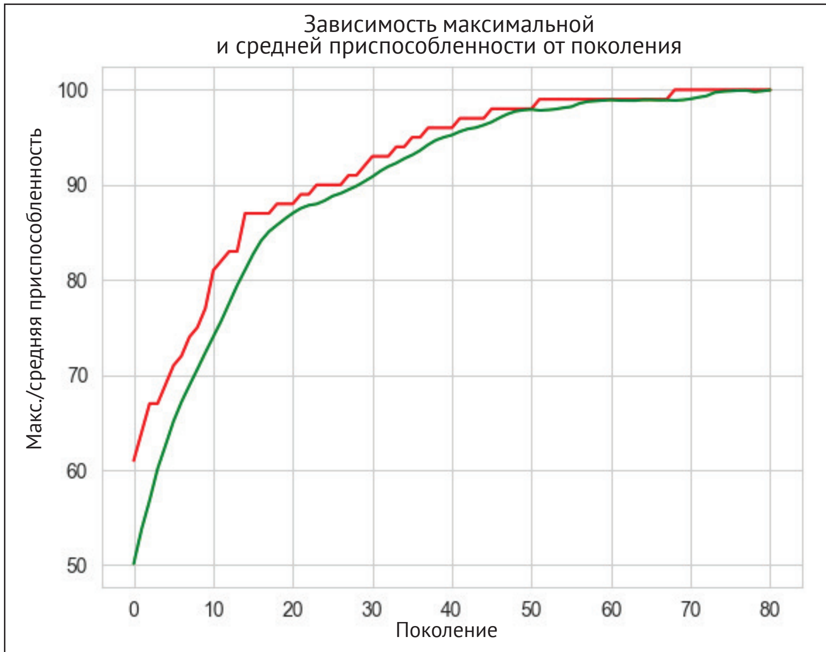


Статистика программы для решения задачи OneMax
после уменьшения размера популяции до 100

3. Для компенсации попробуем увеличить значение `MAX_GENERATIONS` до 80:

```
MAX_GENERATIONS = 80
```

Лучшее решение найдено после 68 поколений.



Статистика программы для решения задачи OneMax
после увеличения количества поколений до 80

Такое поведение типично для генетических алгоритмов – при увеличении популяции для нахождения решения требуется меньше поколений. Однако с ростом размера популяции повышаются требования к вычислительной мощности и памяти, поэтому обычно мы стремимся найти умеренный размер популяции, при котором алгоритм сходится к решению за разумное время.

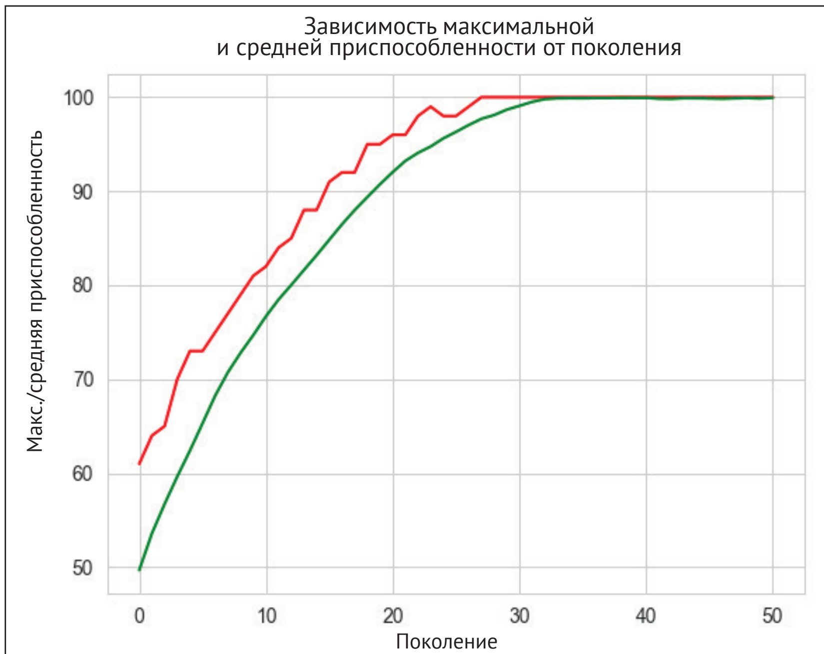
Оператор скрещивания

Откатим изменения и вернемся к первоначальным параметрам (50 поколений, размер популяции 200). Займемся экспериментами с оператором скрещивания, который отвечает за создание потомков.

Для замены одноточечного скрещивания двухточечным нужно лишь определить оператор скрещивания следующим образом:

```
toolbox.register("mate", tools.cxTwoPoint)
```

Теперь алгоритм находит лучшее решение на протяжении всего 27 поколений.



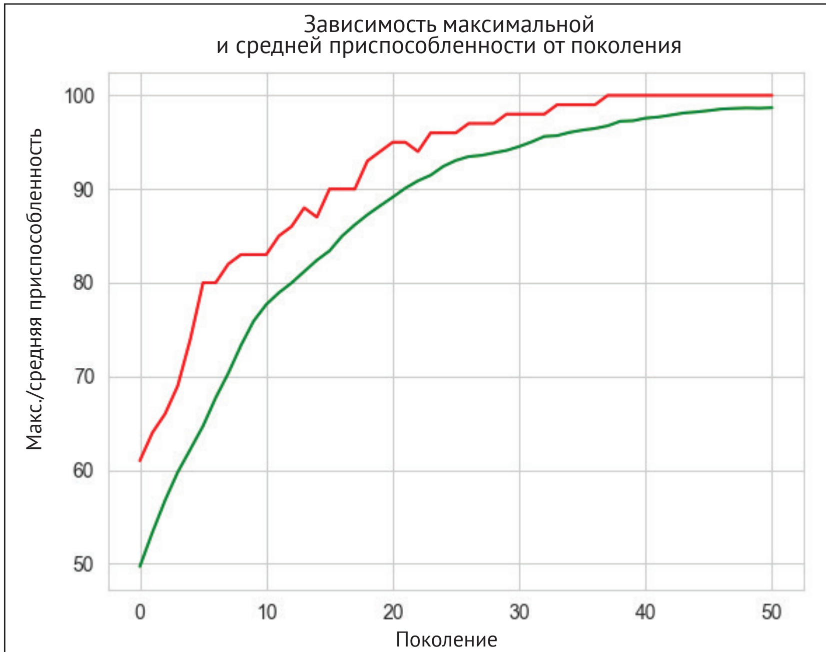
Статистика программы для решения задачи OneMax
после перехода на двухточечное скрещивание

Такое поведение типично для генетических алгоритмов с представлением хромосом двоичными строками, поскольку двухточечное скрещивание – более гибкий способ смешивания генов родителей по сравнению с одноточечным.

Оператор мутации

Снова откатим изменения, поскольку собираемся поэкспериментировать с оператором мутации, отвечающим за случайную модификацию потомка.

1. Для начала увеличим константу P_MUTATION до 0.9. Тогда график станет таким:



Статистика программы для решения задачи OneMax
после увеличения вероятности мутации до 0.9

На первый взгляд, результат удивительный, потому что при увеличении частоты мутаций алгоритм должен вести себя спорадически, но здесь этот эффект незаметен. Вспомним, однако, что у нашего алгоритма есть еще один параметр, относящийся к мутации, indpb – аргумент конкретного использованного в данном случае оператора мутации mutFlipBit:

```
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/ONE_MAX_LENGTH)
```

Если P_MUTATION определяет вероятность мутации индивидуума, то indpb задает вероятность инвертирования каждого бита данного индивидуума. В нашей программе значение 1.0/ONE_MAX_LENGTH, т. е. в среднем в хромосоме мутирует один ген. По-видимому, в задаче OneMax с длиной индивидуума 100 это ограничивает эффект мутации независимо от значения P_MUTATION.

2. Увеличим значение indpb в десять раз:

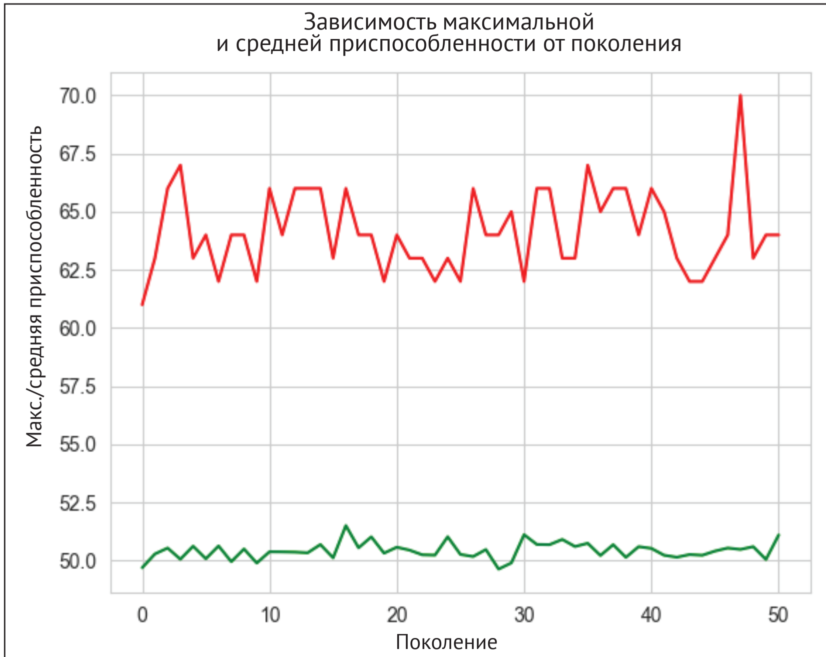
```
toolbox.register("mutate", tools.mutFlipBit, indpb=10.0/ONE_MAX_LENGTH)
```

В результате поведение алгоритма становится спорадическим, как видно из следующего графика.

Статистика программы для решения задачи OneMax
после десятикратного увеличения вероятности мутации одного бита

Мы видим, что в начале работы алгоритму удается улучшать результаты, но довольно быстро он начинает осциллировать без видимого прогресса.

3. Если еще увеличить `indpb`, до `50.0/ONE_MAX_LENGTH`, то алгоритм становится совсем неустойчивым:



Статистика программы для решения задачи OneMax
после пятидесятикратного увеличения вероятности мутации одного бита

Из графика видно, что генетический алгоритм превратился, по сути дела, в случайный поиск – быть может, ему повезет наткнуться на лучшее решение, но никакого уверенного продвижения в сторону хороших решений не наблюдается.

Оператор отбора

Далее займемся оператором отбора. Сначала изменим размер турнира, чтобы посмотреть на совместный эффект этого параметра и вероятности мутации. А затем посмотрим, что будет, если заменить турнирный отбор правилом рулетки.

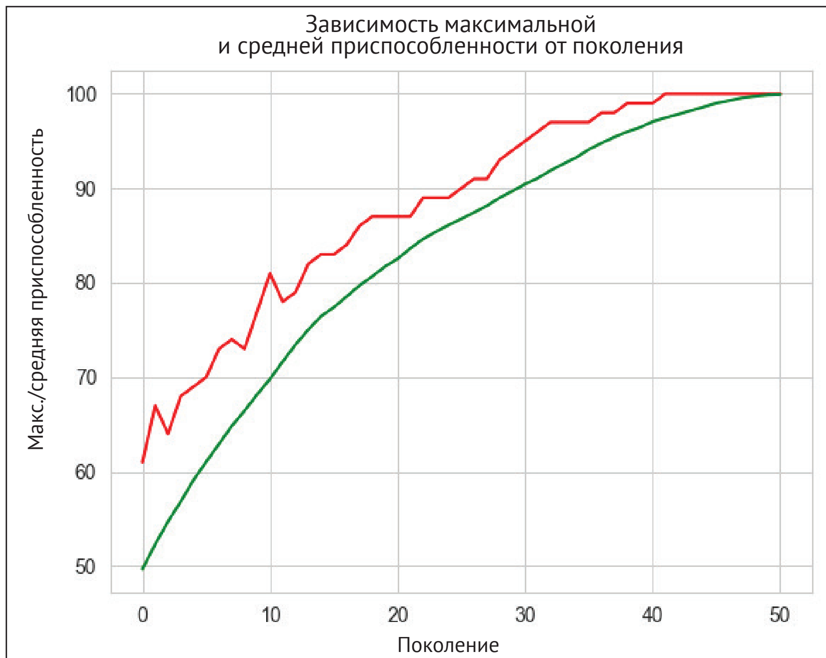
Размер турнира и его связь с вероятностью мутации

Снова восстановим первоначальные параметры программы, а затем внесем новые изменения.

1. Сначала сделаем параметр `tournamentSize` равным 2 (изначально он был равен 3):

```
toolbox.register("select", tools.selTournament, tournsize=2)
```

Не похоже, чтобы это как-то повлияло на поведение алгоритма.

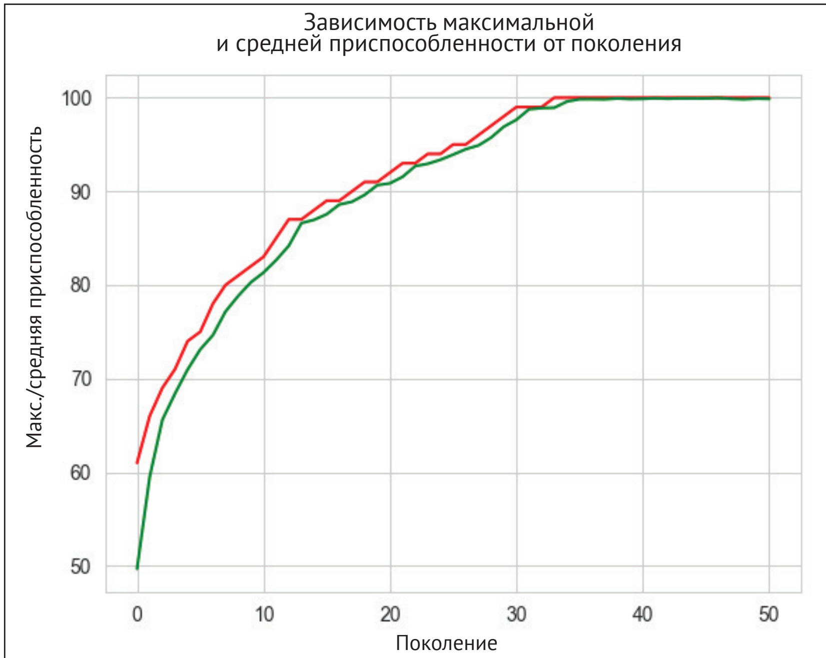


Статистика программы для решения задачи OneMax
после уменьшения размера турнира до 2

2. А что, если резко увеличить размер турнира, скажем, до 100? Поглядим:

```
toolbox.register("select", tools.selTournament, tournsize=100)
```

Алгоритм по-прежнему ведет себя хорошо и находит лучшее решение меньше, чем за 40 поколений. Правда, теперь графики максимальной и средней приспособленности очень похожи:



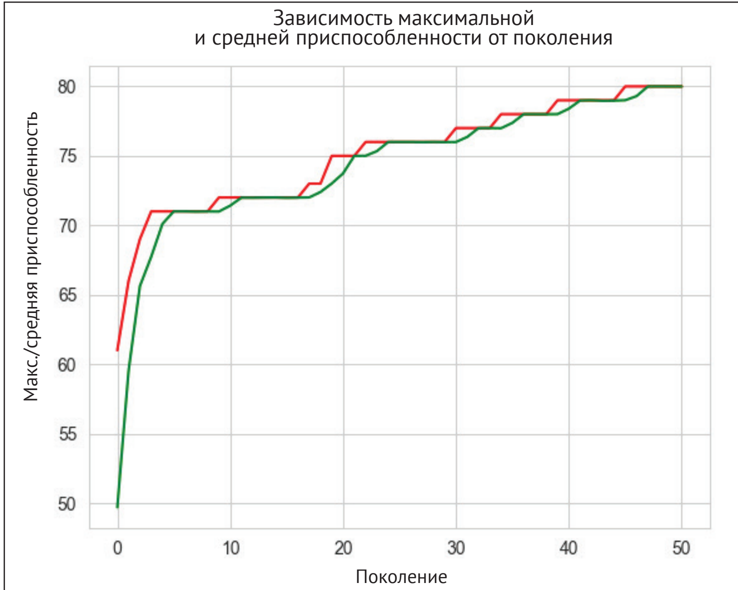
Статистика программы для решения задачи OneMax
после увеличения размера турнира до 100

Такое поведение объясняется тем, что при увеличении размера турнира шансы слабых индивидуумов отобраться падают, а лучшие решения занимают всю популяцию. В реальной задаче такое доминирование может привести к тому, что вся популяция будет насыщена неоптимальными решениями, а лучшее так и не будет найдено (этот феномен называется преждевременной сходимостью). Но в простой задаче OneMax подобной проблемы, похоже, не возникает. Возможно, это связано с тем, что оператор мутации привносит достаточно разнообразия, чтобы поиск решений продвигался в правильном направлении.

3. Чтобы проверить это объяснение, уменьшим вероятность мутации в 10 раз, до 0.01:

```
P_MUTATION = 0.01
```

Ели теперь прогнать алгоритм, то окажется, что результаты перестают улучшаться вскоре после запуска, а затем улучшаются гораздо медленнее и неравномерно. Итоговый результат гораздо хуже, чем после предыдущего прогона.



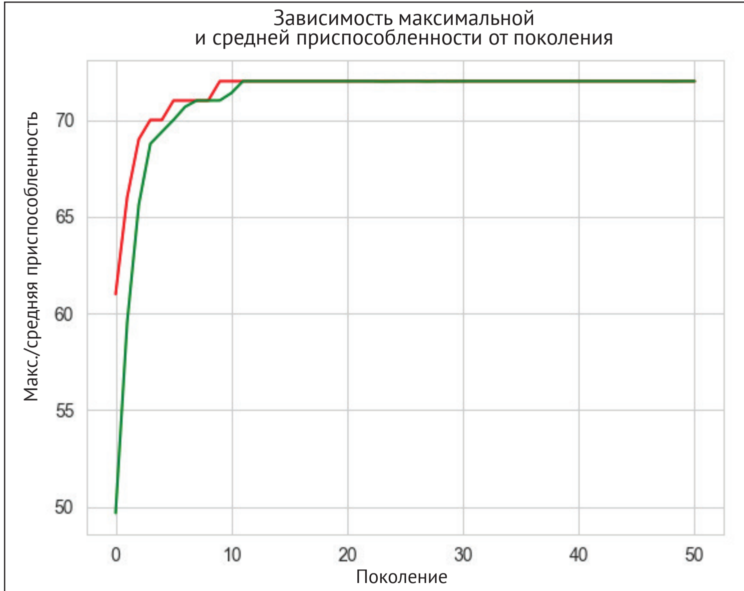
Статистика программы для решения задачи OneMax
с размером турнира 100 и вероятностью мутации 0.01

Это можно интерпретировать следующим образом: из-за большого размера турнира лучшие индивидуумы из начальной популяции занимают всю популяцию после небольшого числа смены поколений, и в течение этого времени мы наблюдаем быстрый рост обеих кривых. Но потом лишь случайная мутация в нужном направлении – инвертирование нулевого бита в единичный – приводит к появлению лучшего индивидуума, этому соответствуют редкие ступеньки на красном графике. Вскоре после этого найденный индивидуум снова занимает всю популяцию, и тогда зеленый график догоняет красный.

4. Чтобы довести эту ситуацию до крайности, уменьшим частоту мутаций еще в 10 раз:

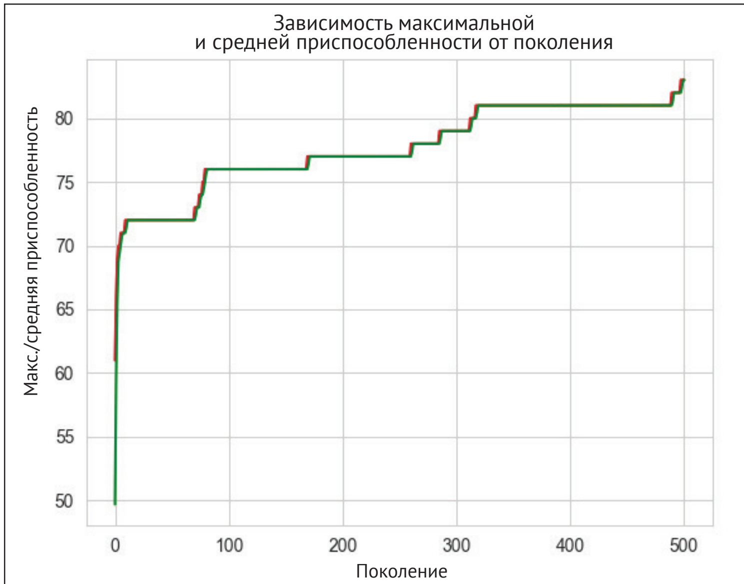
```
P_MUTATION = 0.001
```

Мы наблюдаем в целом такое же поведение, но, поскольку мутации случаются очень редко, улучшений немного и расстояние между ними больше.



Статистика программы для решения задачи OneMax
с размером турнира 100 и вероятностью мутации 0.001

5. Если увеличить количество поколений до 500, то это поведение станет еще нагляднее.

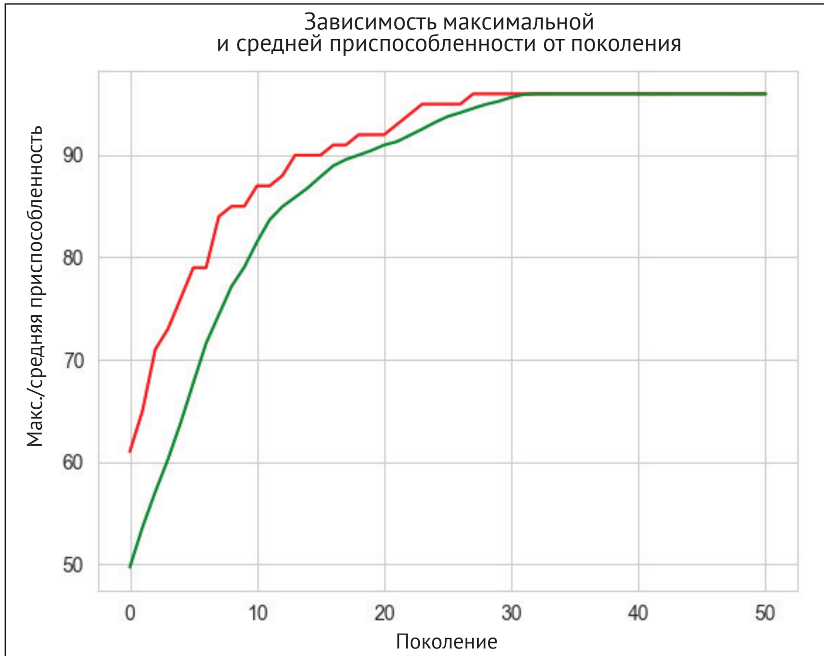


Статистика программы для решения задачи OneMax
с размером турнира 100 и вероятностью мутации 0.001
на протяжении 500 поколений

6. Просто из любопытства вернем размер турнира 3 и сделаем количество поколений равным 50, а низкую частоту мутаций оставим:

```
MAX_GENERATIONS = 50
toolbox.register("select", tools.selTournament, tournsize=3)
```

Получившийся график гораздо ближе к исходному:



Статистика программы для решения задачи OneMax
с размером турнира 3 и вероятностью мутации 0.001 на протяжении 50 поколений

Похоже, что теперь захват популяции тоже имеет место, но происходит гораздо позже, в районе поколения 30, когда наилучшая приспособленность близка к максимальному значению 100. Но более разумная частота мутаций позволила бы найти наилучшее решение, как то было при первоначальных параметрах.

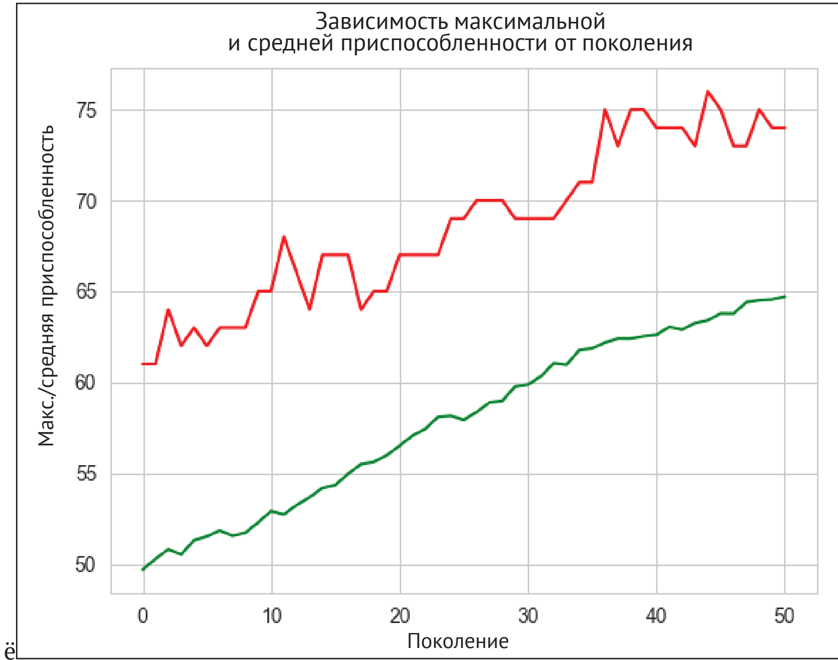
Отбор по правилу рулетки

Снова вернемся к первоначальным параметрам в предвкушении последнего эксперимента. Теперь мы заменим турнирный отбор отбором по правилу рулетки, описанным в главе 2. Делается это так:

```
toolbox.register("select", tools.selRoulette)
```

Это изменение плохо повлияло на результаты алгоритма. На графике видно, что есть много точек, в которых наилучшее решение оказывается забыто

и максимальная приспособленность уменьшается, по крайней мере временно, хотя средняя приспособленность монотонно возрастает. Это объясняется тем, что правило рулетки отбирает индивидуумов с вероятностью, пропорциональной их приспособленности; если различия между индивидуумами невелики, то у более слабых появляется больше шансов отобраться по сравнению с турнирным отбором.



Статистика программы при использовании правила рулетки

Для компенсации этого поведения можно применить элитистский подход, описанный в главе 2. Тогда заданное количество лучших индивидуумов перейдет из текущего поколения в следующее без изменения, поэтому они не будут потеряны. В следующей главе мы исследуем применение элитистского подхода при работе с DEAP.

Резюме

В этой главе мы познакомились с DEAP – гибким каркасом эволюционных вычислений, который будет далее использоваться в книге для решения реальных задач с помощью генетических алгоритмов. Мы узнали о модулях DEAP `creator` и `toolbox` и о том, как они применяются для создания различных компонентов генетических алгоритмов. Затем воспользовались DEAP для написания двух версий Python-программы, решающей задачу OneMax. Первую версию мы реализовали сами от начала до конца, а во второй взяли

встроенные в каркас алгоритмы. В третьей версии был продемонстрирован **зал славы** (**HOF**) – еще одна возможность DEAP. Далее мы поэкспериментировали с параметрами генетического алгоритма, выяснили, как влияет на результаты размер популяции, а также замена операторов отбора, скрещивания и мутации.

В следующей главе мы продолжим начатое и займемся решением реальных комбинаторных задач, в т. ч. задач коммивояжера и маршрутизации транспорта.

Для дальнейшего чтения

За дополнительной информацией обратитесь к следующим ресурсам:

- документация по DEAP: https://deap.readthedocs.io/en/master/;
- исходный код DEAP на GitHub: https://github.com/DEAP/deap.

Глава 4

Комбинаторная оптимизация

В этой главе мы узнаем о том, как использовать генетические алгоритмы в задачах комбинаторной оптимизации. Мы начнем с постановки задач поиска и комбинаторной оптимизации и приведем несколько практических примеров таких задач. Затем проанализируем их и напишем Python-программы для их решения с помощью каркаса DEAP. Будут рассмотрены хорошо известные задачи о рюкзаке, **коммивояжера** и **маршрутизации транспорта**. На закуску мы обсудим вопрос об отображении между генотипом и фенотипом и дилемму исследования–использования.

В этой главе мы:

- разберемся в природе поисковых задач и комбинаторной оптимизации;
- решим задачу о рюкзаке, применив генетический алгоритм, написанный с помощью каркаса DEAP;
- рассмотрим решение задачи коммивояжера с помощью генетического алгоритма и каркаса DEAP;
- решим задачу маршрутизации транспорта с помощью генетического алгоритма и каркаса DEAP;
- узнаем, что такое отображение между генотипом и фенотипом;
- познакомимся с дилеммой исследования–использования и ее связью с элитизмом.

Технические требования

В этой главе нам понадобится Python 3 и следующие библиотеки:

- `deap`;
- `numpy`;
- `matplotlib`;
- `seaborn`.

Кроме того, мы будем использовать эталонные данные из задачи о рюкзаке 0-1 с сайта Rosetta Code (http://rosettacode.org/wiki/Knapsack_problem/0-1)

и библиотеки для задачи коммивояжера TSP (http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/).

Код приведенных в этой главе программ можно найти в репозитории книги на GitHub по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter04. Чтобы увидеть код в действии, посмотрите видео по адресу http://bit.ly/2tfyfMI.

Поисковые задачи и комбинаторная оптимизация

Одна из основных областей применений генетических алгоритмов – поисковые задачи, которые находят широкое применение в логистике, исследовании операций, искусственном интеллекте и машинном обучении. Примерами могут служить нахождение оптимального маршрута доставки посылок, проектирование сетей транспортных авиалиний с хабами, управление портфельными инвестициями и распределение пассажиров между свободными машинами такси.

Поисковый алгоритм решает задачу путем методичного перебора состояний и переходов состояний с целью найти путь из начального состояния в конечное (или целевое). Обычно с каждым переходом состояний ассоциированы некоторые затраты (стоимость) или доходы (выигрыш), а цель поискового алгоритма – найти путь, минимизирующий затраты или максимизирующий выигрыш. Поскольку оптимальный путь – один из многих возможных, такой поиск неразрывно связан с комбинаторной оптимизацией – нахождением оптимального объекта в конечном, но очень большом множестве.

Эти идеи будут проиллюстрированы в следующем разделе при рассмотрении задачи о рюкзаке.

Решение задачи о рюкзаке

Вспомните знакомую ситуацию – сборы перед длительной поездкой. Хочется взять с собой много всего, но место в чемодане ограничено. Мысленно вы приписываете ценность каждому предмету, но у каждого предмета есть также размер (и вес), так что все они конкурируют между собой за место в чемодане. Это одна из многих жизненных ситуаций, в которых возникает задача о рюкзаке, считающаяся самой старой и хорошо изученной задачей комбинаторной оптимизации.

Формально в задаче о рюкзаке имеются следующие компоненты:

- набор предметов, каждый из которых имеет ценность и вес;
- рюкзак (сумка, контейнер), в который можно поместить предметы ограниченного суммарного веса.

Наша цель – отобрать группу предметов максимальной суммарной ценности, так чтобы суммарный вес не превышал емкость контейнера.

В контексте алгоритмов поиска каждый набор предметов представляет *состояние*, а множество всех возможных наборов считается *пространством состояний*. Например, в задаче о рюкзаке 0-1 с n предметами размер пространства состояний равен 2^n и растет очень быстро даже при сравнительно небольших n.

В этой (классической) постановке каждый предмет либо включается ровно один раз, либо не включается вообще, поэтому иногда говорят о рюкзаке 0-1. Но возможны и другие постановки, например каждый предмет может включаться несколько раз (ограниченное или неограниченное количество) или имеется несколько рюкзаков разной емкости.

Задача о рюкзаке возникает во многих реальных процессах, где требуется принимать решения, связанные с выделением ресурсов, например: состав активов в инвестиционном портфеле, минимизация отходов при распиле, получение максимального количества баллов при решении о том, на какие вопросы отвечать в тесте с ограниченным временем.

Чтобы лучше почувствовать задачу, рассмотрим широко известный пример.

Задача о рюкзаке 0-1 с сайта Rosetta Code

На сайте Rosetta Code (`rosettacode.org`) предлагается набор задач по программированию с решениями на разных языках. Одна из них, размещенная по адресу `rosettacode.org/wiki/Knapsack_problem/0-1`, – задача о рюкзаке 0-1, в которой турист должен решить, какие предметы взять с собой в недельную поездку. Всего имеется 22 предмета, каждому из которых турист назначает некоторую ценность, отражающую его важность в путешествии.

В этой задаче максимальная емкость чемодана составляет 400. Список предметов с указанием ценности и веса приведен ниже.

Предмет	Вес	Ценность
Карта	9	150
Компас	13	35
Вода	153	200
Сэндвич	50	160
Глюкоза	15	60
Кружка	68	45
Банан	27	60
Яблоко	39	40
Сыр	23	30
Пиво	52	10
Крем от загара	11	70
Камера	32	30
Футболка	24	15
Брюки	48	10
Зонтик	73	40

Предмет	Вес	Ценность
Непромокаемые штаны	42	70
Непромокаемый плащ	43	75
Бумажник	22	80
Солнечные очки	7	20
Полотенце	18	12
Носки	4	50
Книга	30	10

Прежде чем приступать к решению, нужно обсудить важный вопрос – как мы будем представлять потенциальное решение?

Представление решения

Самый очевидный способ представить решение задачи о рюкзаке 0-1 – воспользоваться списком двоичных значений. Каждый элемент списка соответствует одному предмету. Следовательно, в задаче с сайта Rosetta Code решение будет представлено списком 22 целых чисел, принимающих значения 0 или 1. Значение 1 означает, что турист взял соответствующий предмет, а значение 0 – что оставил. В генетическом алгоритме этот список двоичных значений будет играть роль хромосомы.

Но следует помнить, что суммарный вес выбранных предметов не должен быть больше емкости рюкзака. Один из способов учесть это ограничение – отложить решение до момента оценки. В процессе вычисления оценки мы складываем веса выбранных предметов по одному, игнорируя предметы, которые привели бы к перевесу. С точки зрения генетического алгоритма, это означает, что представление индивидуума в виде хромосомы (**генотип**) может оказаться недостаточным при переходе к фактическому решению (**фенотипу**), поскольку некоторые единичные гены хромосомы игнорируются. Иногда этот переход называют *отображением генотипа в фенотип*.

В следующем разделе мы реализуем описанное представление в классе Python.

Представление задачи на Python

Для инкапсуляции данных в задаче о рюкзаке 0-1 создадим класс Python `Knapsack01Problem`. Его код находится в файле `knapsack.py` по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/knapsack.py.

Класс предоставляет следующие методы:

- `__init_data()`: инициализирует данные, создавая список кортежей. Каждый кортеж содержит название предмета, его вес и ценность;

- getValue(zeroOneList): вычисляет суммарную ценность всех помещенных в список предметов, игнорируя те, которые привели бы к перевесу;
- printItems(zeroOneList): печатает выбранные предметы, игнорируя те, которые привели бы к перевесу.

Метод main класса создает экземпляр класса Knapsack01Problem. Затем он порождает случайное решение и печатает информацию о нем. Если выполнить этот класс как автономную Python-программу, то результат будет выглядеть следующим образом:

```
Случайное решение =
[1 1 1 1 1 0 0 0 0 1 1 1 0 1 0 0 0 1 0 0 0 0]
- Добавляется карта: вес = 9, ценность = 150, вес нарастающим итогом = 9, ценность
нарастающим итогом = 150
- Добавляется компас: вес = 13, ценность = 35, вес нарастающим итогом = 22,
ценность нарастающим итогом = 185
- Добавляется вода: вес = 153, ценность = 200, вес нарастающим итогом = 175,
ценность нарастающим итогом = 385
- Добавляется сэндвич: вес = 50, ценность = 160, вес нарастающим итогом = 225,
ценность нарастающим итогом = 545
- Добавляется глюкоза: вес = 15, ценность = 60, вес нарастающим итогом = 240,
ценность нарастающим итогом = 605
- Добавляется пиво: вес = 52, ценность = 10, вес нарастающим итогом = 292,
ценность нарастающим итогом = 615
- Добавляется крем от загара: вес = 11, ценность = 70, вес нарастающим итогом = 303,
ценность нарастающим итогом = 685
- Добавляется камера: вес = 32, ценность = 30, вес нарастающим итогом = 335,
ценность нарастающим итогом = 715
- Добавляется брюки: вес = 48, ценность = 10, вес нарастающим итогом = 383,
ценность нарастающим итогом = 725
- Суммарный вес = 383, Суммарная ценность = 725
```

Заметим, что последнее вхождение 1 в случайное решение, соответствующее бумажнику, пало жертвой отображения генотипа в фенотип, о котором было сказано в предыдущем разделе. Поскольку вес этого предмета равен 22, суммарный вес превысил бы 400, поэтому он и не был включен в состав решения.

Понятно, что это случайное решение далеко от оптимальности. Попробуем найти оптимальное решение задачи, воспользовавшись генетическим алгоритмом.

Решение с помощью генетического алгоритма

Для решения задачи о рюкзаке 0-1 мы написали на Python программу, находящуюся в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/01-solve-knapsack.py.

Напомним, что представлением хромосомы является список целых чисел, принимающих значение 0 или 1. Поэтому задача принимает вид, похожий на задачу OneMax, решенную в предыдущей главе. Генетическому алгоритму все равно, что именно (какой фенотип) представляет хромосома – список

подлежащих упаковке предметов, коэффициент булева уравнения или просто какое-то двоичное число, – его интересует только сама хромосома (генотип) и ее приспособленность. Отображение хромосомы на представляемое ей решение производится функцией приспособленности, реализованной вне генетического алгоритма. В нашем случае это отображение хромосомы и вычисление приспособленности реализованы методом `getValue()` класса `Knapsack01Problem`.

Это означает, что мы можем воспользоваться той же реализацией генетического алгоритма, что и для задачи OneMax, внеся лишь несколько модификаций.

Ниже описаны основные моменты нашего решения.

1. Сначала создаем экземпляр решаемой задачи о рюкзаке:

    ```
    knapsack = knapsack.Knapsack01Problem()
    ```

2. Затем просим генетический алгоритм вызвать метод `getValue()` этого экземпляра для вычисления приспособленности:

    ```
    def knapsackValue(individual):
        return knapsack.getValue(individual),

    toolbox.register("evaluate", knapsackValue)
    ```

3. Используемые генетические операторы совместимы с представлением хромосомы списком двоичных значений:

    ```
    toolbox.register("select", tools.selTournament, tournsize=3)
    toolbox.register("mate", tools.cxTwoPoint)
    toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/len(knapsack))
    ```

4. По завершении генетического алгоритма вызываем метод `printItems()` для печати найденного лучшего решения:

    ```
    best = hof.items[0]

    print("-- Предметы в рюкзаке = ")
    knapsack.printItems(best)
    ```

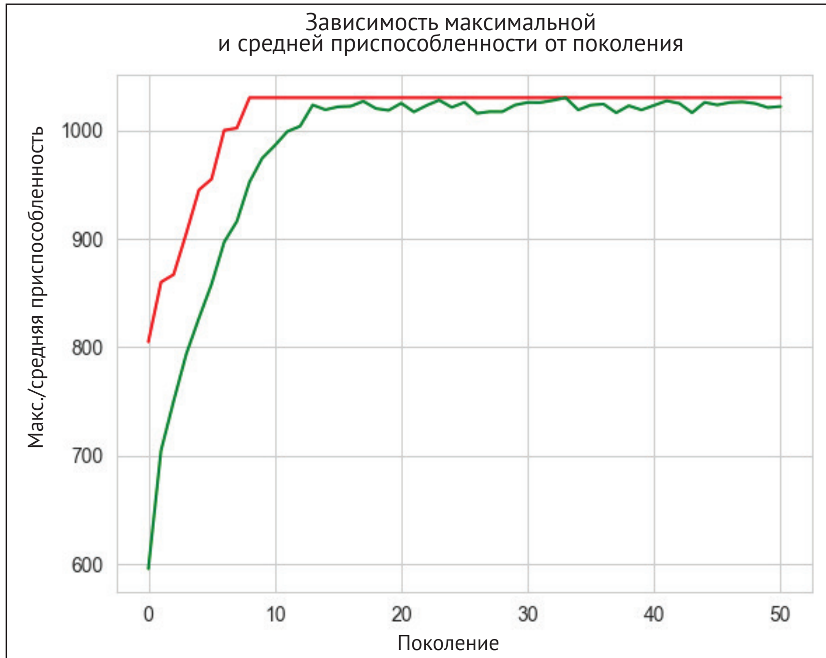
5. Можно также изменить некоторые параметры генетического алгоритма. Поскольку в этой задаче длина двоичной строки равна 22, можно надеяться, что она проще задачи OneMax со строкой длины 100, поэтому попробуем уменьшить размер популяции и максимальное количество поколений.

Прогон алгоритма с 50 поколениями и популяцией размера 50 дает следующий результат:

```
-- Лучший индивидуум = [1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0,
0, 0, 0, 1, 1, 1, 1, 0, 1, 1]
-- Лучшая приспособленность = 1030.0
-- Предметы в рюкзаке =
- Добавляется карта: вес = 9, ценность = 150, вес нарастающим итогом  = 9,
ценность нарастающим итогом = 150
- Добавляется  компас: вес = 13, ценность = 35, вес нарастающим итогом  =
22, ценность нарастающим итогом = 185
- Добавляется  вода: вес = 153, ценность = 200, вес нарастающим итогом  =
175, ценность нарастающим итогом = 385
- Добавляется  сэндвич: вес = 50, ценность = 160, вес нарастающим итогом
= 225, ценность нарастающим итогом = 545
- Добавляется  глюкоза: вес = 15, ценность = 60, вес нарастающим итогом  =
240, ценность нарастающим итогом = 605
- Добавляется  банан: вес = 27, ценность = 60, вес нарастающим итогом  =
267, ценность нарастающим итогом = 665
- Добавляется  крем от загара: вес = 11, ценность = 70, суммарный
вес = 278, ценность нарастающим итогом = 735
- Добавляется  непромокаемые штаны: вес = 42, ценность = 70,
вес нарастающим итогом  = 320, ценность нарастающим итогом = 805
- Добавляется  непромокаемый плащ: вес = 43, ценность = 75,
вес нарастающим итогом  = 363, ценность нарастающим итогом = 880
- Добавляется  бумажник: вес = 22, ценность = 80, вес нарастающим итогом
= 385, ценность нарастающим итогом = 960
- Добавляется  солнечные очки: вес = 7, ценность = 20, вес нарастающим итогом
= 392, ценность нарастающим итогом = 980
- Добавляется  носки: вес = 4, ценность = 50, вес нарастающим итогом  =
396, ценность нарастающим итогом = 1030
- Суммарный вес = 396, суммарная ценность = 1030
```

Известно, что 1030 – действительно оптимальное решение этой задачи. И снова мы видим, что последнее вхождение 1 в хромосому лучшего индивидуума, представляющее книгу, выпало из отображения на фактическое решение, чтобы суммарный вес не вышел за пределы 400.

Из графиков зависимости максимальной и средней приспособленности от номера поколения видно, что лучшее решение было найдено меньше, чем за 10 поколений.



Статистика программы для решения задачи о рюкзаке 0-1

В следующем разделе мы сменим тему и рассмотрим более сложную, но также классическую комбинаторную задачу коммивояжера.

Решение задачи коммивояжера

Допустим, вы управляете небольшим центром обработки и исполнения заказов и должны доставлять посылки заказчикам, располагая всего одним автомобилем. Как оптимально выбрать маршрут, чтобы объехать всех заказчиков и вернуться в исходную точку? Это пример классической задачи коммивояжера.

Задача коммивояжера восходит к 1930 году и является одной из наиболее изученных задач оптимизации. Зачастую она используется для оценки алгоритмов оптимизации. У этой задачи много вариантов, но первоначально она ставилась на примере коммивояжера, которому требуется объехать несколько городов:

> *Пусть дан список городов и известны расстояния между каждыми двумя городами. Найти кратчайший путь, проходящий через все города и возвращающийся в исходную точку.*

Легко показать, что количество возможных путей, проходящих через n городов, равно $(n - 1)!/2$.

На рисунке ниже показан кратчайший путь, проходящий через 15 крупнейших городов Германии.

Кратчайший маршрут коммивояжера,
проходящий через 15 крупнейших городов Германии

Источник: https://commons.wikimedia.org/wiki/File:TSP_Deutschland_3.png.
Автор Kapitan Nemo. Является общественным достоянием

Поскольку в этом случае $n = 15$, количество возможных путей равно $14!/2 = 43\,589\,145\,600$ – пугающее число.

В контексте алгоритмов поиска каждый путь (или частичный путь), проходящий через города, представляет *состояние*, а множество всех возможных путей рассматривается как *пространство состояний*. У каждого пути имеется стоимость – его длина, – и мы ищем путь минимальной стоимости.

Как уже отмечалось, пространство состояний очень велико даже при сравнительно небольшом количестве городов, поэтому рассмотреть каждый путь не представляется возможным. И хотя проложить какой-то путь через все города сравнительно просто, найти оптимальный путь очень трудно.

Файлы эталонных данных TSPLIB

Библиотека TSPLIB содержит примеры задач коммивояжера для реальных городов. Ее поддерживает Гейдельбергский университет по адресу http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/. Оптимальные пути для каждой задачи можно найти по адресу http://comopt.ifi.uniheidelberg.de/software/TSPLIB95/STSP.html.

Примеры из TSPLIB представлены текстовыми файлами с полями, разделенными пробелами. Типичный файл содержит несколько пояснительных строк, за которыми следуют данные о городах. Нас интересуют файлы, содержащие координаты *x*, *y* городов, поэтому мы можем нарисовать города. Например, ниже показан файл `burma14.tsp` (некоторые строки для краткости опущены):

```
NAME: burma14
TYPE: TSP
...
NODE_COORD_SECTION
   1  16.47       96.10
   2  16.47       94.44
   3  20.09       92.54
   ...
  12  21.52       95.59
  13  19.41       97.13
  14  20.09       94.55
EOF
```

Интерес представляют строки между `NODE_COORD_SECTION` и `EOF`. В некоторых файлах заголовок называется не `NODE_COORD_SECTION`, а `DISPLAY_DATA_SECTION`.

Готовы ли мы решить задачу? Сначала нужно договориться о представлении потенциального решения. Этим мы займемся в следующем разделе.

Представление решения

В задаче коммивояжера города обычно представляются числами от 0 до $n - 1$, а возможные решения – последовательностями таких чисел. Например, в задаче с пятью городами решения имеют вид [0, 1, 2, 3, 4], [2, 4, 3, 1, 0] и т. д. Каждое решение можно оценить, просуммировав расстояния между соседними в последовательности городами и прибавив еще расстояние между последним и первым городом. Такой список мы будем использовать для представления хромосомы в генетическом алгоритме.

Класс Python, описанный в следующем разделе, читает содержимое TSPLIB-файла и вычисляет все попарные расстояния между городами. Кроме того, он вычисляет полное расстояние в заданном решении, представленном описанным выше списком.

Представление задачи на Python

Для инкапсуляции данных задачи коммивояжера мы написали на Python класс `TravelingSalesmanProblem`. Его код находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/tsp.py.

Этот класс предоставляет следующие закрытые методы:

- `__create_data()`: читает указанный файл из интернета, извлекает координаты городов, вычисляет попарные расстояния между городами

и запоминает их в матрице расстояний (двумерном массиве). Затем сериализует местоположения городов и вычисленные расстояния в файле на диске с помощью встроенной утилиты `pickle`;
- `__read_data()`: читает сериализованные данные, а если их еще нет, вызывает `__create_data()` для их подготовки.

Эти методы вызываются из конструктора, поэтому данные инициализируются в момент создания экземпляра.

Кроме того, класс предоставляет следующие открытые методы:

- `getTotalDistance(indices)`: вычисляет полную длину пути, проходящего через города с указанными индексами;
- `plotData(indices)`: рисует путь, проходящий через города с указанными индексами.

Метод `main` класса вызывает вышеперечисленные методы. Сначала он создает экземпляр задачи `bayg29` (29 городов в Баварии), затем вычисляет длину оптимального решения (описанного в файле http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/bayg29.opt.tour) и, наконец, рисует его. Если запустить этот класс как автономную Python-программу, то будет напечатано:

```
Имя задачи: bayg29
Оптимальное решение = [0, 27, 5, 11, 8, 25, 2, 28, 4, 20, 1, 19, 9, 3, 14, 17,
13, 16, 21, 10, 18, 24, 6, 22, 7, 26, 15, 12, 23]
Оптимальное расстояние = 9074.147
```

График оптимального решения показан на рисунке ниже.

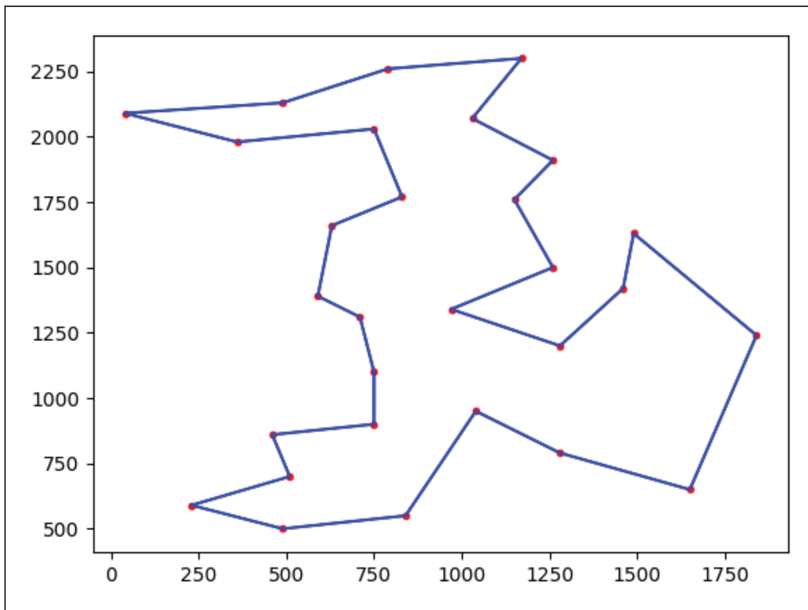


График оптимального решения задачи коммивояжера bayg29.
Красными точками обозначены города

Теперь попробуем найти это оптимальное решение с помощью генетического алгоритма.

Решение с помощью генетического алгоритма

Для первой попытки решить задачу коммивояжера с помощью генетического алгоритма мы написали на Python программу `02-solve-tsp-first-attempt.py`, находящуюся по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/02-solve-tsp-first-attempt.py.

Ниже описаны основные шаги решения.

1. Вначале программа создает экземпляр задачи `bayg29`:

   ```
   TSP_NAME = "bayg29"
   tsp = tsp.TravelingSalesmanProblem(TSP_NAME)
   ```

2. Затем нужно определить стратегию приспособления. Мы хотим минимизировать состояние, а это значит, что нужно определить минимизирующий класс `Fitness` с одной целью, в котором задан один отрицательный вес:

   ```
   creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
   ```

3. Мы уже говорили, что в роли хромосомы для генетического алгоритма выступает список целых чисел (индексов городов) от 0 до $n - 1$, где n – количество городов. Так, показанное выше оптимальное решение задачи `bayg29` представлено такой хромосомой:

   (0, 27, 5, 11, 8, 25, 2, 28, 4, 20, 1, 19, 9, 3, 14, 17, 13, 16, 21, 10, 18, 24, 6, 22, 7, 26, 15, 12, 23)

   Ниже приведен код, реализующий хромосому такого вида.

   ```
   creator.create("Individual", array.array, typecode='i',
   fitness=creator.FitnessMin)
   toolbox.register("randomOrder", random.sample, range(len(tsp)),
   len(tsp))
   toolbox.register("individualCreator", tools.initIterate,
   creator.Individual, toolbox.randomOrder)
   toolbox.register("populationCreator", tools.initRepeat, list,
   toolbox.individualCreator)
   ```

   Сначала создается класс `Individual`, расширяющий массив целых чисел и дополняющий его атрибутом типа `FitnessMin`.
   Затем регистрируется оператор `randomOrder`, который применяет функцию `random.sample()` к диапазону, соответствующему задаче коммивояжера (длины, равной количеству городов n). В результате генерируется случайный список индексов от 0 до $n - 1$.
   Далее регистрируется оператор `IndividualCreator`. Он вызывает оператор `randomOrder` и обходит созданный им список с целью создать хромосому, состоящую из индексов городов.

И напоследок регистрируется оператор populationCreator, который порождает список индивидуумов, вызывая в цикле оператор IndividualCreator.

4. Реализовав хромосому, мы можем определить функцию вычисления приспособленности. Для этого служит функция tspDistance(), которая вызывает метод getTotalDistance() класса TravelingSalesmanProblem:

```
def tpsDistance(individual):
    return tsp.getTotalDistance(individual), # return a tuple

toolbox.register("evaluate", tpsDistance)
```

5. Следующий шаг – определение генетических операторов. Мы будем использовать турнирный отбор с турниром размера 3, как и в предыдущих примерах:

```
toolbox.register("select", tools.selTournament, tournsize=3)
```

6. Но что касается операторов скрещивания и мутации, нужно иметь в виду, что теперь хромосома – не просто список целых чисел, а список индексов, представляющий порядок посещения городов, поэтому мы не вправе перемешивать части двух списков или произвольно менять индекс. Необходимо использовать специализированные операторы, предназначенные для порождения допустимых списков индексов. В главе 2 мы рассматривали один такой оператор – упорядоченное скрещивание. Ниже используется реализация этого оператора в каркасе DEAP:

```
toolbox.register("mate", tools.cxOrdered)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=1.0/len(tsp))
```

7. Вот и настало время реализовать сам генетический алгоритм. Мы воспользуемся встроенным в DEAP алгоритмом eaSimple и его объектами статистики и залом славы, которые дают информацию, необходимую для последующего отображения результатов:

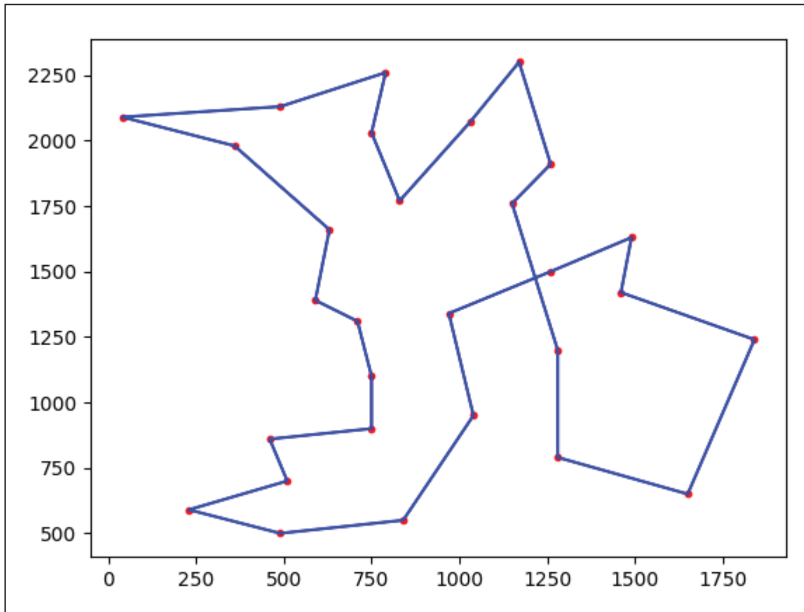
```
population, logbook = algorithms.eaSimple(population, toolbox,
                                          cxpb=P_CROSSOVER,
                                          mutpb=P_MUTATION,
                                          ngen=MAX_GENERATIONS,
                                          stats=stats,
                                          halloffame=hof,
                                          verbose=True)
```

Выполнение этой программы с заданными в начале файла константами (размер популяции 300, 200 поколений, вероятность скрещивания 0.9, вероятность мутации 0.1) дает такие результаты:

```
-- Лучший индивидуум = Individual('i', [0, 27, 11, 5, 20, 4, 8, 25, 2,
28, 1, 19, 9, 3, 14, 17, 13, 16, 21, 10, 18, 12, 23, 7, 26, 22, 6, 24, 15])
-- Лучшая приспособленность = 9549.9853515625
```

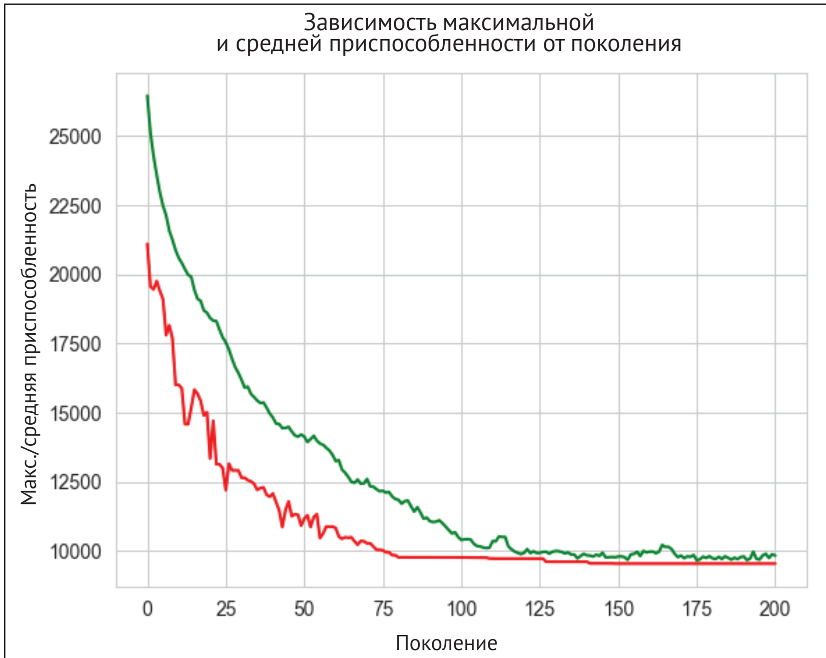
Лучшая найденная приспособленность (9549.98) не сильно отличается от известного оптимального расстояния 9074.14.

Затем программа рисует два графика. На первом показан лучший найденный путь.



Лучшее решение,
найденное в ходе первой попытки решить задачу bayg29

На втором графике показана статистика работы генетического алгоритма. На этот раз мы стремились минимизировать, а не максимизировать приспособленность, поскольку наша цель – нахождение наименьшего расстояния.

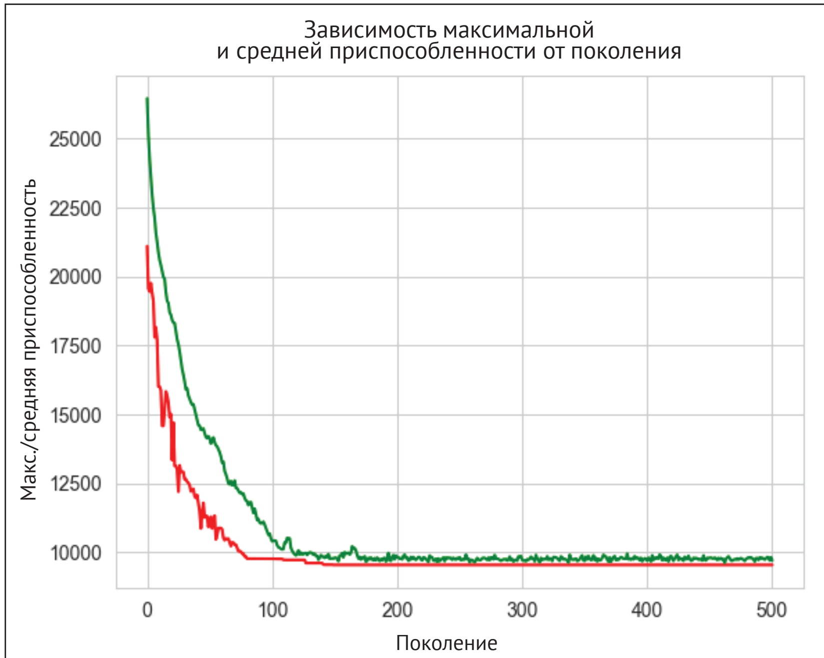


Статистика первой версии программы для решения задачи bayg29

Поскольку мы нашли хорошее, но не лучшее решение, можно попробовать улучшить результат. Например, изменить размер популяции, количество поколений и вероятности. Можно также заменить генетические операторы другими, совместимыми с типом хромосомы. Можно даже изменить начальное значение генератора случайных чисел, чтобы посмотреть, влияет ли оно на результат, или прогнать программу несколько раз с разными начальными значениями. Но в следующем разделе мы вместо всего этого применим элитизм и повысим степень исследования.

Улучшение результатов благодаря дополнительному исследованию и элитизму

Попытавшись увеличить количество поколений в предыдущей версии программы, мы придем к выводу, что решение не улучшается – программа застревает в неоптимальном решении, достигнутом раньше поколения 200, как показано на следующем рисунке для 500 поколений.



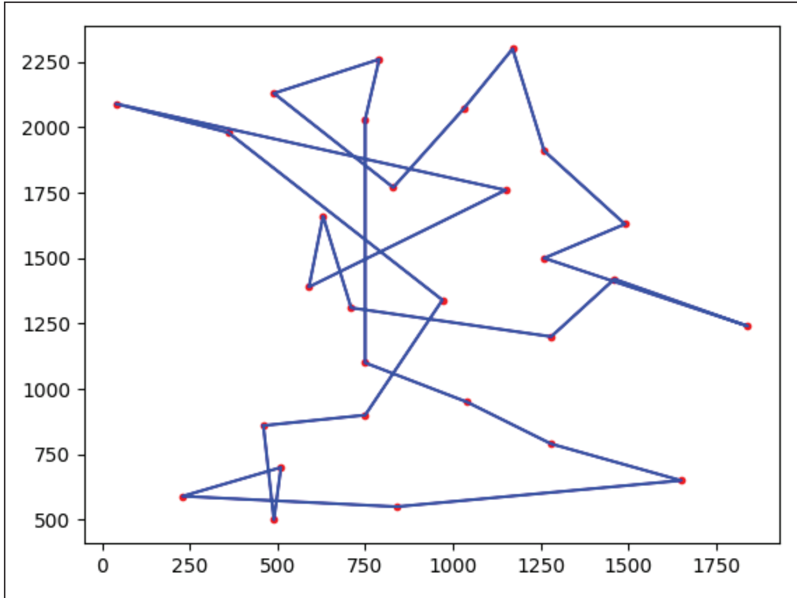
Статистика первой версии программы с 500 поколениями

А дальше из сходства графиков средней и лучшей приспособленности можно сделать вывод, что это решение захватило всю популяцию, и потому мы не увидим улучшения, если только не повезет с мутацией. В терминологии генетических алгоритмов это означает, что **использование** взяло верх над **исследованием**. Под использованием понимается удовлетворение уже имеющимися результатами, а под исследованием – предпочтение поиску новых решений. Нахождение тонкого баланса между тем и другим может привести к улучшению результатов.

Один из способов повысить степень исследования – уменьшить размер турнира с 3 до 2:

```
toolbox.register("select", tools.selTournament, tournsize=2)
```

В главе 2 мы говорили о том, что это повышает шансы менее успешных индивидуумов отобраться. Эти индивидуумы могут стать ключом к нахождению лучших решений в будущем. Но, запустив программу после такого изменения, мы будем горько разочарованы: наилучшая приспособленность превышает 13 000, а лучшее решение выглядит так:



Лучшее решение, найденное программой с размером турнира 2

Плохой результат можно объяснить, взглянув на графики статистики:

Статистика программы с размером турнира 2

Этот график показывает, что нам не удается сохранить лучшие решения. Из пилообразной формы графика, скачущего между хорошими и плохими значениями, следует, что хорошие решения быстро теряются из-за более либеральной схемы отбора, допускающей отбор не слишком удачных решений. В результате исследование заходит слишком далеко, и для компенсации нужно немного повысить степень использования. Это позволяет сделать механизм **элитизма**, с которым мы познакомились в главе 2.

Элитизм позволяет сохранить лучшие решения, защитив их от применения операторов отбора, скрещивания и мутации. Для его реализации придется залезть под капот и модифицировать код алгоритма DEAP `algorithms.eaSimple()`, поскольку каркас не дает прямого способа обойти эти операторы.

Модифицированный алгоритм `eaSimpleWithElitism()` можно найти в файле `elitism.py` по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/elitism.py.

Метод `eaSimpleWithElitism()` аналогичен оригинальному методу `eaSimple()`, но теперь объект `halloffame` используется для реализации механизма элитизма. Индивидуумы, хранящиеся в объекте `halloffame`, просто копируются в следующее поколение, не подвергаясь воздействию операторов отбора, скрещивания и мутации. Для этого нужно внести следующие модификации:

- вместо того чтобы отбирать индивидуумов в количестве, равном размеру популяции, мы отбираем их меньше на столько, сколько индивидуумов находится в зале славы:

```
offspring = toolbox.select(population, len(population) - hof_size)
```

- после применения генетических операторов индивидуумы добавляются из зала славы в популяцию:

```
offspring.extend(halloffame.items)
```

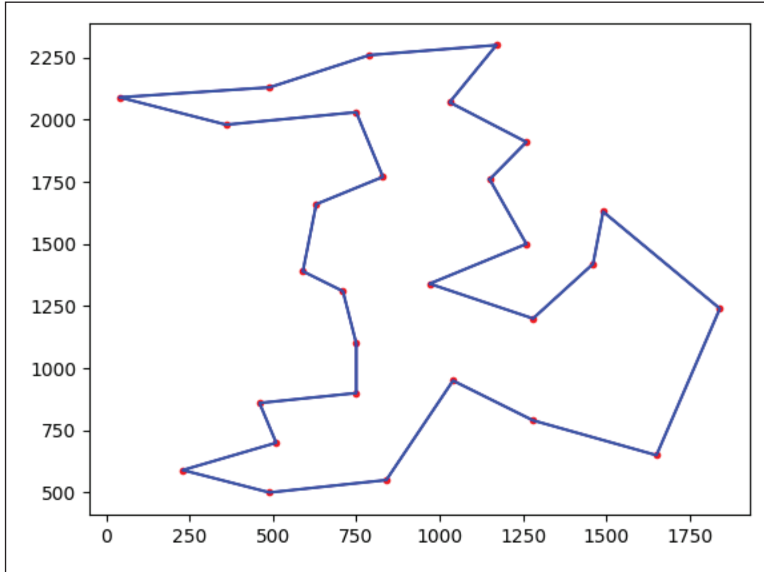
Теперь можно заменить вызов `algorithms.eaSimple()` вызовом `elitism.eaSimpleWithElitism()`, не изменяя никаких других параметров. Затем зададим константу `HALL_OF_FAME_SIZE` равной 30, говоря тем самым, что мы всегда хотим сохранять в популяции 30 лучших индивидуумов.

Модифицированный код программы можно найти по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/03-solve-tsp.py.

При выполнении этой версии программы мы находим оптимальное решение:

```
-- Лучший индивидуум = Individual('i', [0, 23, 12, 15, 26, 7, 22, 6, 24,
18, 10, 21, 16, 13, 17, 14, 3, 9, 19, 1, 20, 4, 28, 2, 25, 8, 11, 5, 27])
-- Лучшая приспособленность = 9074.146484375
```

График этого решения совпадает с показанным ранее графиком оптимального решения.



Лучшее решение,
найденное программой с размером турнира 2 и элитизмом

Графики статистики показывают, что нам удалось устранить наблюдавшийся ранее шум. Также удалось сохранять расстояние между лучшим и средним значением в течение гораздо более длительного времени, чем раньше.

Статистика программы с размером турнира 2 и элитизмом

В следующем разделе мы рассмотрим задачу о маршрутизации транспорта, добавляющую интересный новый поворот.

Решение задачи о маршрутизации транспорта

Допустим теперь, что мы управляем более крупным центром обработки и исполнения заказов. Требуется по-прежнему доставлять посылки заказчикам, но теперь в нашем распоряжении целый парк автомобилей. Как с его помощью оптимально доставить посылки заказчикам?

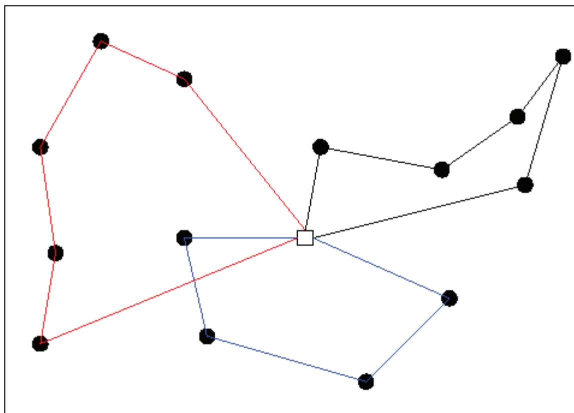
Это пример задачи о маршрутизации транспорта – обобщения задачи коммивояжера. В простейшем случае задача состоит из трех компонентов:

- список подлежащих посещению адресов;
- количество автомобилей;
- местоположение гаража, который является начальной и конечной точкой каждого маршрута.

У этой задачи много вариантов, например: наличие нескольких гаражей, доставка в оговоренный срок, разные типы транспорта (скажем, разной вместимости или с разной топливной эффективностью) и т. д.

Наша цель – минимизировать стоимость, которую тоже можно определить по-разному, например: время доставки всех посылок, затраты на топливо или расхождение во времени разъезда всех автомобилей.

Ниже приведен пример задачи о маршрутизации транспорта с тремя автомобилями. Города обозначены черными кружками, гараж – белым квадратиком, а маршруты трех автомобилей – разными цветами.



Пример задачи о маршрутизации транспорта
с тремя автомобилями

Источник: https://commons.wikimedia.org/wiki/
File:Figure_illustrating_the_vehicle_routing_problem.png.
Автор PierreSelim. Является общественным достоянием

В нашем примере целью является оптимизация времени доставки всех посылок. Поскольку все автомобили разъезжают одновременно, этот показатель определяется для автомобиля с самым длинным маршрутом. Поэтому можно сказать, что цель – минимизация длины самого длинного маршрута всех автомобилей доставки. Так, при наличии трех автомобилей каждое решение состоит из трех маршрутов. Мы обсчитываем все три, но для вычисления приспособленности оставляем только самый длинный – чем длиннее этот маршрут, тем хуже оценка. Поэтому неявно ставится задача сократить протяженность всех трех маршрутов, а также по возможности уравнять их длины.

Поскольку задачи коммивояжера и маршрутизации транспорта похожи, мы можем воспользоваться ранее написанным кодом. Для этого представим данные в задаче о маршрутизации следующим образом:

- экземпляр задачи коммивояжера, а именно список городов с координатами (или попарные расстояния между ними);
- местоположение гаража, совпадающее с одним из городов и представленное индексом этого города;
- количество автомобилей.

В следующих двух разделах мы покажем, как реализовать эту идею.

Представление решения

Как обычно, первым делом нужно ответить на вопрос о том, как представлять решение задачи. Чтобы не слишком отклоняться от уже решенной задачи коммивояжера, мы представим решение в виде списка, содержащего числа от 0 до $(n - 1) + (m - 1)$, где n – количество городов, а m – количество автомобилей. Например, если городов 10, а автомобилей 3 ($n = 10$, $m = 3$), то список будет содержать числа от 0 до 11:

(1, 3, 4, 6, 11, 9, 7, 2, 10, 5, 8, 0)

Первые n чисел (в нашем случае – от 0 до 9), как и раньше, представляют города, а последние $m - 1$ (10 и 11) будут использоваться как разделители, разбивающие список на маршруты. Например, список (1, 3, 4, 6, **11**, 9, 7, 2, **10**, 5, 8, 0) разбит на такие три маршрута:

(1, 3, 4, 6), (9, 7, 2), (5, 8, 0)

Индекс гаража придется удалить, потому что он не является частью какого-либо маршрута. Если, например, гараж имеет индекс 7, то окончательный список маршрутов будет выглядеть следующим образом:

(1, 3, 4, 6), (9, 2), (5, 8, 0)

При вычислении длины маршрута следует помнить, что маршрут начинается и заканчивается в гараже (городе с индексом 7). Поэтому для вычисления длин и рисования маршрутов мы будем использовать такие данные:

(7, 1, 3, 4, 6, 7), (7, 9, 2, 7), (7, 5, 8, 0, 7)

В следующем разделе мы увидим, как эта идея реализуется на Python.

Представление задачи на Python

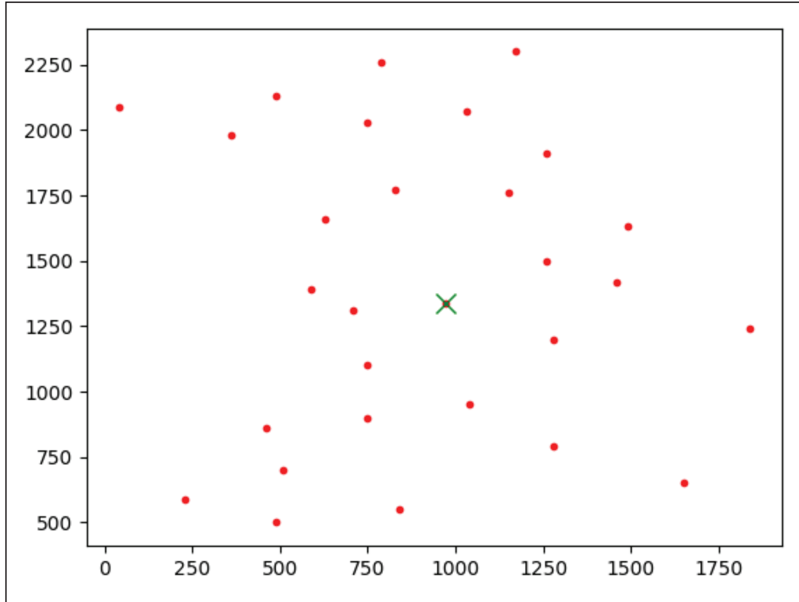
Для инкапсуляции данных задачи о маршрутизации мы создали Python `VehicleRoutingProblem`, который находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/vrp.py.

Класс `VehicleRoutingProblem` содержит в качестве атрибута экземпляр класса `TravelingSalesmanProblem`, который используется как контейнер индексов городов вместе с их координатами и расстояниями друг от друга. При создании экземпляра `VehicleRoutingProblem` конструктор создает и инициализирует внутренний экземпляр `TravelingSalesmanProblem`.

При инициализации класса `VehicleRoutingProblem` передаются данные, необходимые для `TravelingSalesmanProblem`, а также индекс гаража и количество автомобилей. Дополнительно класс `VehicleRoutingProblem` предоставляет следующие открытые методы:

- `getRoutes(indices)`: разбивает заданный список на маршруты, обнаруживая разделители;
- `getRouteDistance(indices)`: вычисляет полную длину пути, который начинается в гараже и проходит через все города, определяемые списком индексов;
- `getMaxDistance(indices)`: вычисляет максимум среди длин путей, описываемых заданными индексами, после разбиения списка индексов на маршруты;
- `getTotalDistance(indices)`: вычисляет итоговую длину пути для маршрутов, описываемых заданными индексами;
- `plotData(indices)`: разбивает список индексов на отдельные маршруты и рисует каждый маршрут своим цветом.

При запуске в качестве автономной программы метод `main` создает экземпляр `VehicleRoutingProblem`, указав в качестве объекта задачи коммивояжера `bayg29` (решенную в предыдущем разделе), и вызывает эти методы. Задается количество автомобилей 3 и индекс гаража 12 (соответствующий центральному городу). На рисунке ниже показаны местоположения городов (красные точки) и гаража (зеленый крестик).

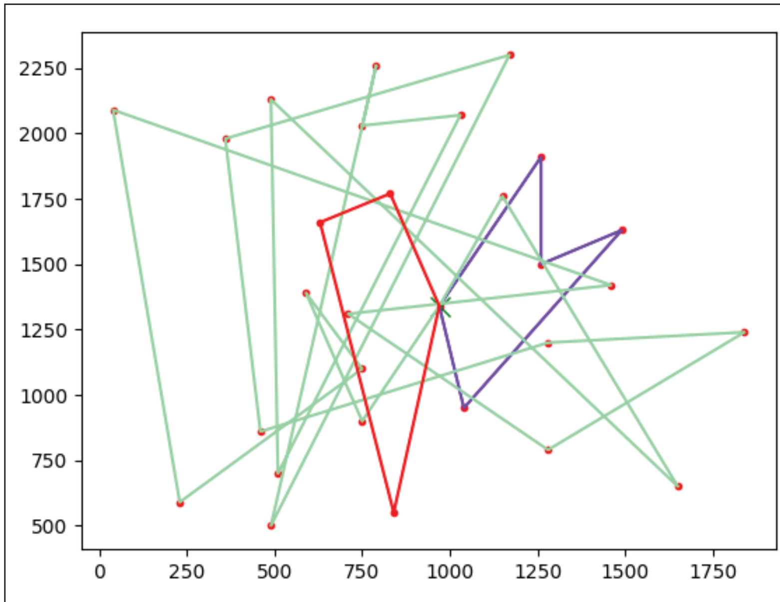


Задача о маршрутизации транспорта с базовой задачей коммивояжера bayg29. Красными точками обозначены города, крестиком – гараж

Метод `main` порождает случайное решение, разбивает его на маршруты и вычисляет их длины, как показано в следующем фрагменте:

```
random solution = [27, 23, 7, 18, 30, 14, 19, 3, 16, 2, 26, 9, 24, 22, 15,
17, 28, 11, 21, 12, 8, 4, 5, 13, 25, 6, 0, 29, 10, 1, 20]
route breakdown = [[27, 23, 7, 18], [14, 19, 3, 16, 2, 26, 9, 24, 22, 15,
17, 28, 11, 21, 8, 4, 5, 13, 25, 6, 0], [10, 1, 20]]
total distance = 26653.845703125
max distance = 21517.686
```

Заметим, что исходный список индексов теперь разбит на отдельные маршруты с помощью индексов-разделителей (29 и 30). Это случайное решение изображено на рисунке ниже.



Случайное решение задачи о маршрутизации с тремя автомобилями

Как и следовало ожидать, решение далеко от оптимального. Сразу видно, что порядок объезда городов на самом длинном (зеленом) маршруте неэффективен, к тому же зеленый маршрут гораздо длиннее двух других (красного и фиолетового).

В следующем подразделе мы попробуем найти хорошие решения с помощью генетического алгоритма.

Решение с помощью генетического алгоритма

Решение задачи о маршрутизации транспорта с применением генетического алгоритма находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/04-solve-vrp.py.

Поскольку мы выбрали примерно такое представление решения, как в задаче коммивояжера, можно использовать тот же подход, что в предыдущем разделе. Можно было бы также воспользоваться механизмом элитизма, взяв версию алгоритма, разработанную ранее. Тогда код будет очень похож на написанный для задачи коммивояжера.

Ниже приведены подробности решения.

1. Сначала программа создает экземпляр класса `VehicleRoutingProblem`, указав имя задачи коммивояжера `bayg29` (там хранятся данные о городах), а также индекс гаража 12 и количество автомобилей 3:

```
TSP_NAME = "bayg29"
```

```
NUM_OF_VEHICLES = 3
DEPOT_LOCATION = 12
vrp = vrp.VehicleRoutingProblem(TSP_NAME, NUM_OF_VEHICLES, DEPOT_LOCATION)
```

2. Функция приспособленности должна минимизировать длину самого длинного из трех маршрутов:

```
def vrpDistance(individual):
    return vrp.getMaxDistance(individual),

toolbox.register("evaluate", vrpDistance)
```

3. В качестве оператора отбора мы снова выбираем турнир размера 2, дополненный механизмом элитизма, и оператор скрещивания и мутации, специализированные для упорядоченных списков:

```
# Генетические операторы
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxUniformPartialyMatched, indpb=2.0/len(vrp))
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=1.0/len(vrp))
```

4. Поскольку задача маршрутизации в принципе труднее задачи коммивояжера, мы увеличиваем размер популяции и количество поколений:

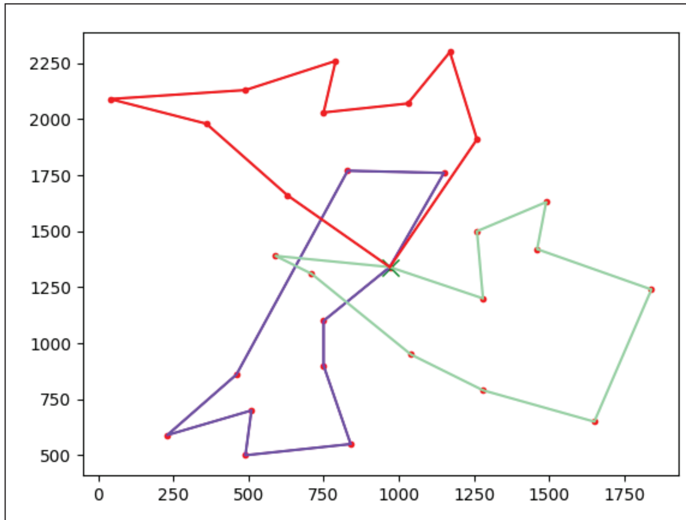
```
# Параметры генетического алгоритма
POPULATION_SIZE = 500
P_CROSSOVER = 0.9
P_MUTATION = 0.2
MAX_GENERATIONS = 1000
HALL_OF_FAME_SIZE = 30
```

Теперь можно запустить программу. Результаты приведены ниже – три маршрута, максимальная длина равна 3857:

```
-- Лучший индивидуум = Individual('i', [0, 20, 17, 16, 13, 21, 10, 14,
3, 29, 15, 23, 7, 26, 12, 22, 6, 24, 18, 9, 19, 30, 27, 11, 5, 4, 8, 25, 2,
28, 1])
-- Лучшая приспособленность = 3857.36376953125
-- Разбиение на маршруты = [[0, 20, 17, 16, 13, 21, 10, 14, 3], [15, 23, 7, 26,
22, 6, 24, 18, 9, 19], [27, 11, 5, 4, 8, 25, 2, 28, 1]]
-- полное расстояние = 11541.875
-- макс. расстояние = 3857.3638
```

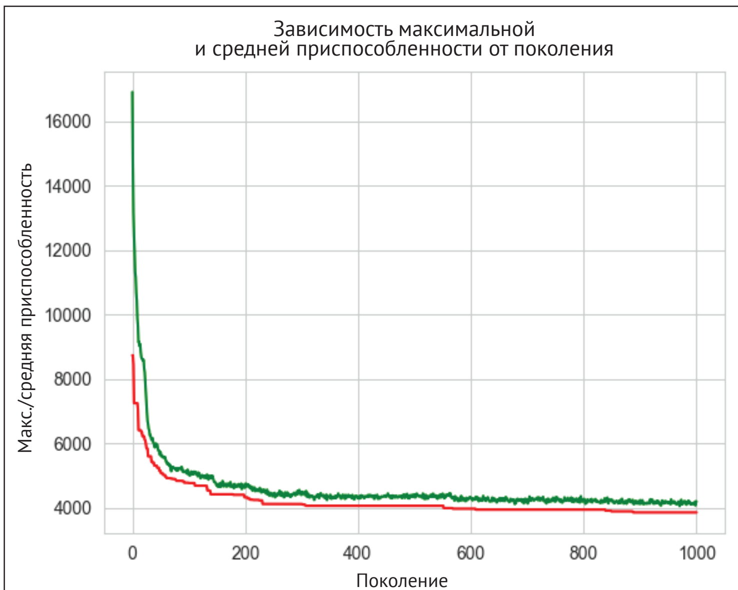
Снова подчеркнем, что решение разбито на три маршрута, в качестве разделителей используются два наибольших индекса (29, 30), а индекс гаража игнорируется. В итоге мы нашли три маршрута: два проходят через 9 городов, а третий – через 10.

Ниже показано, как выглядят эти маршруты.



Наилучшее решение,
найденное для задачи маршрутизации с тремя автомобилями

На следующем графике показано, что оптимизация в основном была произведена на протяжении первых 300 поколений, а потом улучшения были незначительны.



Статистика программы для задачи о маршрутизации с тремя автомобилями

А что, если изменить количество автомобилей? Увеличим его до шести и, не внося больше никаких изменений, запустим программу:

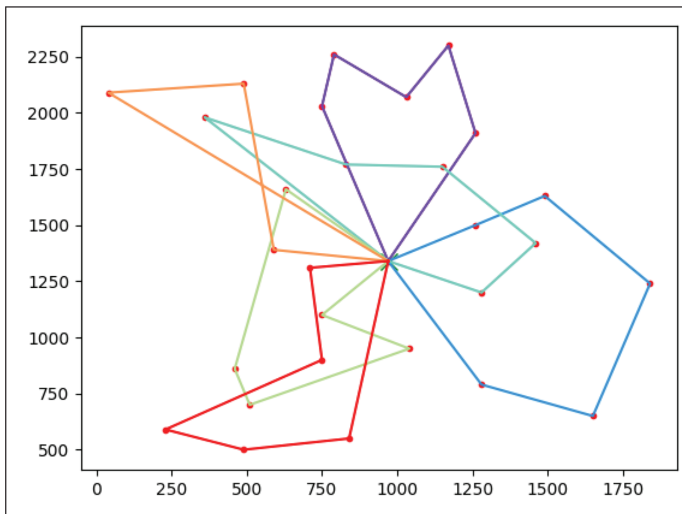
```
NUM_OF_VEHICLES = 6
```

Результаты показаны ниже – шесть маршрутов, максимальная длина 2803:

```
-- Лучший индивидуум = Individual('i', [27, 11, 5, 8, 4, 33, 12, 24, 6,
22, 7, 23, 29, 28, 20, 0, 26, 15, 32, 3, 18, 13, 17, 1, 31, 19, 25, 2, 30,
9, 14, 16, 21, 10])
-- Лучшая приспособленность = 2803.584716796875
-- Разбиение на маршруты = [[27, 11, 5, 8, 4], [24, 6, 22, 7, 23], [28, 20, 0,
26, 15], [3, 18, 13, 17, 1], [19, 25, 2], [9, 14, 16, 21, 10]]
-- полное расстояние = 16317.9892578125
-- макс. расстояние = 2803.5847
```

Заметим, что удвоение количества автомобилей не привело к пропорциональному уменьшению максимальной длины маршрута (2803 при 6 автомобилях и 3857 при трех). Вероятно, это объясняется тем, что каждый маршрут по-прежнему должен начинаться и заканчиваться в гараже, который автоматически добавляется к городам на маршруте.

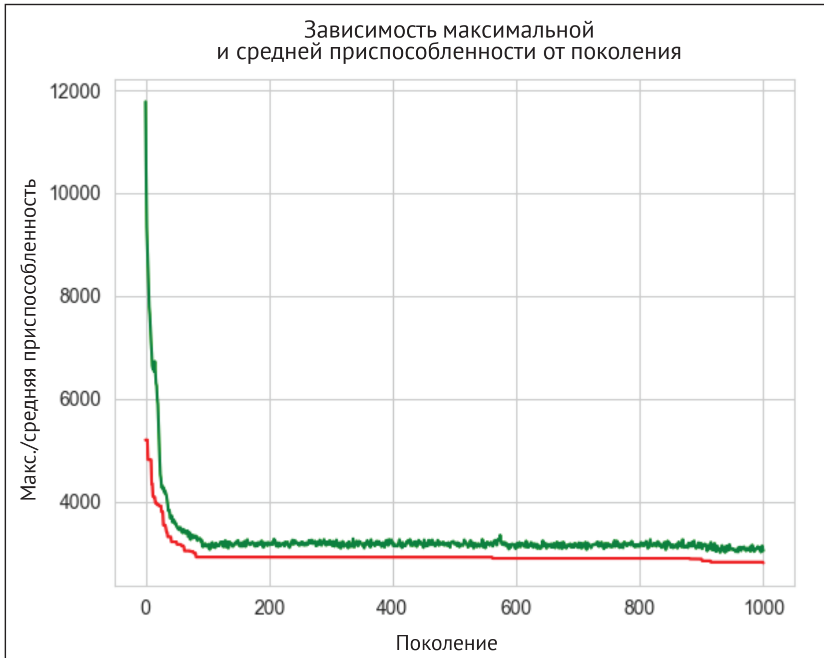
Решение с шестью маршрутами изображено на рисунке ниже.



Наилучшее решение,
найденное для задачи маршрутизации с шестью автомобилями

Интересно, что оранжевый маршрут не оптимален. Поскольку мы просили генетический алгоритм минимизировать **самый длинный** маршрут, то он вправе не выполнять оптимизацию более коротких маршрутов. Вам предлагается самостоятельно модифицировать решение, так чтобы остальные маршруты тоже оптимизировались.

Как и в случае с тремя автомобилями, график статистики показывает, что оптимизация в основном завершилась к 200-му поколению, а потом изменения были незначительны.



Статистика программы для задачи о маршрутизации с шестью автомобилями

Найденное решение выглядит разумно. Можно ли его улучшить? Что, если изменить количество автомобилей или положение гаража? Использовать другие генетические операторы или другие параметры, быть может, даже другой критерий приспособленности? Призываем вас поэкспериментировать и получить полезный опыт.

Резюме

В этой главе мы познакомились с задачами поиска и комбинаторной оптимизации. Мы рассмотрели три классические комбинаторные задачи, имеющие многочисленные практические применения: задача о рюкзаке, задача коммивояжера и задача о маршрутизации транспорта. Во всех трех задачах подход был одинаковым: определить подходящее представление решения, создать инкапсулирующий его класс, а затем воспользоваться этим классом в генетическом алгоритме. Для всех трех задач нам удалось найти решение в результате экспериментов с отображением генотипа в фенотип и исследования, подкрепленного элитизмом.

В следующей главе мы рассмотрим еще одно семейство задач – с ограничениями – и начнем с классической задачи об N ферзях.

Для дальнейшего чтения

За дополнительной информацией рекомендуем обратиться к следующим источникам:

- о решении задачи о рюкзаке с помощью динамического программирования см. книгу Giuseppe Ciaburro «Keras Reinforcement Learning Projects»;
- о задаче маршрутизации транспорта см. книгу Giuseppe Ciaburro «Keras Reinforcement Learning Projects».

Глава 5

Задачи с ограничениями

В этой главе мы научимся использовать генетические алгоритмы для решения задач с ограничениями. Начнем с того, что такое соблюдение ограничений и какое отношение оно имеет к задачам поиска и комбинаторной оптимизации. Затем мы рассмотрим несколько практических примеров задач с ограничениями и решим их на Python с помощью каркаса DEAP. Первой в нашем списке будет хорошо известная задача об *N* ферзях, за ней – задача о составлении графика дежурств медсестер и, наконец, задача о раскраске графа. Попутно мы расскажем о различии между жесткими и мягкими ограничениями и о том, как учесть его в решении.

В этой главе мы:

- узнаем о природе задач с ограничениями;
- решим задачу об *N* ферзях с помощью генетического алгоритма, написанного с применением каркаса DEAP;
- решим задачу о составлении графика дежурств медсестер;
- решим задачу о раскраске графа;
- узнаем, что такое жесткие и мягкие ограничения и как они применяются при решении задачи.

Технические требования

В этой главе нам понадобится Python 3 и следующие библиотеки:

- `deap`;
- `numpy`;
- `matplotlib`;
- `seaborn`;
- `networkx` (о ней будет рассказано здесь же).

Код приведенных в этой главе программ можно найти в репозитории книги на GitHub по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter05. Чтобы увидеть код в действии, посмотрите видео по адресу http://bit.ly/39233Qn.

Соблюдение ограничений в поисковых задачах

В предыдущей главе мы рассмотрели решение задач поиска, заключающееся в методичном оценивании состояний и переходов между состояниями. С каждым переходом состояний ассоциированы затраты или доходы, а цель заключается в том, чтобы минимизировать затраты или максимизировать доходы. Задачи с ограничениями – разновидность поисковых задач, в которых состояния должны удовлетворять ряду ограничений. Если нам удастся выразить ограничения в виде затрат, а затем минимизировать затраты, то задача с ограничениями сведется к общей задаче поиска.

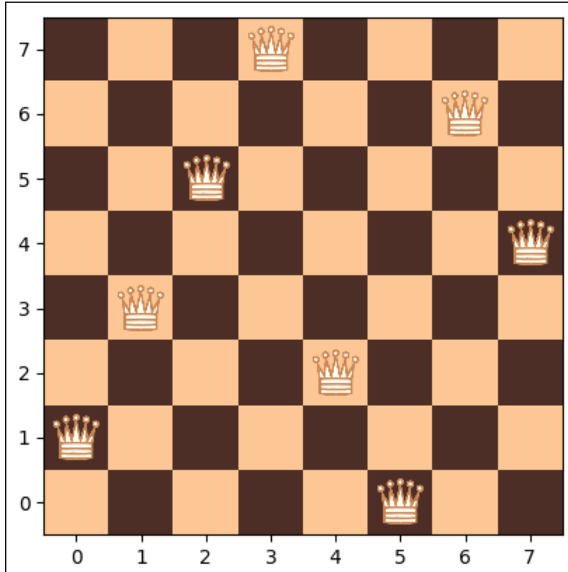
Как и задачи комбинаторной оптимизации, задачи с ограничениями имеют важные применения в таких областях, как искусственный интеллект, исследование операций и распознавание образов. Хорошее понимание природы таких задач может помочь при решении разнообразных, на первый взгляд, несвязанных проблем. Поскольку задачи с ограничениями бывают очень трудны, открываются широкие перспективы для применения генетических алгоритмов.

На примере задачи об *N* ферзях, представленной в следующем разделе, мы покажем, в чем состоит специфика задач с ограничениями, и продемонстрируем подход к их решению, очень напоминающий то, что мы видели в предыдущей главе.

Решение задачи об *N* ферзях

Эта классическая задача первоначально называлась **задачей о восьми ферзях** и ставилась на шахматной доске 8×8. Требовалось расставить на доске восемь ферзей, так чтобы никакие два не били друг друга. Иными словами, на одной горизонтали, на одной вертикали и на одной диагонали не может находиться два ферзя. Задача об *N* ферзях ставится точно так же, только на доске $N \times N$ для *N* ферзей.

Известно, что задача имеет решение для любого натурального n, кроме $n = 2$ и $n = 3$. В исходной задаче о восьми ферзях решений 92, из них 12 уникальных, не сводимых друг к другу симметрией. Одно из них представлено на рисунке ниже.



Одно из 92 возможных решений задачи о восьми ферзях

Применив методы комбинаторики, можно показать, что на доске 8×8 восемь ферзей можно расставить 4 426 165 368 способами. Но если мы хотим, чтобы никакие два ферзя не находились на одной горизонтали или на одной вертикали, то число способов сокращается до 8!, т. е. 40 320. Мы воспользуемся этой идеей в следующем подразделе, когда будем выбирать, как представить данные задачи.

Представление решения

При решении задачи об N ферзях мы можем воспользоваться знанием о том, что на каждой горизонтали находится ровно один ферзь и что никакие два ферзя не занимают одну вертикаль. Это значит, что любое потенциальное решение можно представить в виде упорядоченного списка целых чисел, или индексов, каждый из которых представляет вертикаль, занятую ферзем, находящимся в данной строке.

Например, в задаче о четырех ферзях на доске 4×4 может быть такой список индексов:

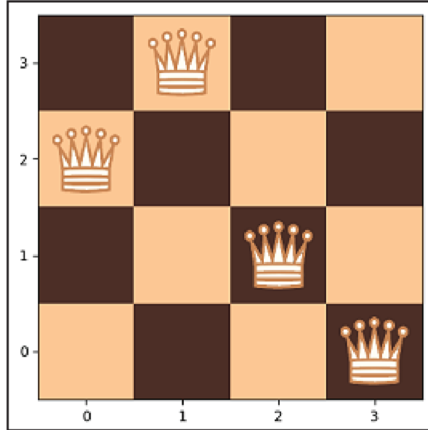
```
(3, 2, 0, 1)
```

Это означает следующее:

- ферзь на первой горизонтали занимает позицию 3 (четвертую вертикаль);
- ферзь на второй горизонтали занимает позицию 2 (третью вертикаль);
- ферзь на третьей горизонтали занимает позицию 0 (первую вертикаль);

- ферзь на четвертой горизонтали занимает позицию 1 (вторую вертикаль).

Такая расстановка изображена на рисунке ниже.

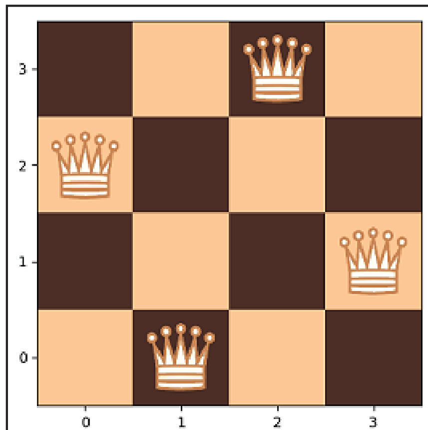


Расстановка ферзей,
соответствующая списку (3, 2, 0, 1)

Аналогично списку индексов

```
(1, 3, 0, 2)
```

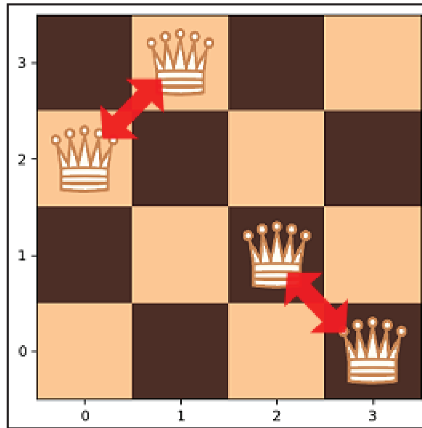
соответствует такая расстановка:



Расстановка ферзей,
соответствующая списку (1, 3, 0, 2)

В таких потенциальных решениях возможно лишь одно нарушение ограничения: два или более ферзей находятся на одной диагонали.

Например, в первом из двух рассмотренных выше решений есть два нарушения:



Нарушение ограничений в расстановке ферзей,
соответствующей списку (3, 2, 0, 1)

Но во второй расстановке никаких нарушений нет.

Это значит, что при оценивании решений, представленных таким образом, мы должны находить занятые несколькими ферзями **диагонали**.

Описанное представление лежит в основе Python-класса, который мы напишем в следующем подразделе.

Представление задачи на Python

Для инкапсуляции задачи об *N* ферзях мы создали Python-класс `NQueensProblem`. Он находится в репозитории книги на GitHub по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/queens.py.

Конструктору класса передается размер задачи. Класс предоставляет следующие открытые методы:

- `getViolationsCount(positions)`: подсчитывает количество нарушений в решении, представленном списком индексов;
- `plotBoard(positions)`: рисует расстановку ферзей на доске, соответствующую решению.

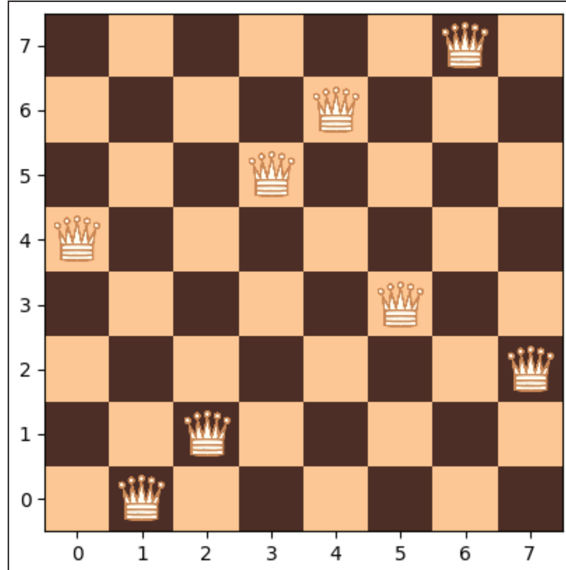
Метод `main` вызывает методы класса, чтобы создать экземпляр задачи о восьми ферзях и проверить следующее потенциальное решение:

```
(1, 2, 7, 5, 0, 3, 4, 6)
```

Затем он рисует это решение и вычисляет количество нарушений ограничения. Печатается следующий результат:

```
Number of violations = 3
```

Расстановка выглядит, как показано ниже, – сможете вы найти все три нарушения?



Расстановка ферзей,
соответствующая списку (1, 2, 7, 5, 0, 3, 4, 6)

В следующем подразделе мы применим к задаче об *N* ферзях генетический алгоритм.

Решение с помощью генетического алгоритма

Программа на Python для решения задачи об *N* ферзях находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/01-solve-n-queens.py.

Поскольку мы выбрали представление задачи списком (или массивом) индексов, похожее на представление задачи коммивояжера и задачи о маршрутизации транспорта (см. главу 4), то сможем применить такой же подход, что и раньше. И снова воспользуемся механизмом элитизма, взяв модифицированную версию простого генетического алгоритма, разработанную нами в главе 4.

Ниже описаны основные шаги решения.

1. Первым делом создаем экземпляр класса `NQueensProblem`, передавая конструктору количество ферзей:

   ```
   nQueens = queens.NQueensProblem(NUM_OF_QUEENS)
   ```

2. Поскольку мы стремимся минимизировать количество нарушений ограничений (в идеале свести к нулю), определяем единственную цель – минимизирующую стратегию приспособления:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. Так как решение представлено упорядоченным списком целых чисел, описывающих позиции ферзей, воспользуемся следующими элементами инструментария для создания начальной популяции:

```
# создать оператор, который генерирует случайную перестановку индексов:
toolbox.register("randomOrder", random.sample,
range(len(nQueens)), len(nQueens))

toolbox.register("individualCreator", tools.initIterate,
creator.Individual, toolbox.randomOrder)
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. Определим функцию приспособленности – количество нарушений в расстановке ферзей, представленной конкретным решением:

```
def getViolationsCount(individual):
    return nQueens.getViolationsCount(individual),

toolbox.register("evaluate", getViolationsCount)
```

5. В качестве оператора отбора берем турнир размера 2, а операторы скрещивания и мутации делаем совместимыми с упорядоченными списками:

```
# Генетические операторы
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxUniformPartialyMatched, indpb=2.0/len(vrp))
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=1.0/len(vrp))
```

6. Как и раньше, используем механизм элитизма, без изменения копируя индивидуумов из зала славы (лучших в текущем поколении) в следующее поколение. В предыдущей главе мы видели, что этот подход хорошо работает с турнирным отбором размера 2:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                  toolbox,
                                                  cxpb=P_CROSSOVER,
                                                  mutpb=P_MUTATION,
                                                  ngen=MAX_GENERATIONS,
                                                  stats=stats,
                                                  halloffame=hof,
                                                  verbose=True)
```

7. Поскольку задача об *N* ферзях может иметь несколько решений, печатаем всех индивидуумов из зала славы, а не только первого, чтобы понять, сколько решений мы нашли:

```
print("- Лучшие решения:")
for i in range(HALL_OF_FAME_SIZE):
    print(i, ": ", hof.items[i].fitness.values[0], " -> ",
hof.items[i])
```

Ранее мы видели, что при выбранном нами представлении количество расстановок, подлежащих проверке, сокращается примерно до 40 000, это довольно маленькая задача. Чтобы было интереснее, увеличим число ферзей до 16, тогда количество комбинаций равно 16!, т. е. 20 922 789 888 000 – колоссальное число. Количество решений этой задачи тоже велико – немногим меньше 15 миллионов. Но при таком громадном числе расстановок искать правильные решения – все равно что иголку в стоге сена.

Прежде чем запускать программу, зададим параметры алгоритма:

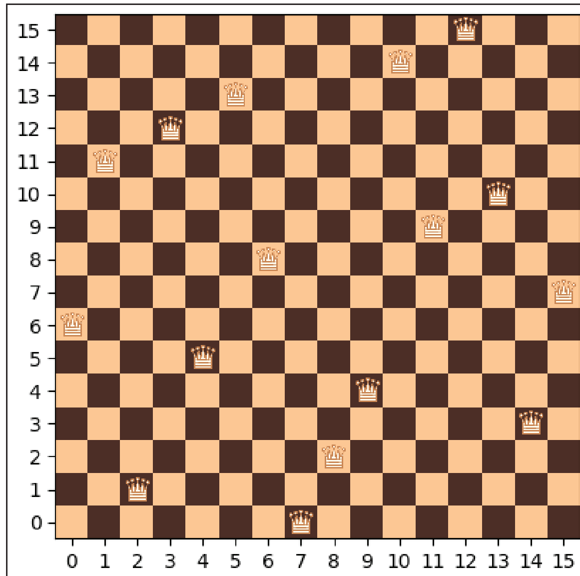
```
NUM_OF_QUEENS = 16
POPULATION_SIZE = 300
MAX_GENERATIONS = 100
HALL_OF_FAME_SIZE = 30
P_CROSSOVER = 0.9
P_MUTATION = 0.1
```

При таких параметрах программа печатает:

```
gen nevals min avg
0   300    3   10.4533
1   246    3   8.85333
...
23  250    1   4.38
24  227    0   4.32
...
- Лучшие решения:
0 : 0.0 -> Individual('i', [7, 2, 8, 14, 9, 4, 0, 15, 6, 11, 13, 1, 3, 5, 10, 12])
1 : 0.0 -> Individual('i', [7, 2, 6, 14, 9, 4, 0, 15, 8, 11, 13, 1, 3, 5, 12, 10])
...
7 : 0.0 -> Individual('i', [14, 2, 6, 12, 7, 4, 0, 15, 8, 11, 3, 1, 9, 5, 10, 13])
8 : 1.0 -> Individual('i', [2, 13, 6, 12, 7, 4, 0, 15, 8, 14, 3, 1, 9, 5, 10, 11])
...
```

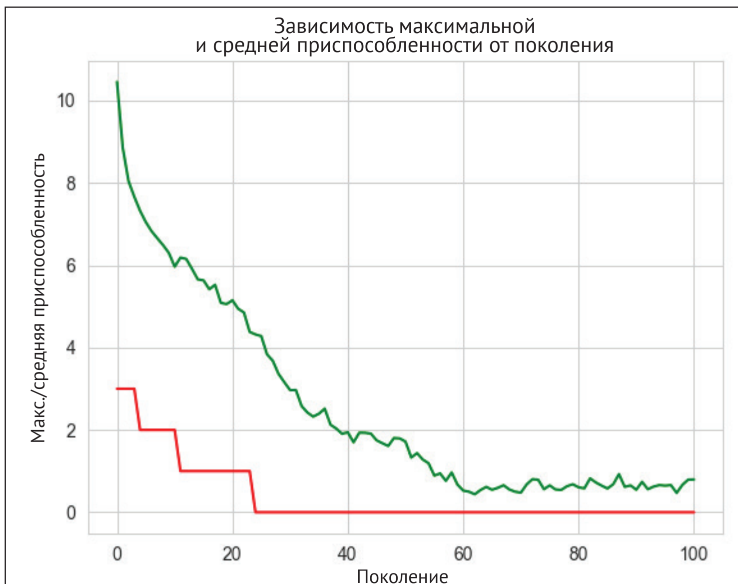
Из этой распечатки видно, что первое решение было найдено в поколении 24, значение приспособленности для него равно 0, т. е. нарушений не было. Кроме того, мы видим, что в этом прогоне было найдено восемь разных решений, это индивидуумы 0–7 в зале славы с приспособленностью 0. Уже для следующего элемента приспособленность равна 1, т. е. имеется одно нарушение.

На первом рисунке, созданном программой, изображена расстановка 16 ферзей на доске 16×16, соответствующая первому найденному решению – (7, 2, 8, 14, 9, 4, 0, 15, 6, 11, 13, 1, 3, 5, 10, 12):



Правильная расстановка 16 ферзей, найденная программой

На втором рисунке изображены графики зависимости минимальной и средней приспособленности от номера поколения. По ним видно, что хотя лучшая приспособленность 0 была найдена довольно рано, в 24-м поколении, средняя приспособленность продолжает убывать, потому что обнаруживаются новые решения.



Статистика программы для задачи о 16 ферзях

Увеличив `MAX_GENERATIONS` до 400 и не меняя больше ничего, мы найдем 38 решений. При `MAX_GENERATIONS = 500` все 50 индивидуумов в зале славы являются правильными расстановками. Попробуйте поиграть с другими сочетаниями параметров алгоритма, а также решить задачу об *N* ферзях при других значениях *N*.

В следующем разделе мы перейдем от расстановки фигур на доске к составлению графика работ.

Решение задачи о составлении графика дежурств медсестер

Пусть требуется составить недельный график дежурств медсестер в отделении больницы. В сутках три смены – утренняя, дневная и вечерняя, и в каждой смене должно быть не менее одной из восьми работающих в отделении медсестер. Если вам кажется, что это просто, примите во внимание перечень действующих в больнице правил:

- сестре не разрешается работать две смены подряд;
- сестре не разрешается работать более пяти смен в неделю;
- количество сестер в смене должно удовлетворять дополнительным ограничениям:
  - в утренней смене: 2–3 сестры;
  - в дневной смене: 2–4 сестры;
  - в ночной смене: 1–2 сестры.

Кроме того, у каждой сестры могут быть пожелания. Например, одна сестра предпочитает только утренние смены, другая не хочет работать днем и т. д.

Эта задача о составлении графика дежурств медсестер (СГДМ) имеет много вариантов. Например, у сестер могут быть различные специальности, можно разрешить сверхурочную работу, даже сами смены могут быть разными – скажем, 8- и 12-часовыми.

Теперь вы, наверное, согласитесь, что лучше поручить составление графика программе. И почему бы не воспользоваться нашими знаниями о генетических алгоритмах? Как обычно, начнем с представления решения.

Представление решения

Мы решили представлять график в виде двоичного списка (или массива), поскольку интуитивно понятно, как его интерпретировать, а с точки зрения генетических алгоритмов такое представление выглядит естественно.

Каждой сестре соответствует двоичная строка, представляющая 21 смену в неделю. Значение 1 означает, что сестра должна выйти в эту смену. Рассмотрим, например, следующий список:

```
(0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0)
```

Его можно разбить на группы по три значения, представляющие смены, в которые сестра выходит на работу в каждый день недели:

Воскресенье	Понедельник	Вторник	Среда	Четверг	Пятница	Суббота
(0,1,0)	(1,0,1)	(0,1,1)	(0,0,0)	(0,0,1)	(1,0,0)	(0,1,0)
дневная	утренняя и ночная	дневная и ночная	не выходит	ночная	утренняя	дневная

Графики дежурств всех сестер можно конкатенировать, создав один длинный двоичный список, представляющий полное решение.

При оценке решения этот длинный список можно разбить на графики отдельных медсестер и проверять нарушения ограничений. Так, в примере выше имеется два случая, когда идут подряд две единицы (дневная и ночная смены, а позже – ночная и утренняя). Количество смен в неделю для данной сестры можно вычислить, подсчитав количество единиц, – получится 8 смен. Можно также проверить, удовлетворяются ли пожелания медсестер, для чего нужно сравнить назначение сестры на смены с ее пожеланиями.

Наконец, чтобы проверить ограничения на количество сестер в смене, можно просуммировать недельные графики всех сестер и посмотреть, есть ли смены, в которых сестер больше максимально допустимого или меньше минимально допустимого предела.

Но прежде чем переходить к реализации, нужно обсудить различие между жесткими и мягкими ограничениями.

Жесткие и мягкие ограничения

При составлении графика дежурств медсестер следует помнить, что некоторые больничные правила нарушать нельзя. Если график содержит хотя бы одно нарушение этих правил, то он считается недопустимым. В общем случае такие ограничения называются **жесткими**.

С другой стороны, пожелания медсестер можно считать **мягкими ограничениями**. Мы хотели бы их удовлетворить, и решение, содержащее мало таких нарушений или не содержащее их вовсе, считается лучше, чем решение с большим числом нарушений. Тем не менее нарушение мягких ограничений не делает решение недопустимым.

В задаче об *N* ферзях все ограничения – на размещение по вертикали, горизонтали или диагонали – были жесткими. Если бы мы не нашли ни одного решения без нарушений, задачу нельзя было бы считать решенной. С другой стороны, больничные правила считаются жесткими ограничениями, а пожелания медсестер – мягкими. Поэтому мы ищем решение, не нарушающее больничных правил и минимизирующее количество невыполненных пожеланий сестер.

С мягкими ограничениями мы обращаемся так же, как в любой задаче оптимизации, т. е. пытаемся минимизировать количество их нарушений, но вот как быть с жесткими ограничениями? Есть несколько стратегий.

- Найти такое представление (кодировку) решения, при котором **исключается сама возможность** нарушить жесткое ограничение. В задаче

об N ферзях нам удалось найти представление, исключающее возможность двух из трех нарушений – на размещение нескольких ферзей на одной горизонтали или вертикали. Это значительно упростило решение задачи. Но в общем случае такого представления может и не быть.

- При оценивании решений **отбрасывать** те, что нарушают хотя бы одно жесткое ограничение. Недостаток такого подхода – в том, что информация, содержащаяся в этих решениях, теряется, а она может оказаться ценной. В результате процесс оптимизации сильно замедляется.
- При оценивании решений **исправлять** те, что нарушают жесткие ограничения. Иными словами, найти такой способ модификации решения, который устранял бы нарушения. Для большинства задач найти подобную процедуру исправления трудно или невозможно, к тому же сам процесс исправления может привести к значительной потере информации.
- При оценивании решений штрафовать те, что нарушают жесткие ограничения. Это уменьшает оценку решения и делает его менее желательным, но не исключает полностью, поэтому содержащаяся в нем информация не теряется. При таком подходе жесткое ограничение рассматривается как мягкое, но с более высоким штрафом. Проблема в том, как выбрать подходящую величину штрафа. Если штраф слишком велик, то решения, по сути дела, исключаются, а если слишком мал, то недопустимое решение может оказаться оптимальным.

Мы остановимся на четвертом подходе – будем штрафовать нарушения жестких ограничений сильнее, чем мягких. Для этого мы напишем функцию стоимости, в которой цена нарушения жесткого ограничения больше, чем мягкого. Затем полная стоимость будет использоваться как функция приспособленности, подлежащая минимизации. Все это реализовано в представлении задачи, которое мы обсудим в следующем подразделе.

Представление задачи на Python

Для инкапсуляции задачи о составлении графика дежурств мы создали Python-класс `NurseSchedulingProblem`. Он находится в файле https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/nurses.py.

Конструктор класса принимает параметр `hardConstraintPenalty` – коэффициент штрафа за нарушение жесткого ограничения и инициализирует различные параметры задачи.

```
# список медсестер
self.nurses = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

# пожелания медсестер к сменам - утренняя, дневная, ночная
self.shiftPreference = [[1, 0, 0], [1, 1, 0], [0, 1, 1], [0, 1, 0], [0, 0, 1],
[1, 1, 1], [0, 1, 1], [1, 1, 1]]
# минимальное и максимальное количество медсестер в сменах - утренней, дневной,
# ночной
```

```
self.shiftMin = [2, 2, 1]
self.shiftMax = [3, 4, 2]

# максимальное количество смен, которые разрешено отработать медсестре в неделю
self.maxShiftsPerWeek = 5
```

В классе имеется также метод для преобразования заданного графика в словарь, содержащий графики для каждой медсестры:

- `getNurseShifts(schedule)`

Следующие методы используются для подсчета нарушений разных типов:

- `countConsecutiveShiftViolations(nurseShiftsDict)`
- `countShiftsPerWeekViolations(nurseShiftsDict)`
- `countNursesPerShiftViolations(nurseShiftsDict)`
- `countShiftPreferenceViolations(nurseShiftsDict)`

Дополнительно класс предоставляет следующие открытые методы:

- `getCost(schedule)`: вычисляет полную стоимость различных нарушений графика. В этом методе используется переменная `hardConstraintPenalty`;
- `printScheduleInfo(schedule)`: печатает график и сведения о нарушениях.

Метод `main` вызывает методы класса: создает экземпляр задачи и проверяет случайно выбранное решение. Если `hardConstraintPenalty = 10`, то результат мог бы выглядеть так:

```
Случайное решение =
[1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 0 1 0 1 0 1 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1
 0 0 1 0 1 0 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 0 1 0 1 0 1 1 0 1 0 1 1 1
 1 0 1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 0 1 0 0 1 1 0 1 1 1 0 0 0 0 0
 0 1 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 1 1 0 0 0 0 1 1 0 1 1 0
 0 1 0 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1]

Графики медсестер:
A : [1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 0 1 0 1 0 1]
B : [1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1 0 0 1 0 1]
C : [0 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 0 1]
D : [0 1 0 1 1 0 1 0 1 1 1 1 0 1 0 0 0 1 1 0 1]
E : [1 1 1 0 1 1 0 1 1 1 1 1 0 1 0 0 1 1 0 1 1]
F : [1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 1 0 0 0 0]
G : [0 0 0 0 1 1 0 0 1 1 1 1 0 0 0 0 1 1 0 1 1]
H : [0 0 1 0 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1]
нарушений ограничения на две смены подряд = 40

смен в неделю = [9, 13, 13, 12, 15, 5, 10, 9]
нарушений ограничения на число смен в неделю = 46

сестер в смене = [4, 2, 4, 1, 6, 6, 4, 4, 5, 4, 5, 5, 2, 4, 4, 4, 5, 3, 4, 3, 7]
нарушений ограничения на число сестер в смене = 30

нарушений пожеланий сестер = 30

Общая стоимость = 1190
```

Как видим, случайно сгенерированное решение, скорее всего, будет содержать много нарушений, а значит, его стоимость будет высока. В следующем подразделе мы попытаемся минимизировать стоимость и исключить все нарушения жестких ограничений.

Решение на основе генетического алгоритма

Для решения задачи о составлении графика дежурств мы написали на Python программу, которая находится в файле https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/02-solve-nurses.py.

Поскольку решение представлено списком (или массивом) двоичных значений, мы можем воспользоваться тем же подходом, который успешно применяли к решению других задач, например задачи о рюкзаке в главе 4.

Ниже описаны основные шаги решения.

1. Сначала создаем экземпляр класса `NurseSchedulingProblem` со значением `hardConstraintPenalty`, равным константе `HARD_CONSTRAINT_PENALTY`:

```
nsp = nurses.NurseSchedulingProblem(HARD_CONSTRAINT_PENALTY)
```

2. Поскольку мы стремимся минимизировать стоимость, определяем единственную цель – минимизирующую стратегию приспособления:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. Поскольку решение представлено списком, содержащим значения 0 и 1, воспользуемся следующими элементами инструментария для создания начальной популяции:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.zeroOrOne, len(nsp))
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. Функция приспособленности вычисляет стоимость различных нарушений в графике, представленном одним решением:

```
def getCost(individual):
    return nsp.getCost(individual),

toolbox.register("evaluate", getCost)
```

5. В качестве оператора отбора берем турнир размера 2, будем также использовать двухточечный оператор скрещивания и мутацию инвертированием бита, поскольку они удобны при работе с двоичными списками:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/len(nsp))
```

6. Продолжаем применять элитизм, а конкретно копируем без изменения лучших индивидуумов из зала славы в следующее поколение:

```
population, logbook = elitism.eaSimpleWithElitism(population,
toolbox, cxpb=P_CROSSOVER, mutpb=P_MUTATION,
ngen=MAX_GENERATIONS, stats=stats, halloffame=hof, verbose=True)
```

7. По завершении алгоритма распечатываем сведения о лучших найденных решениях:

```
nsp.printScheduleInfo(best)
```

Прежде чем запускать программу, зададим параметры алгоритма:

```
POPULATION_SIZE = 300
P_CROSSOVER = 0.9
P_MUTATION = 0.1
MAX_GENERATIONS = 200
HALL_OF_FAME_SIZE = 30
```

Кроме того, зададим штраф за нарушение жестких ограничений равным 1, уравняв тем самым стоимость нарушения жестких и мягких ограничений:

```
HARD_CONSTRAINT_PENALTY = 1
```

Программа, запущенная с такими параметрами, печатает следующую информацию:

```
-- Лучшая приспособленность = 3.0
-- График =
Графики медсестер:
A : [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0]
B : [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0]
C : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
D : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]
E : [0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0]
F : [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0]
G : [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1]
H : [1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0]
нарушений ограничения на две смены подряд = 0

смен в неделю = [5, 6, 2, 5, 4, 5, 5, 5]
нарушений ограничения на число смен в неделю = 1

сестер в смене = [2, 2, 1, 2, 2, 1, 2, 2, 1, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 1]
нарушений ограничения на число сестер в смене = 0

нарушений пожеланий сестер = 2
```

Может показаться, что это неплохо, потому что имеется всего три нарушения ограничений. Но одно из них – несоблюдение количества смен в неделю: сестре B запланировано шесть смен, тогда как разрешено не больше пяти. Этого достаточно, чтобы решение было признано недопустимым.

Попробуем избавиться от таких нарушений, задав штраф за нарушение жесткого ограничения равным 10:

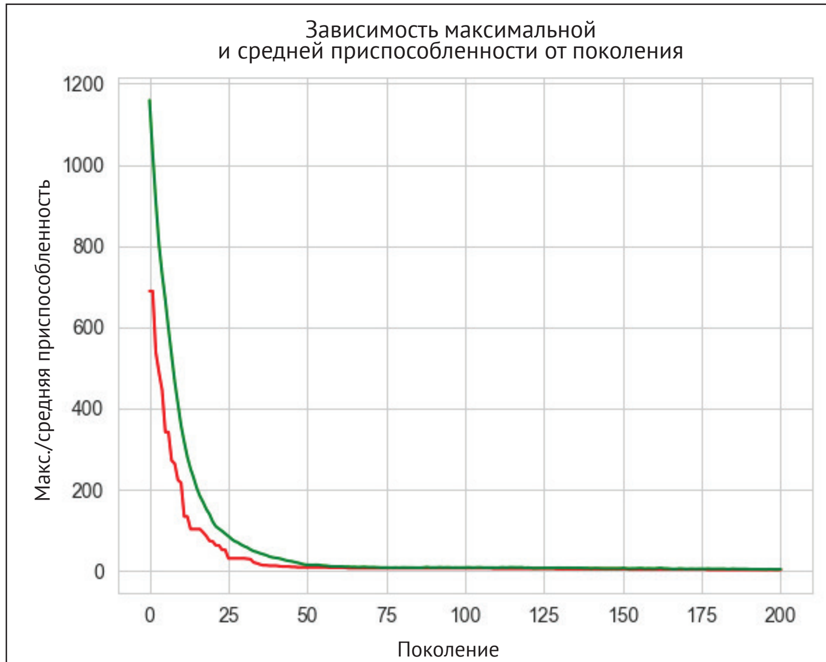
```
HARD_CONSTRAINT_PENALTY = 10
```

Теперь результат выглядит так:

```
-- Лучшая приспособленность = 3.0
-- График =
Графики медсестер:
A : [0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
B : [1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0]
C : [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1]
D : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]
E : [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
F : [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0]
G : [0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0]
H : [1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0]
нарушений ограничения на две смены подряд = 0

смен в неделю = [4, 5, 5, 5, 3, 5, 5, 5]
нарушений ограничения на число смен в неделю = 0

сестер в смене = [2, 2, 1, 2, 2, 1, 2, 2, 2, 2, 2, 1, 2, 2, 1, 2, 2, 2, 2, 2, 1]
нарушений ограничения на число сестер в смене = 0

нарушений пожеланий сестер = 3
```

Нарушений снова три, но на этот раз нарушены только мягкие ограничения, поэтому решение допустимо.

На рисунке ниже изображены графики зависимости минимальной и средней приспособленности от номера поколения. Мы видим, что за первые 40–50 поколений алгоритм сумел исключить все нарушения жестких ограничений, после чего были лишь небольшие улучшения, связанные с исключением очередного нарушения мягких ограничений.



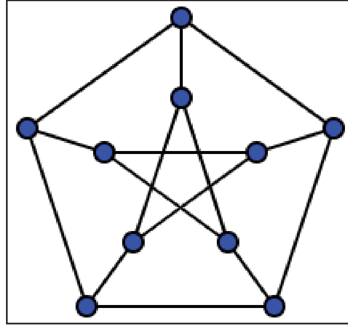
Статистика программы для задачи
о составлении графика дежурств медсестер

В нашей задаче оказалось достаточно сделать штраф за нарушение жестких ограничений в 10 раз больше. В других задачах может понадобиться более высокое значение. Попробуйте поэкспериментировать с другими параметрами, а также с определениями генетических операторов.

Компромисс между жесткими и мягкими ограничениями присутствует и в нашей следующей задаче – о раскраске графа.

Решение задачи о раскраске графа

В разделе математики, называемом *теорией графов*, рассматриваются графы – структурированные множества объектов, описывающие отношения между парами этих объектов. Объекты представлены вершинами (или узлами) графа, а отношения между парами объектов – ребрами, соединяющими вершины. Обычно вершины графа изображаются кружочками, а ребра – линиями, как на следующем графе Петерсена, названном в честь датского математика Юлиуса Петерсена.



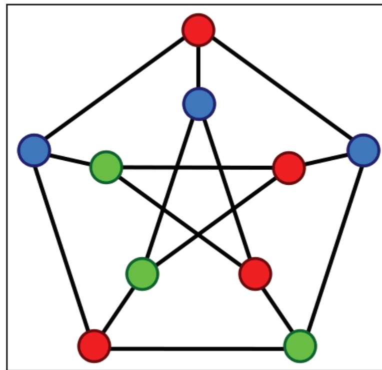
Граф Петерсена

Источник: https://commons.wikimedia.org/wiki/File:Petersen1_tiny.svg.
Автор: Leshabirukov. Публикуется по лицензии Creative Commons CC BY-SA 3.0:
https://creativecommons.org/licenses/by-sa/3.0/deed.en

Графы чрезвычайно полезны, поскольку позволяют представить и исследовать на удивление широкий спектр практических структур, паттернов и связей, например социальные сети, сети генерации и распределения электроэнергии, структуры веб-сайтов, лингвистические связи, компьютерные сети, структуры атомов, закономерности миграции и многое другое.

Задача о раскраске графа заключается в том, чтобы назначить каждой вершине цвет таким образом, чтобы никакие две пары смежных (соединенных ребром) вершин не были выкрашены в один цвет. Такая раскраска называется **правильной**.

На рисунке ниже показан правильно раскрашенный граф Петерсена:



Правильная раскраска графа Петерсена

Источник: https://en.wikipedia.org/wiki/File:Petersen_graph_3-coloring.svg.
Является общественным достоянием

На раскраску часто налагается дополнительное требование – использовать **минимально возможное** число цветов. Например, граф Петерсена можно раскрасить в три цвета (см. рисунок выше). Но в два цвета его раскрасить

невозможно. В теории графов говорят, что *хроматическое число* этого графа равно трем.

Почему раскраска вершин графа так интересна? Потому, что многие реальные задачи можно сформулировать таким образом, что для их решения нужно раскрасить некоторый граф. Например, составление расписания занятий для студентов или сменного графика для рабочих можно рассматривать как граф, в котором смежные вершины представляют занятия или смены, приводящие к конфликту. Конфликтом могут быть лекции, проходящие в одно и то же время, или несколько смен подряд (знакомо звучит?). Поэтому назначение одного лица на обе лекции (или в обе смены) сделает расписание недопустимым. Если каждому лицу сопоставить цвет, то раскрашивание смежных вершин в разные цвета устранит все конфликты. Задачу об *N* ферзях, рассмотренную в начале этой главы, можно сформулировать как задачу о раскраске графа: вершинами графа являются поля шахматной доски, а вершины, находящиеся на одной горизонтали, вертикали или диагонали, соединены ребром. Можно привести и другие примеры: распределение частот между радиостанциями, планирование избыточности электрических сетей, время включения и выключения светофоров и даже решение головоломки судоку.

Надеюсь, я убедил вас, что задача о раскраске графа заслуживает внимания. Как обычно, начнем с подходящего представления ее решения.

Представление решения

Обобщением двоичного списка (или массива) является список целых чисел, в котором каждое число представляет уникальный цвет, а элементы списка соответствуют вершинам графа.

Например, поскольку в графе Петерсена 10 вершин, мы можем сопоставить каждой вершине индекс от 0 до 9. Тогда задачу о раскраске этого графа можно представить списком из 10 элементов.

Рассмотрим, к примеру, такое представление:

```
(0, 2, 1, 3, 1, 2, 0, 3, 3, 0)
```

Вот как оно интерпретируется:

- используется четыре цвета, представленных числами 0, 1, 2, 3;
- первая, седьмая и десятая вершины окрашены в цвет 0;
- третья и пятая вершины окрашены в цвет 1;
- вторая и шестая вершины окрашены в цвет 2;
- четвертая, восьмая и девятая вершины окрашены в цвет 3.

Чтобы оценить это решение, мы должны перебрать все пары смежных вершин и проверить, окрашены ли они в один цвет. Если это так, то ограничение на раскраску нарушено, а наша цель – свести количество нарушений к нулю и тем самым найти правильную раскраску.

Но не надо забывать, что мы еще должны минимизировать количество цветов. Если оно нам заранее известно, то можно сразу зарезервировать такое количество целых чисел для представления цветов. А если нет? Можно было бы начать с оценки (проще говоря, догадки) нужного числа цветов. Если удастся

найти решение с таким числом цветом, то уменьшим его и попробуем снова. Если не удастся, увеличим и попытаемся еще раз, пока не найдем минимальное число цветов, при котором есть решение. Однако можно найти искомое число цветов быстрее, если воспользоваться жесткими и мягкими ограничениями.

Жесткие и мягкие ограничения в задаче о раскраске графа

Решая задачу о составлении графика дежурств медсестер, мы обратили внимание на различие между жесткими ограничениями (которые необходимо соблюсти, чтобы решение считалось допустимым) и мягкими ограничениями (которые желательно соблюсти в наилучшем решении). В задаче о раскраске графа требование о назначении цветов – смежные вершины не должны быть раскрашены в один цвет – является жестким ограничением. Чтобы решение считалось допустимым, количество нарушений этого требования должно быть равно нулю.

С другой стороны, минимизация количества цветов может рассматриваться как мягкое ограничение. Мы хотели бы его выполнить, но не ценой нарушения жесткого ограничения. Это открывает возможность запустить алгоритм с количеством цветов, превышающим нашу оценку минимума, а затем поручить алгоритму минимизировать его, пока – в идеале – не будет найдено истинное минимальное число цветов.

Как и в задаче составления графика дежурств, мы реализуем этот подход, введя функцию стоимости и считая, что стоимость нарушения жесткого ограничения больше, чем стоимость использования лишних цветов. Тогда суммарная стоимость выступает в роли подлежащей минимизации функции приспособленности. Эту функциональность можно поместить в Python-класс, который мы опишем в следующем подразделе.

Представление задачи на Python

Для инкапсуляции задачи о раскраске графа мы написали Python-класс `GraphColoringProblem`, который находится по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/graphs.py.

При реализации этого класса мы воспользовались Python-пакетом с открытым исходным кодом NetworkX (https://networkx.github.io), который, среди прочего, позволяет работать с графами и изображать их. Граф, рассматриваемый в задаче о раскраске, будет экземпляром класса графа из NetworkX. Мы не будем создавать его с нуля, а воспользуемся уже включенными в библиотеку примерами, одним из которых является граф Петерсена.

Конструктор класса `GraphColoringProblem` принимает подлежащий раскраске граф в качестве параметра. Дополнительно он принимает параметр `hardConstraintPenalty`, представляющий штраф за нарушение жесткого ограничения.

Затем конструктор создает список вершин графа и матрицу смежности, которая позволяет быстро определить, являются ли две вершины смежными.

```
self.nodeList = list(self.graph.nodes)
self.adjMatrix = nx.adjacency_matrix(graph).todense()
```

Для вычисления количества нарушений ограничения на раскраску при заданном назначении цветов предназначен метод `getViolationsCount(colorArrangement)`.

Метод `getNumberOfColors(colorArrangement)` служит для вычисления количества цветов, используемых в данном назначении цветов.

Дополнительно класс предоставляет следующие открытые методы:

- `getCost(colorArrangement)`: вычисляет полную стоимость данного назначения цветов;
- `plotGraph(colorArrangement)`: изображает раскрашенный граф.

Метод `main` вызывает методы класса: создает экземпляр графа Петерсена и проверяет для него случайно выбранное назначение цветов (не более пяти). Кроме того, он устанавливает параметр `hardConstraintPenalty` равным 10:

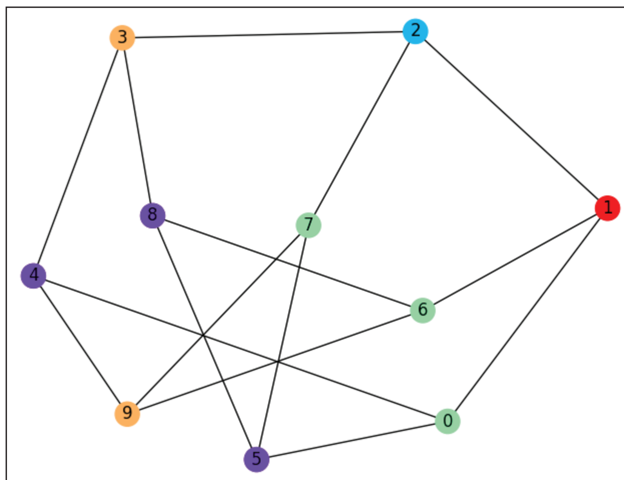
```
gcp = GraphColoringProblem(nx.petersen_graph(), 10)
solution = np.random.randint(5, size=len(gcp))
```

Результат может выглядеть следующим образом:

```
решение = [2 4 1 3 0 0 2 2 0 3]
число цветов = 5
число нарушений = 1
стоимость = 15
```

Поскольку в этом решении используется пять цветов и имеет место одно нарушение ограничения на раскраску, его стоимость равна 15.

Граф, соответствующий этому решению, изображен ниже. Видите ли вы, где нарушение ограничения?



Граф Петерсена, неправильно раскрашенный в пять цветов

В следующем подразделе мы применим генетический алгоритм, чтобы раскрасить граф без нарушений и одновременно минимизировать количество цветов.

Решение с помощью генетического алгоритма

Для решения задачи о раскраске графа мы написали на Python программу, которая находится в файле https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/03-solve-graphs.py.

Поскольку решение представлено списком целых чисел, нам придется немного модифицировать подход, применяемый к двоичному списку.

Ниже описаны шаги решения.

1. Сначала создается экземпляр класса `GraphColoringProblem`, которому передается объект подлежащего раскраске графа NetworkX – в данном случае уже знакомый нам граф Петерсена – и значение параметра `hardConstraintPenalty`, равное константе `HARD_CONSTRAINT_PENALTY`:

```
gcp = graphs.GraphColoringProblem(nx.petersen_graph(),
HARD_CONSTRAINT_PENALTY)
```

2. Поскольку мы стремимся минимизировать стоимость, определяем единственную цель – минимизирующую стратегию приспособления:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. Так как решение представлено списком целых чисел, соответствующих цветам, нобходимо определить генератор случайных чисел, который будет возвращать целое число от 0 до количества цветов минус 1. Это случайное число будет представлять цвет. Затем определим оператор, который порождает список таких случайных чисел длиной, равной количеству вершин графа, т. е. случайным образом назначим цвет каждой вершине. Наконец, определим оператор, который создает всю популяцию индивидуумов:

```
toolbox.register("Integers", random.randint, 0, MAX_COLORS - 1)
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.Integers, len(gcp))
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. Функция приспособленности вычисляет полную стоимость решения, включающую стоимость нарушений и число использованных цветов, вызывая метод `getCost()` класса `GraphColoringProblem`:

```
def getCost(individual):
    return gcp.getCost(individual),

toolbox.register("evaluate", getCost)
```

5. В качестве генетических операторов отбора и скрещивания можно использовать те же операции, что для двоичных списков. Однако оператор мутации придется изменить. Инвертирование бита заменяет 0 на 1 и наоборот, а здесь мы должны заменить одно целое число другим, тоже случайно сгенерированным и принадлежащим допустимому диапазону. Это делает оператор `mutUniformInt` – нужно только указать диапазон:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, low=0,
up=MAX_COLORS - 1, indpb=1.0/len(gcp))
```

6. Продолжаем использовать элитистский подход – копируем лучших на данный момент индивидуумов из зала славы в следующее поколение:

```
population, logbook = elitism.eaSimpleWithElitism(population,
toolbox, cxpb=P_CROSSOVER, mutpb=P_MUTATION,
ngen=MAX_GENERATIONS, stats=stats, halloffame=hof, verbose=True)
```

7. По завершении алгоритма печатаем сведения о лучшем найденном решении.

Прежде чем запускать программу, зададим параметры алгоритма:

```
POPULATION_SIZE = 100
P_CROSSOVER = 0.9
P_MUTATION = 0.1
MAX_GENERATIONS = 100
HALL_OF_FAME_SIZE = 5
```

Дополнительно зададим штраф за нарушение жестких ограничений – 10 и количество цветов – тоже 10:

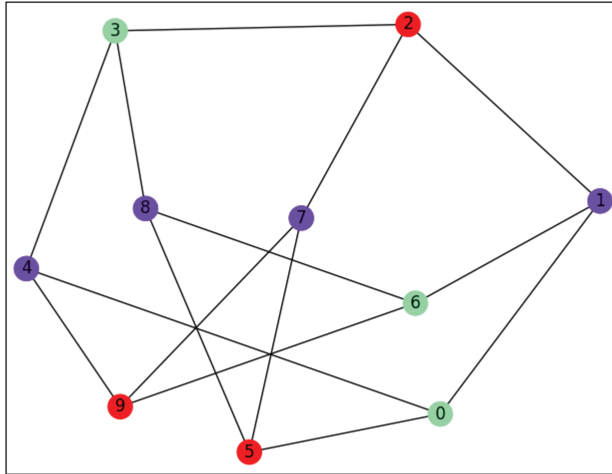
```
HARD_CONSTRAINT_PENALTY = 10
MAX_COLORS = 10
```

При запуске программы с этими параметрами печатается такой результат:

```
-- Лучший индивидуум = [5, 0, 6, 5, 0, 6, 5, 0, 0, 6]
-- Лучшая приспособленность = 3.0

число цветов = 3
число нарушений = 0
стоимость = 3
```

Таким образом, алгоритм смог найти правильную раскраску графа тремя цветами, обозначенными числами 0, 5, 6. Как уже было сказано, сами значения целых чисел несущественны – важно лишь, чтобы они были различны. Хроматическое число графа Петерсена действительно равно 3.

Эта программа создает следующее изображение, иллюстрирующее правильность решения:



Граф Петерсена,
правильно раскрашенный программой в три цвета

На следующем рисунке показаны графики зависимости минимальной и средней приспособленности от номера поколения. Видно, что решение найдено довольно быстро, потому что граф Петерсена невелик.

Статистика программы для задачи о раскраске графа Петерсена

Чтобы испытать программу на большем графе, возьмем граф Мычельского пятого порядка. Этот граф содержит 23 вершины и 7 ребер, известно, что его хроматическое число равно 5.

```
gcp = graphs.GraphColoringProblem(nx.mycielski_graph(5),
HARD_CONSTRAINT_PENALTY)
```

При тех же параметрах, что и выше, задав 10 цветов, мы получим следующие результаты:

```
-- Лучший индивидуум = [9, 6, 9, 4, 0, 0, 6, 5, 4, 5, 1, 5, 1, 1, 6, 6, 9, 5,
9, 6, 5, 1, 4]
-- Лучшая приспособленность = 6.0

число цветов = 6
число нарушений = 0
стоимость = 6
```

Но поскольку мы знаем, что хроматическое число этого графа равно 5, это решение не оптимально. А как найти оптимальное? И что, если бы хроматическое число не было известно заранее? Один из возможных подходов – изменить параметры генетического алгоритма, например увеличить размер популяции (и, наверное, размер зала славы) и (или) количество поколений. Другой подход – повторить поиск, уменьшив количество цветов. Поскольку алгоритм нашел решение с 6 цветами, уменьшим максимальное количество цветов до пяти и посмотрим, сможет ли алгоритм найти допустимое решение.

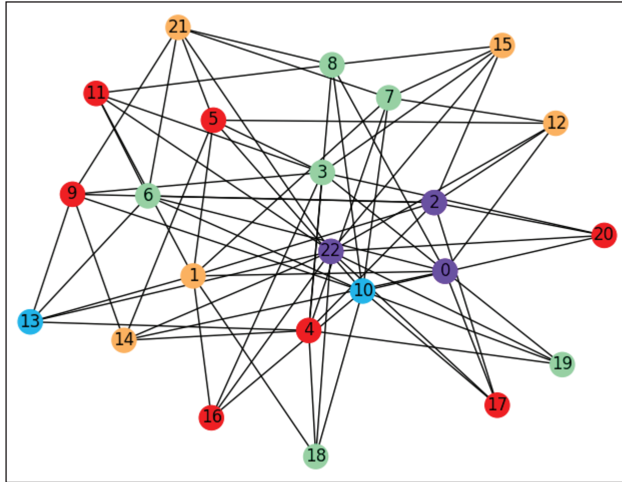
```
MAX_COLORS = 5
```

А с чего вдруг алгоритм найдет решение с пятью цветами сейчас, если не смог найти его с первой попытки? Уменьшив количество цветов с 10 до 5, мы значительно сократим пространство поиска – в данном случае с 10^{23} до 5^{23} (поскольку в графе 23 вершины), и у алгоритма появится больше шансов найти оптимальное решение даже за сравнительно короткое время и с ограниченным размером популяции. Поэтому даже если при первой попытке мы подошли достаточно близко к решению, имеет смысл уменьшать число цветов до тех пор, пока алгоритм не перестанет находить лучшие решения.

В нашем случае, начав с пяти цветов, алгоритм довольно быстро найдет нужное нам решение:

```
-- Лучший индивидуум = [0, 3, 0, 2, 4, 4, 2, 2, 2, 4, 1, 4, 3, 1, 3, 3, 4, 4,
2, 2, 4, 3, 0]
-- Лучшая приспособленность = 5.0

число цветов = 5
число нарушений = 0
стоимость = 5
```

Раскрашенный граф выглядит следующим образом:



Граф Мычельского,
правильно раскрашенный программой в пять цветов

Попытавшись уменьшить максимальное количество цветов до четырех, мы обязательно получим хотя бы одно нарушение.

Можете попробовать другие графы и поэкспериментировать с параметрами алгоритма.

Резюме

В этой главе мы познакомились с задачами с ограничениями, тесно связанными с ранее рассмотренными задачами комбинаторной оптимизации. Затем мы исследовали три классические задачи с ограничениями: задачу об *N* ферзях, задачу о составлении графика дежурств медсестер и задачу о раскраске графа. В каждой из них мы применили ставший уже привычным процесс: нахождение подходящего представления решения, создание класса, который инкапсулирует задачу и оценивает заданное решение, и написание генетического алгоритма, использующего этот класс. Попутно, занимаясь проверкой допустимости решения, мы познакомились с понятием жестких и мягких ограничений.

До сих пор мы сталкивались только с задачами поиска в дискретном пространстве состояний и переходов между ними. В следующей главе мы изучим задачи поиска в непрерывном пространстве и тем самым продемонстрируем гибкость генетических алгоритмов.

Для дальнейшего чтения

Дополнительную информацию по темам, рассмотренным в этой главе, см. в следующих источниках:

- задачи оптимизации с ограничениями: книга Prateek Joshi «Artificial Intelligence with Python»;
- введение в теорию графов: книга Alberto Boschetti, Luca Massaron «Python Data Science Essentials», второе издание;
- пособие по NetworkX: https://networkx.github.io/documentation/stable/tutorial.html.

Глава 6

Оптимизация непрерывных функций

Эта глава посвящена решению непрерывных задач оптимизации с помощью генетических алгоритмов. Мы начнем с описания хромосом и генетических операторов, которые обычно используются в популяциях с вещественным представлением, а затем перейдем к инструментам, предлагаемым каркасом DEAP в этой части. Далее рассмотрим несколько примеров задач непрерывной оптимизации и их решение на Python с применением каркаса DEAP. К ним относятся функция «подставка для яиц» (Eggholder), функция Химмельблау, а также условная оптимизация функции Симионеску. Попутно мы узнаем, как искать несколько решений, удовлетворяющих ограничениям, с помощью ниш и разделения.

В этой главе мы:

- узнаем о хромосомах и генетических операторах для работы с вещественными числами;
- воспользуемся каркасом DEAP для оптимизации непрерывных функций;
- оптимизируем функцию Eggholder;
- оптимизируем функцию Химмельблау;
- выполним условную оптимизацию функции Симионеску;
- воспользуемся параллельным и последовательным образованием ниш для нахождения оптимумов многомодальных функций.

Технические требования

В этой главе нам понадобится Python 3 и следующие библиотеки:

- `deap`;
- `numpy`;
- `matplotlib`;
- `seaborn`.

Код приведенных в данной главе программ можно найти в репозитории книги на GitHub по адресу https://github.com/PacktPublishing/Hands-On-Genetic-

Algorithms-with-Python/tree/master/Chapter06. Чтобы увидеть код в действии, посмотрите видео по адресу http://bit.ly/2REfGuT.

Хромосомы и генетические операторы для задач с вещественными числами

В предыдущих главах мы решали задачи, в которых требовалось методично оценивать состояния и переходы между состояниями. Решения этих задач было удобно представлять списками (или массивами) двоичных или целых чисел. Но в этой главе мы будем рассматривать задачи с непрерывным пространством состояний, когда решения представлены вещественными числами с плавающей точкой. В главе 2 мы говорили, что представление вещественных чисел двоичными или целочисленными списками – далеко не идеальный способ, а проще и лучше иметь дело со списками вещественных чисел.

В главе 2 мы уже приводили в качестве примера задачу с тремя вещественными параметрами, в которой хромосома имеет вид

$[x_1, x_2, x_3]$

где x_1, x_2, x_3 – вещественные числа, например:

```
[1.23, 7.2134, -25.309] или [-30.10, 100.2, 42.424].
```

Кроме того, мы отмечали, что методы отбора работают одинаково как для целых, так и для вещественных хромосом, но методы скрещивания и мутации необходимо разрабатывать для вещественных хромосом специально. Обычно они применяются к соответственным элементам списка по отдельности, как описано ниже.

Рассмотрим две хромосомы: $[x_1, x_2, x_3]$ и $[y_1, y_2, y_3]$. Применение операции скрещивания к соответственным генам следующим образом порождает потомка:

- o_1 – результат применения оператора скрещивания к x_1 и y_1;
- o_2 – результат применения оператора скрещивания к x_2 и y_2;
- o_3 – результат применения оператора скрещивания к x_3 и y_3.

Аналогично оператор мутации применяется к каждому элементу по отдельности, поэтому o_1, o_2 и o_3 могут быть подвергнуты мутации.

Перечислим некоторые распространенные вещественные операторы.

- **Скрещивание смешением** (Blend Crossover – **BLX**), когда каждый потомок случайным образом выбирается из интервала, образованного родителями:

  $$[parent_1 - \alpha(parent_2 - parent_1), parent_2 + \alpha(parent_2 - parent_1)].$$

  Коэффициент α обычно принимается равным 0.5, поэтому интервал выбора в два раза шире интервала между родителями.
- **Имитация двоичного скрещивания** (Simulated Binary Crossover – SBX), когда два потомка порождаются по следующей формуле, гаран-

тирующей, что среднее значение потомков равно среднему значению родителей:

$offsping_1 = ½ [(1 + β]parent_1 + (1 – β)parent_2]);$
$offsping_2 = ½ [(1 – β]parent_1 + (1 + β)parent_2]).$

Коэффициент β, называемый **коэффициентом распределения**, вычисляется в виде комбинации случайно выбранного значения и заранее заданного параметра η, называемого индексом распределения, или **коэффициентом скученности** (crowding factor). Чем больше значение η, тем больше потомки похожи на своих родителей. Обычно η выбирается в диапазоне от 10 до 20.

- Нормально распределенная (или гауссова) мутация, когда исходное значение заменяется случайным числом, выбираемым из нормального распределения с заданными средним и стандартным отклонением.

В следующем разделе мы увидим, как вещественные хромосомы и генетические операторы поддерживаются каркасом DEAP.

Использование DEAP совместно с непрерывными функциями

При решении задач непрерывного поиска каркас DEAP можно использовать для оптимизации функций почти так же, как мы делали до сих пор. Нужно лишь внести несколько незначительных изменений.

Для кодирования хромосом используется список (или массив) чисел с плавающей точкой. Но следует помнить, что встроенные в DEAP генетические операторы не будут работать с индивидуумами, расширяющими класс `numpy.ndarray`, вследствие особенностей операций срезки и сравнения объектов.

Чтобы можно было работать с индивидуумами на основе `numpy.ndarray`, мы должны переопределить генетические операторы. Это описано в документации по DEAP, в разделе «Inheriting from NumPy». По данной причине, а также из соображений производительности обычно при работе с DEAP отдают предпочтение обыкновенным спискам или массивам вещественных чисел, встроенным в сам язык Python.

Что касается вещественных генетических операторов, то каркас DEAP предлагает несколько готовых реализаций в модулях `crossover` и `mutation`:

- `cxBlend()` – реализация скрещивания смешением, коэффициент α передается в параметре `alpha`;
- `cxSimulatedBinary()` – реализация имитации двоичного скрещивания, величина η передается в параметре `eta`;
- `mutGaussian()` – реализация нормально распределенной мутации, среднее и стандартное отклонение передаются в аргументах `mu` и `sigma`.

Кроме того, поскольку непрерывные функции обычно оптимизируются в ограниченной области, а не во всем пространстве, DEAP предоставляет два

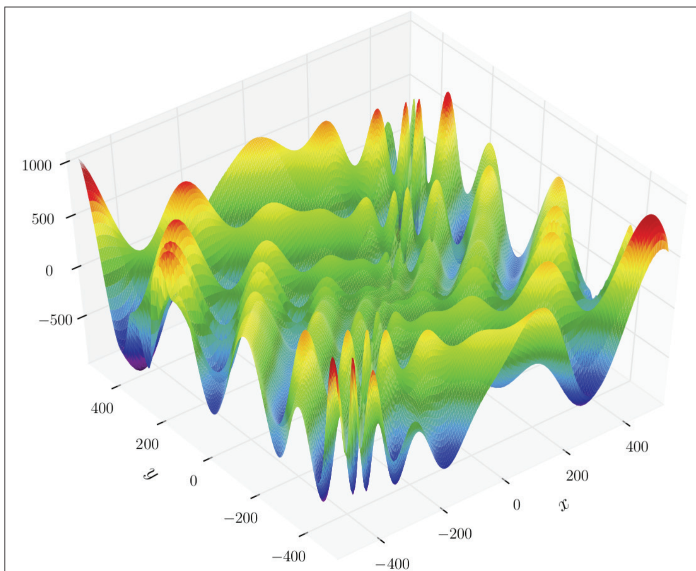
оператора, которые принимают параметры, описывающие границу, и гарантируют, что результирующие индивидуумы не выйдут за эти границы:

- `cxSimulatedBinaryBounded()` – ограниченный вариант оператора `cxSimulatedBinary()`, принимающий аргументы `low` и `up` – нижнюю и верхнюю границы области поиска соответственно;
- `mutPolynomialBounded()` – ограниченный оператор мутации, в котором распределение вероятностей задается полиномиальной (а не гауссовой) функцией. Этот оператор также принимает аргументы `low` и `up` – нижнюю и верхнюю границы области поиска. Кроме того, он принимает коэффициент скученности (параметр `eta`) – чем он больше, тем ближе мутант к исходному значению. Соответственно, при малых значениях `eta` мутант будет сильно отличаться от оригинала.

В следующем разделе мы продемонстрируем использование ограниченных операторов на примере оптимизации классической тестовой функции.

Оптимизация функции Eggholder

Функция Eggholder (она называется так потому, что ее форма напоминает подставку для яиц) часто используется для тестирования алгоритмов оптимизации. Нахождение единственного глобального минимума этой функции считается трудной задачей из-за большого количества локальных минимумов.



Функция Eggholder

Источник: https://en.wikipedia.org/wiki/File:Eggholder_function.pdf.
Автор: Gaortizg. Публикуется по лицензии Creative Commons CC BY-SA 3.0:
https://creativecommons.org/licenses/by-sa/3.0/deed.en

Математически функция описывается выражением

$$f(x,y) = -(y+47)\cdot\sin\sqrt{\left|\frac{x}{2}+(y+47)\right|} - x\cdot\sin\sqrt{|x-(y+47)|}$$

и обычно вычисляется в области поиска, ограниченной интервалом [–512, 512] по каждому измерению.

Известно, что глобального минимума эта функция достигает в точке

x=512, *y = 404.2319*,

в которой равна –959.6407.

В следующем подразделе мы попробуем найти глобальный минимум, применив генетический алгоритм.

Оптимизация функции Eggholder с помощью генетического алгоритма

Для оптимизации функции Eggholder с помощью генетического алгоритма мы написали программу, которая находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter06/01-optimize-eggholder.py.

Ниже описаны основные шаги программы.

1. Сначала определяются константы, а именно: количество измерений 2 (т. к. функция определена на плоскости *xy*) и границы области:

```
DIMENSIONS = 2                           # количество измерений
BOUND_LOW, BOUND_UP = -512.0, 512.0 # границы, одинаковые для всех измерений
```

2. Поскольку мы работаем с вещественными числами, ограниченными некоторой областью, далее определим вспомогательную функцию, порождающую числа с плавающей точкой, равномерно распределенные в этой области:

В этой функции предполагается, что верхняя и нижняя границы по всем измерениям одинаковы.

```
def randomFloat(low, up):
    return [random.uniform(l, u) for l, u in zip([low] *
DIMENSIONS, [up] * DIMENSIONS)]
```

3. Далее определим оператор `attrFloat`. В нем только что определенная функция используется для порождения одного случайного числа с плавающей точкой в заданном диапазоне. Этот оператор будет затем использован оператором `individualCreator` для создания случайных индивидуумов. А тот, в свою очередь, – оператором `populationCreator`, порождающим заданное количество индивидуумов.

```
toolbox.register("attrFloat", randomFloat, BOUND_LOW, BOUND_UP)
toolbox.register("individualCreator", tools.initIterate,
creator.Individual, toolbox.attrFloat)
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. Мы собираемся минимизировать функцию Eggholder, поэтому ее и будем использовать как функцию приспособленности. Поскольку индивидуум представлен списком чисел с плавающей точкой длины 2, выделим из него значения *x* и *y* и вычислим функцию:

```
def eggholder(individual):
    x = individual[0]
    y = individual[1]
    f = (-(y + 47.0) * np.sin(np.sqrt(abs(x/2.0 + (y + 47.0))))
- x * np.sin(np.sqrt(abs(x - (y + 47.0)))))
    return f,   # вернуть кортеж

toolbox.register("evaluate", eggholder)
```

5. Далее разберемся с генетическими операторами. Поскольку оператор отбора не зависит от типа индивидуума, а у нас уже есть удачный опыт использования турнирного отбора размера 2 в сочетании с элитизмом, то и будем продолжать в том же духе. С другой стороны, операторы скрещивания и мутации нужно специализировать для чисел с плавающей точкой в заданных границах, поэтому воспользуемся предоставленным DEAP операторами `cxSimulatedBinaryBounded` для скрещивания и `mutPolynomialBounded` для мутации:

```
# Генетические операторы
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxSimulatedBinaryBounded,
low=BOUND_LOW, up=BOUND_UP, eta=CROWDING_FACTOR)
toolbox.register("mutate", tools.mutPolynomialBounded,
low=BOUND_LOW, up=BOUND_UP, eta=CROWDING_FACTOR,
indpb=1.0/DIMENSIONS)
```

6. Как и раньше, воспользуемся модифицированным вариантом простого алгоритма генетического поиска, встроенного в DEAP, в который мы добавили элитизм, – будем сохранять лучших индивидуумов (в зале славы) и копировать их в следующее поколение без изменения:

```
population, logbook = elitism.eaSimpleWithElitism(population,
toolbox, cxpb=P_CROSSOVER, mutpb=P_MUTATION,
ngen=MAX_GENERATIONS, stats=stats, halloffame=hof, verbose=True)
```

7. Начнем с задания параметров генетического алгоритма. Поскольку оптимизировать функцию Eggholder нелегко, зададим относительно большую популяцию, при небольшом числе измерений на это можно пойти:

```
# Параметры генетического алгоритма
POPULATION_SIZE = 300
P_CROSSOVER = 0.9
P_MUTATION = 0.1
MAX_GENERATIONS = 300
HALL_OF_FAME_SIZE = 30
```

8. В дополнение к предыдущим параметрам появляется один новый – коэффициент скученности (`eta`), задействованный в операциях скрещивания и мутации:

```
CROWDING_FACTOR = 20.0
```

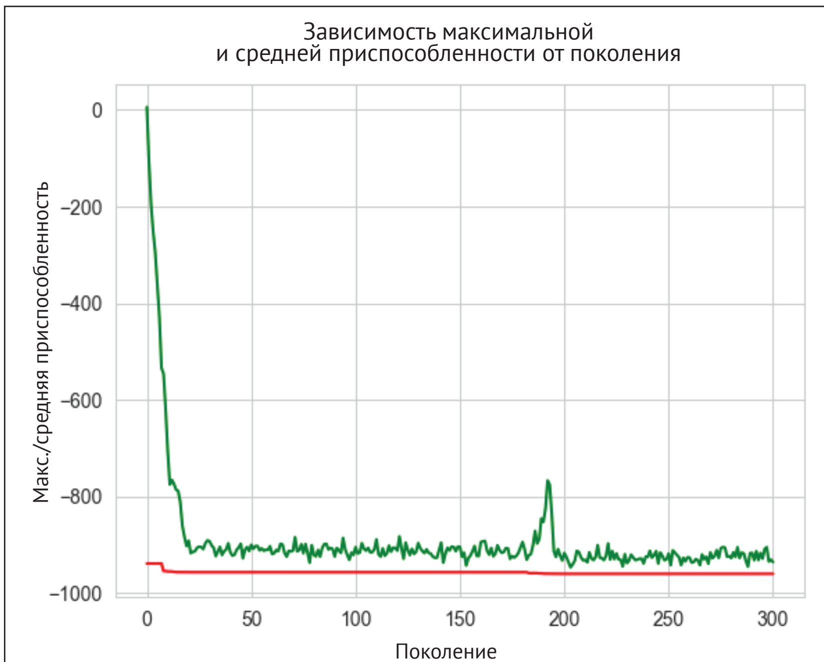
Можно также определить отдельные коэффициенты скученности для скрещивания и для мутации.

Вот теперь мы готовы запустить программу. Ниже показаны результаты, полученные с заданными параметрами:

```
-- Лучший индивидуум = [512.0, 404.23180541839946]
-- Лучшая приспособленность = -959.6406627208509
```

Это означает, что мы нашли глобальный минимум.

Из показанных ниже графиков статистики видно, что алгоритм сразу же нашел какой-то локальный минимум, а затем производил небольшие улучшения, пока в конечном итоге не вышел на глобальный минимум.



Статистика первой программы оптимизации функции Eggholder

Интересно, что произошло в окрестности поколения 180 – мы изучим этот феномен в следующем подразделе.

Повышение скорости сходимости посредством увеличения частоты мутаций

Увеличив масштаб в нижней части графика приспособленности, мы заметим довольно сильный скачок лучшего найденного результата (красная линия) в районе поколения 180, сопровождаемый резкими колебаниями средних результатов (зеленая линия).

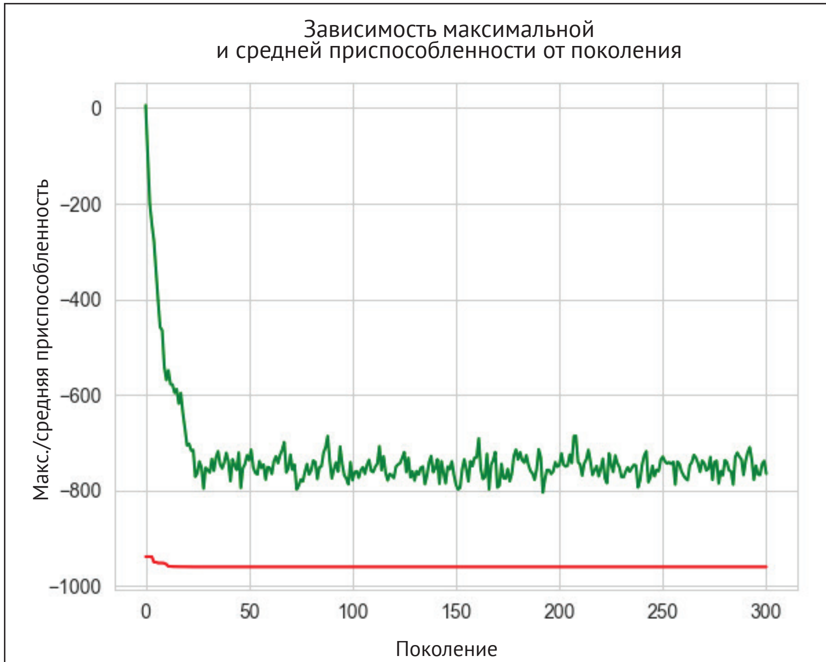
Увеличенная часть графиков статистики первой версии программы

Это наблюдение можно объяснить тем, что увеличение шума приводит к более быстрому получению хороших результатов. Возможно, это еще одно проявление уже знакомой нам дилеммы **исследования–использования** – увеличение степени исследования (которое выглядит на рисунке как шум) может ускорить отыскание глобального минимума. Простой способ повысить степень исследования – увеличить вероятность мутации. По счастью, использование элитизма – сохранение лучших результатов – не даст нам зайти в исследовании слишком далеко и сделать поиск случайным.

Для проверки этой идеи увеличим вероятность мутации с 0.1 до 0.5:

```
P_MUTATION = 0.5
```

Модифицированная таким образом программа находит глобальный минимум гораздо быстрее – это видно по графику, на котором красная линия (лучший результат) достигает оптимума быстро, а зеленая линия (средняя приспособленность) сильнее флуктуирует и дальше отстоит от лучшего результата.

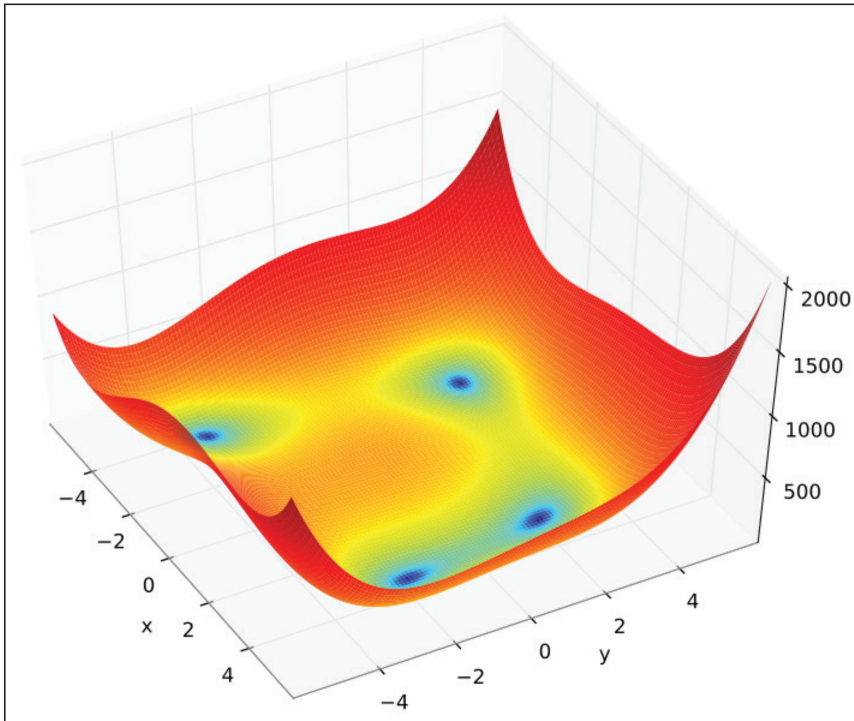


Статистика программы оптимизации функции Eggholder
с увеличенной вероятностью мутации

Будем иметь в виду эту идею в следующем разделе, работая с тестовой функцией Химмельблау.

Оптимизация функции Химмельблау

На рисунке ниже изображена функция Химмельблау, которая также часто используется для тестирования алгоритмов оптимизации.



Функция Химмельблау

Источник: https://commons.wikimedia.org/wiki/File:Himmelblau_function.svg.
Автор: Morn the Gorn. Является общественным достоянием

Математически эта функция описывается следующей формулой:

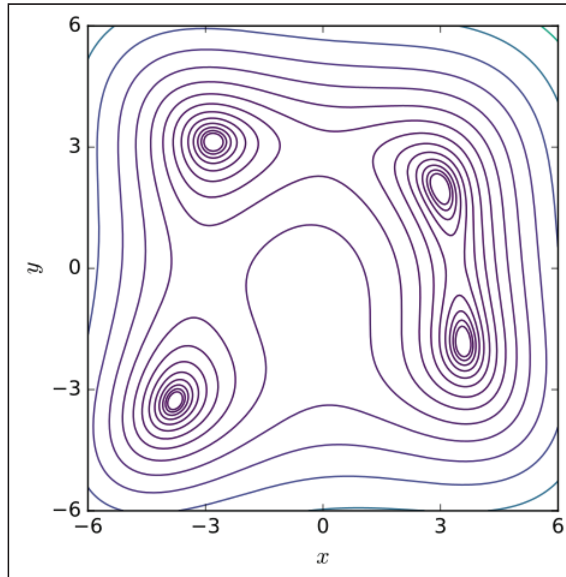
$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2.$

Обычно она вычисляется в области, ограниченной интервалом [–5, 5] по каждому измерению.

Хотя эта функция кажется проще функции Eggholder, особый интерес представляет ее *многомодальность*, т. е. наличие нескольких глобальных минимумов. Точнее, у функции имеется четыре глобальных минимума, равных 0, в следующих точках:

- $x = 3.0$, $y = 2.0$;
- $x = -2.805118$, $y = 3.131312$;
- $x = -3.779310$, $y = -3.283186$;
- $x = 3.584458$, $y = -1.848126$.

Эти точки показаны на следующей диаграмме линий уровня:



Линии уровня функции Химмельблау

Источник: https://commons.wikimedia.org/wiki/File:Himmelblau_contour.svg.
Автор: Nicoguaro. Публикуется по лицензии Creative Commons CC BY 4.0:
https://creativecommons.org/licenses/by/4.0/deed.en

В процессе оптимизации многомодальных функций часто требуется найти все минимумы (или хотя бы большинство их). Но в следующем подразделе мы начнем с нахождения одного.

Оптимизация функции Химмельблау с помощью генетического алгоритма

Программа нахождения одного минимума функции Химмельблау находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter06/02-optimize-himmelblau.py.

Эта программа похожа на программу оптимизации функции Eggholder, но имеет несколько отличий:

1. Область оптимизации ограничена интервалом [−5.0, 5.0] по всем измерениям:

```
BOUND_LOW, BOUND_UP = -5.0, 5.0 # границы по всем измерениям
```

2. В качестве функции приспособленности используется функция Химмельблау:

```
def himmelblau(individual):
    x = individual[0]
    y = individual[1]
    f = (x ** 2 + y - 11) ** 2 + (x + y ** 2 - 7) ** 2
    return f,   # вернуть кортеж

toolbox.register("evaluate", himmelblau)
```

3. Поскольку оптимизируемая функция имеет несколько минимумов, интересно в конце прогона понаблюдать за распределением найденных решений. Поэтому добавим диаграмму рассеяния, показывающую все четыре глобальных минимума и окончательную популяцию на одной и той же плоскости *xy*:

```
plt.figure(1)
globalMinima = [[3.0, 2.0], [-2.805118, 3.131312], [-3.779310, -3.283186],
[3.584458, -1.848126]]
plt.scatter(*zip(*globalMinima), marker='X', color='red', zorder=1)
plt.scatter(*zip(*population), marker='.', color='blue', zorder=0)
```

4. Также распечатаем содержимое зала славы – лучших найденных индивидуумов:

```
print("- Лучшие решения:")
for i in range(HALL_OF_FAME_SIZE):
    print(i, ": ", hof.items[i].fitness.values[0], " -> ",
hof.items[i])
```

Результаты, напечатанные после прогона программы, показывают, что мы нашли один из четырех минимумов ($x = 3.0$, $y = 2.0$):

```
-- Лучший индивидуум = [2.9999999999987943, 2.0000000000007114]
-- Лучшая приспособленность = 4.523490304795033e-23
```

Из распечатки зала славы видно, что все его члены представляют одно и то же решение:

```
- Лучшие решения:
0 : 4.523490304795033e-23 -> [2.9999999999987943, 2.0000000000007114]
1 : 4.523732642865117e-23 -> [2.9999999999987943, 2.000000000000697]
2 : 4.523900512465748e-23 -> [2.9999999999987943, 2.0000000000006937]
3 : 4.5240633333565856e-23 -> [2.9999999999987943, 2.00000000000071]
...
```

Следующий рисунок, на котором изображено распределение популяции, подтверждает, что генетический алгоритм сошелся к одному из четырех минимумов функции – находящемуся в точке (x = 3.0, y = 2.0):

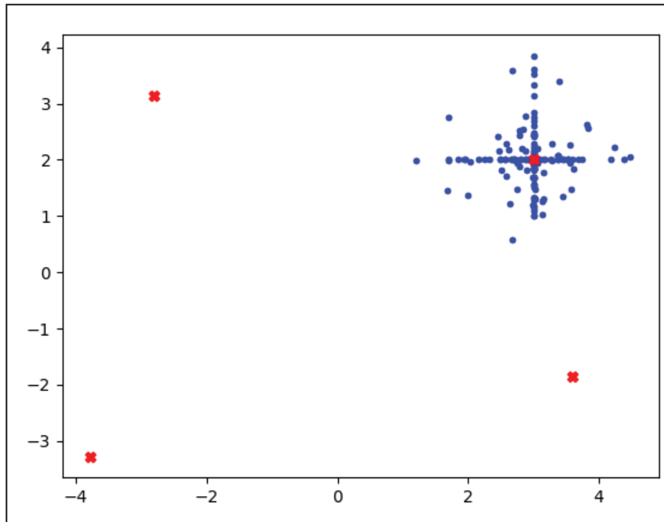


Диаграмма рассеяния популяции в конце первого прогона
и все четыре минимума функции

Кроме того, видно, что для многих индивидуумов в популяции одна из координат x и y совпадает с координатой найденного минимума.

Полученные результаты типичны для генетического алгоритма – он находит глобальный минимум и сходится к нему. Поскольку в данном случае минимумов несколько, ожидается сходимость к какому-то одному. К какому именно, зависит от случайной инициализации алгоритма. Как вы, наверное, помните, мы задавали фиксированное начальное значение генератора случайных чисел (42):

```
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

Это сделано для того, чтобы обеспечить повторяемость результатов, но на практике обычно задаются разные начальные значения для разных прогонов – для этого нужно либо закомментировать приведенные выше строки, либо явно задавать различные значения константы.

Например, если задать начальное значение 13, то будет найдено решение (x = −2.805118, y = 3.131312), показанное на следующем рисунке.

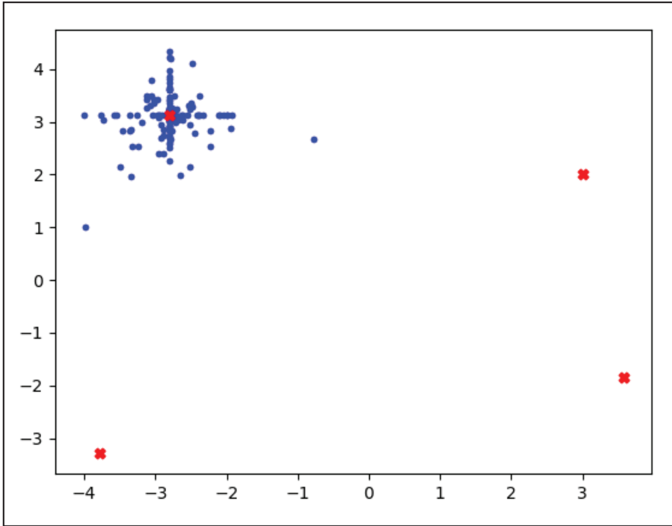


Диаграмма рассеяния популяции в конце второго прогона
и все четыре минимума функции

Если далее задать начальное значение 17, то программа найдет решение (x = 3.584458, y = –1.848126), как видно по следующему рисунку.

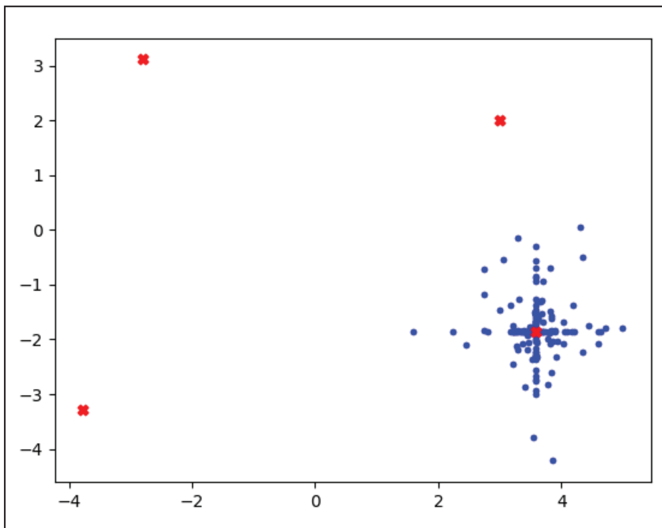


Диаграмма рассеяния популяции в конце третьего прогона
и все четыре минимума функции

Но что, если мы хотим найти все четыре глобальных минимума за один прогон? Как показано в следующем подразделе, генетические алгоритмы предлагают и такую возможность.

Использование ниш и разделения для отыскания нескольких решений

В главе 2 мы упоминали о нишах и разделении в генетических алгоритмах как способе смоделировать несколько частей – ниш – в естественной окружающей среде. Эти ниши заселены различными видами, или субпопуляциями, которые используют уникальные ресурсы, доступные в данной нише. Особи, сосуществующие в одной нише, конкурируют за ресурсы. Реализация механизма разделения в генетическом алгоритме побуждает индивидуумов исследовать новые ниши и может использоваться для нахождения нескольких оптимальных решений, каждое из которых трактуется как отдельная ниша. Один из способов обеспечить разделение – модифицировать значение приспособленности каждого индивидуума с помощью какой-то функции от комбинации расстояний до всех остальных индивидуумов и тем самым штрафовать популяцию за скученность посредством распределения локальных ресурсов между ее индивидуумами.

Попробуем применить эту идею к оптимизации функции Химмельблау и посмотрим, поможет ли это найти все четыре минимума в одном прогоне. Эта попытка реализована в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter06/03-optimize-himmelblau-sharing.py.

Данная программа основана на предыдущей, но мы внесли кое-какие важные изменения.

1. Прежде всего для реализации механизма разделения обычно требуется оптимизировать функцию, возвращающую положительные значения приспособленности, и искать максимумы, а не минимумы. Это позволит применить деление исходных значений приспособленности как способ уменьшить приспособленность и практически распределить ресурсы между соседними индивидуумами. Значения функции Химмельблау находятся в диапазоне от 0 до (примерно) 2000, поэтому мы можем модифицировать ее, возвращая 2000 минус исходное значение. Тогда все значения новой функции будут положительны, а минимумы станут максимумами. При этом положение оптимумов не изменится, так что наша цель по-прежнему состоит в отыскании этих точек.

```
def himmelblauInverted(individual):
    x = individual[0]
    y = individual[1]
    f = (x ** 2 + y - 11) ** 2 + (x + y ** 2 - 7) ** 2
    return 2000.0 - f, # return a tuple

toolbox.register("evaluate", himmelblauInverted)
```

2. Для завершения преобразования переопределим стратегию приспособления, сделав ее максимизирующей:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

3. Определим две новые константы:

```
DISTANCE_THRESHOLD = 0.1
SHARING_EXTENT = 5.0
```

4. Теперь нужно реализовать сам механизм разделения. Удобно сделать это внутри оператора отбора. Именно там проверяются значения приспособленности всех индивидуумов и отбираются родители для следующего поколения. Это позволяет включить код, который будет пересчитывать приспособленности перед отбором и в целях мониторинга сохранять исходные значения. Для этого мы реализовали новую функцию `selTournamentWithSharing()` с такой же сигнатурой, как у оригинальной функции `tools.selTournament()`, которой мы пользовались до сих пор:

```
def selTournamentWithSharing(individuals, k, tournsize,
fit_attr="fitness"):
```

Эта функция сначала запоминает исходные значения приспособленности, чтобы впоследствии их можно было достать. Затем она перебирает всех индивидуумов и для каждого находит величину `sharingSum`, на которую нужно поделить его приспособленность. Она вычисляется как сумма расстояний между текущим индивидуумом и всеми остальными индивидуумами в популяции. Если расстояние меньше порога `DISTANCE_THRESHOLD`, то к сумме добавляется величина

$$1 - \frac{distance}{DISTANCE_THRESHOLD} \times \frac{1}{SHARING_EXTEND}.$$

Это означает, что приспособленность будет уменьшаться на большую величину, когда:

- нормированное расстояние между индивидуумами меньше;
- константа `SHARING_EXTENT` больше.

После пересчета приспособленностей всех индивидуумов производится турнирный отбор с использованием новых значений приспособленности:

```
selected = tools.selTournament(individuals, k, tournsize, fit_attr)
```

Наконец, мы восстанавливаем исходные приспособленности:

```
for i, ind in enumerate(individuals):
    ind.fitness.values = origFitnesses[i],
```

5. И последний штрих – добавить график, на котором показаны места расположения лучших индивидуумов – попавших в зал славы – на плоскости *xy*, а также известные точки оптимумов, как мы уже делали раньше:

```
plt.figure(2)
plt.scatter(*zip(*globalMaxima), marker='x', color='red', zorder=1)
plt.scatter(*zip(*hof.items), marker='.', color='blue', zorder=0)
```

Результаты работы этой программы нас не разочаруют. Распечатка зала славы показывает, что мы нашли все четыре оптимума:

```
- Лучшие решения:
0 : 1999.9997428476076 -> [3.00161237138945, 1.9958270919300878]
1 : 1999.9995532774788 -> [3.585506608049694, -1.8432407550446581]
2 : 1999.9988186889173 -> [3.585506608049694, -1.8396197402430106]
3 : 1999.9987642838498 -> [-3.7758887140006174, -3.285804345540637]
4 : 1999.9986563457114 -> [-2.8072634380293766, 3.125893564009283]
...
```

Следующий рисунок, на котором показано распределение индивидуумов из зала славы, подтверждает этот вывод.

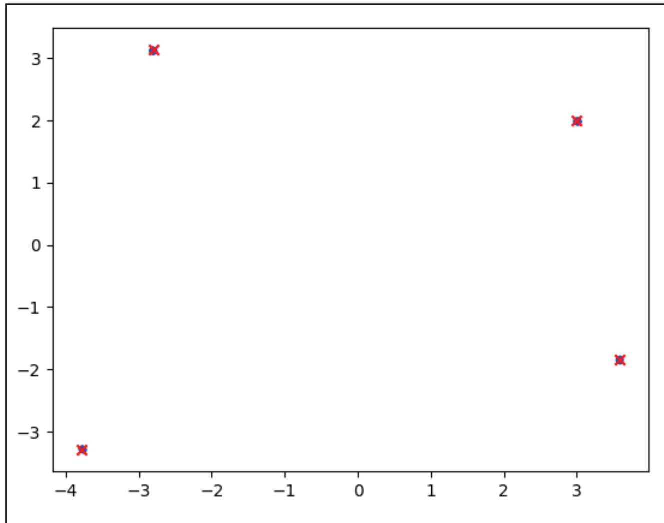


Диаграмма рассеяния решений в конце прогона
с использованием ниш и все четыре минимума функции

А на следующей диаграмме рассеяния показано распределение всей популяции вокруг четырех найденных решений:

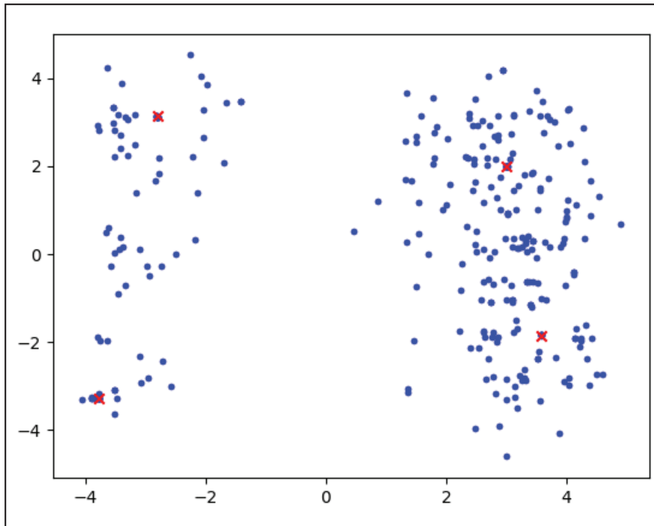


Диаграмма рассеяния популяции в конце прогона
с использованием ниш и все четыре минимума функции

Полученный результат выглядит впечатляюще, но нужно помнить, что в реальных задачах реализовать этот подход может быть труднее. Во-первых, изменения, внесенные нами в процедуру отбора, увеличивают сложность вычислений и время работы алгоритма. Кроме того, размер популяции обычно приходится увеличивать, чтобы она могла покрыть все интересующие области. В некоторых случаях трудно подобрать значения констант, относящихся к разделению, например если мы заранее не знаем, насколько пики близки друг к другу. Но эту технику всегда можно использовать, чтобы в первом приближении найти интересующие области, а затем исследовать каждую из них, применив стандартную версию алгоритма.

Альтернативный подход к нахождению нескольких точек оптимума – условная оптимизация. Это тема следующего раздела.

Функция Симионеску и условная оптимизация

На первый взгляд, функция Симионеску не выглядит особенно интересной. Однако с ней связаны ограничения, благодаря которым с ней очень увлекательно работать, да и на вид она приятна.

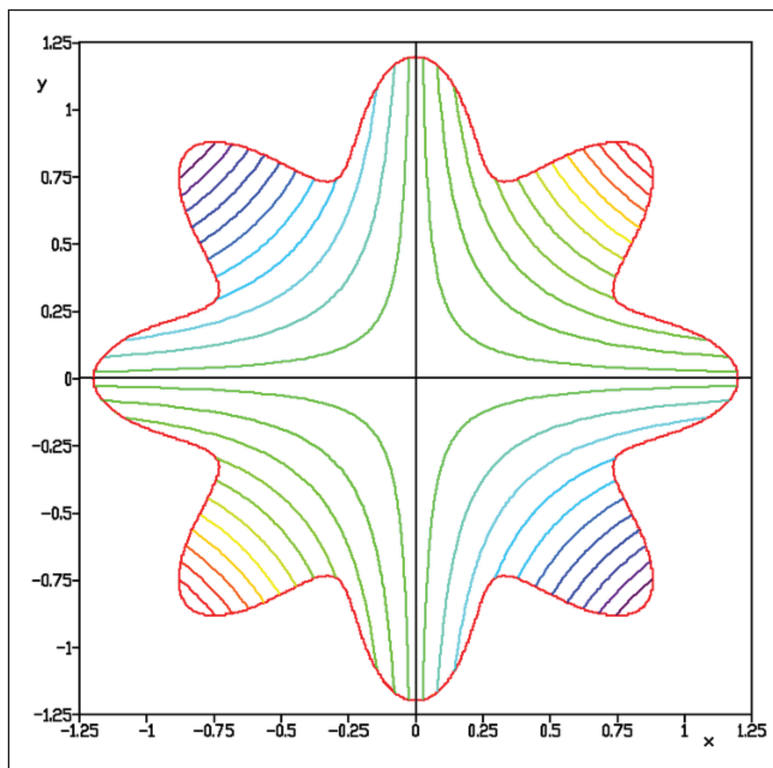
Эта функция обычно вычисляется в пространстве поиска, ограниченном интервалом [−1.25, 1.25] по каждому измерению, а математически она описывается формулой

$$f(x, y) = 0.1xy,$$

где на x, y накладывается следующее условие:

$$x^2 + y^2 \leq \left[1 + 0.2 \cdot \cos\left(8 \cdot \text{arctg}\frac{x}{y}\right)\right].$$

Это условие ограничивает значения *x* и *y*, считающиеся допустимыми для функции Симионеску. Результат показан на следующей диаграмме линий уровня:



Линии уровня ограниченной функции Симионеску

Источник: https://commons.wikimedia.org/wiki/File:Simionescu%27s_function.PNG.
Автор: Simiprof. Публикуется по лицензии Creative Commons CC BY-SA 3.0:
https://creativecommons.org/licenses/by-sa/3.0/deed.en

Граница в форме цветка обусловлена ограничением, а цвета линий уровня обозначают фактические значения – красный соответствует бóльшим, фиолетовый – меньшим. Если бы не ограничение, то минимумы находились бы в точках (1.25, –1.25) и (–1.25, 1.25). Но из-за ограничения глобальные минимумы функции находятся в точках:

- $x = 0.84852813$, $y = -0.84852813$;
- $x = -0.84852813$, $y = 0.84852813$.

Это кончики двух противоположных лепестков, содержащих фиолетовые линии уровня. В обеих точках достигается минимальное значение –0.072.

В следующем подразделе мы попробуем найти эти минимумы с помощью генетического алгоритма для вещественных чисел.

Условная оптимизация с помощью генетического алгоритма

Мы уже имели дело с ограничениями в главе 5, когда рассматривали задачи поиска. Но если в задачах поиска под ограничениями понимались недопустимые состояния или комбинации, то здесь ограничения определены в непрерывном пространстве в виде математических неравенств.

Однако подходы в обоих случаях похожи, а различаются только реализации. Еще раз перечислим эти подходы.

- Лучше всего исключить саму возможность нарушения ограничения – если это удается. Мы так и поступали в этой главе с самого начала, когда задавали ограниченные области определения функций. По существу, это простые ограничения на каждую входную переменную. Мы смогли соблюсти их, генерируя начальные популяции в заданных границах и используя ограниченные генетические операторы типа `cxSimulatedBinaryBounded()`, которые не выводят за границы областей. К сожалению, этот подход становится труднее реализовать, когда ограничения сложнее, чем просто верхняя и нижняя границы по каждому измерению.
- Второй подход заключается в том, чтобы отбрасывать решения, нарушающие хотя бы одно ограничение. Как мы уже отмечали, при этом теряется информация, содержащаяся в отброшенных решениях, что может сильно замедлить процесс оптимизации.
- Следующий подход – исправлять решения, которые нарушают ограничения, т. е. модифицировать их так, чтобы нарушения были устранены. Но часто это трудно реализовать, к тому же процесс исправления может привести к значительной потере информации.
- Наконец, в главе 5 нас выручил подход, заключающийся в том, чтобы штрафовать решения, которые нарушают ограничения, уменьшая оценку решения и делая его менее желательным. В задачах поиска мы реализовали его, создав функцию стоимости, которая увеличивала стоимость на фиксированную величину для каждого нарушения ограничения. В непрерывном пространстве мы можем либо использовать фиксированный штраф, либо увеличивать его в зависимости от серьезности нарушения.

В последнем подходе – штрафовании оценки за нарушение ограничений – можно использовать предлагаемый каркасом DEAP механизм *штрафной функции*, который мы продемонстрируем в следующем подразделе.

Оптимизация функции Симионеску с помощью генетического алгоритма

Программа оптимизации функции Симионеску находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter06/04-optimize-simionescu.py.

Эта программа очень похожа на первую программу в данной главе, предназначенную для оптимизации функции Eggholder, но имеет следующие отличия:

1. Константы, задающие границы, соответствуют области определения функции Симионеску:

```
BOUND_LOW, BOUND_UP = -1.25, 1.25
```

2. Добавлена новая константа, задающая фиксированный штраф за нарушение ограничения:

```
PENALTY_VALUE = 10.0
```

3. Приспособленность теперь определяется функцией Симионеску:

```
def simionescu(individual):
    x = individual[0]
    y = individual[1]
    f = 0.1 * x * y
    return f,   # вернуть кортеж

toolbox.register("evaluate",simionescu)
```

4. Дальше начинается самое интересное: мы определяем новую функцию `feasible()`, которая задает допустимую часть области определения с помощью ограничения. Она возвращает `True`, если значения `x`, `y` удовлетворяют ограничениям, и `False` в противном случае:

```
def feasible(individual):
    x = individual[0]
    y = individual[1]
    return x**2 + y**2 <= (1 + 0.2 * math.cos(8.0 * math.atan2(x, y)))**2
```

5. Затем мы используем оператор DEAP `toolbox.decorate()` в сочетании с функцией `tools.DeltaPenalty()`, чтобы модифицировать (декорировать) исходную функцию приспособленности – штрафовать за несоблюдение ограничений. Функция `DeltaPenalty` принимает функцию `feasible()` и фиксированную величину штрафа в качестве параметров:

```
toolbox.decorate("evaluate", tools.DeltaPenalty(feasible, PENALTY_VALUE))
```

Функция `DeltaPenalty` может принимать еще третий параметр, который представляет расстояние от допустимой области и налагает больший штраф за увеличение этого расстояния.

Теперь можно запускать программу. Результаты показывают, что мы действительно нашли один из двух известных минимумов:

```
-- Лучший индивидуум = [0.8487712463169383, -0.8482833185888866]
-- Лучшая приспособленность = -0.07199984895485578
```

А что насчет второго? Читайте дальше – мы найдем его в следующем подразделе.

Использование ограничений для нахождения нескольких решений

Ранее в этой главе при оптимизации функции Химмельблау мы думали о том, как найти больше одного минимума, и предложили два способа сделать это – изменить начальное значение генератора случайных чисел или воспользоваться нишами и разделением. Здесь мы продемонстрируем третий способ, основанный на ограничениях!

Ту технику, которую мы применили для функции Химмельблау, иногда называют параллельным образованием ниш, поскольку мы пытаемся найти несколько решений одновременно. Но, как уже отмечалось, на практике у нее есть несколько недостатков. С другой стороны, метод последовательного образования ниш ищет по одному решению за раз. Для этого мы применяем генетический алгоритм, как обычно, и находим лучшее решение. Затем обновляем функцию приспособленности, так что область, в которой решения уже найдены, штрафуется, т. е. побуждаем алгоритм исследовать другие области пространства задачи. Это можно повторять несколько раз, пока алгоритм не перестанет находить новые решения.

Интересно, что штрафование областей вокруг уже найденных решений можно реализовать, налагая ограничения на пространство поиска, и, коль скоро мы уже умеем применять ограничения к функции, то можем воспользоваться этими знаниями для реализации последовательного образования ниш.

Для нахождения второго минимума функции Симионеску мы написали программу, которая находится по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter06/05-optimize-simionescu-second.py.

Она отличается от предыдущей только в двух местах.

1. Мы добавили константу, которая задает пороговое расстояние до ранее найденных решений – если новое решение отстоит от любого из уже найденных меньше, чем на это расстояние, то оно штрафуется.

   ```
   DISTANCE_THRESHOLD = 0.1
   ```

2. И мы добавили второе ограничение в функцию `feasible()`, написав условное предложение с несколькими ветвями. Новое ограничение применяется к входным значениям, отстоящим от уже найденного решения ($x = 0.848$, $y = -0.848$) меньше, чем на величину порога:

```
def feasible(individual):
    x = individual[0]
    y = individual[1]
    if x**2 + y**2 > (1 + 0.2 * math.cos(8.0 * math.atan2(x, y)))**2:
        return False
    elif (x - 0.848)**2 + (y + 0.848)**2 < DISTANCE_THRESHOLD**2:
        return False
    else:
        return True
```

Результаты работы этой программы показывают, что мы действительно нашли второй минимум:

```
-- Лучший индивидуум = [-0.8473430282562487, 0.8496942440090975]
-- Лучшая приспособленность = -0.07199824938105727
```

Рекомендуем добавить вторую точку минимума в качестве еще одного ограничения в функцию `feasible()` и убедиться, что программа не находит других минимумов в пространстве поиска.

Резюме

В этой главе мы познакомились с задачами оптимизации в непрерывном пространстве поиска и тем, как их можно представить и решить с помощью генетических алгоритмов, а конкретно с применением каркаса DEAP. Затем мы изучили несколько практических примеров задач непрерывной оптимизации – функцию Eggholder, функцию Химмельблау и функцию Симионеску – и их решения на Python. Мы также рассмотрели подходы к нахождению нескольких решений и учету ограничений.

В следующих главах мы продемонстрируем применение различных изученных приемов к решению задач машинного обучения и искусственного интеллекта. В первой из них мы дадим краткий обзор обучения с учителем, а затем покажем, как генетические алгоритмы могут улучшить результат моделей обучения, отбирая наиболее релевантные части имеющегося набора данных.

Для дальнейшего чтения

За дополнительной информацией обращайтесь к следующим источникам:

- математическая оптимизация, нахождение минимумов функций: http://scipylectures.org/advanced/mathematical_optimization/;
- оптимизация тестовых функций и наборов данных: https://www.sfu.ca/~ssurjano/optimization.html;
- введение в условную оптимизацию: https://web.stanford.edu/group/sisl/k12/optimization/MO-unit3-pdfs/3.1introandgraphical.pdf;
- обработка ограничений в DEAP: https://deap.readthedocs.io/en/master/tutorials/advanced/constraints.html.

ЧАСТЬ III

ПРИЛОЖЕНИЯ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ В ИСКУССТВЕННОМ ИНТЕЛЛЕКТЕ

В этой части мы будем использовать генетические алгоритмы для улучшения результатов, полученных различными алгоритмами машинного обучения

Глава 7

Дополнение моделей машинного обучения методами выделения признаков

В этой главе описывается применение генетических алгоритмов для повышения качества моделей машинного обучения с учителем путем выбора наилучшего подмножества признаков из предоставленных входных данных. Мы начнем с краткого введения в машинное обучение, а затем опишем два основных вида задач машинного обучения с учителем: регрессию и классификацию. После этого обсудим потенциальные преимущества выделения признаков с точки зрения качества модели. Далее продемонстрируем использование генетических алгоритмов для нахождения истинных признаков, сгенерированных задачей регрессии *тест Фридмана-1*. И наконец, мы воспользуемся набором данных Zoo, чтобы создать модель классификации и повысить ее точность – опять-таки посредством применения генетических алгоритмов для выделения наилучших признаков.

В этой главе рассматриваются следующие вопросы:

- основные понятия машинного обучения с учителем, а также задачи регрессии и классификации;
- положительное влияние выделения признаков на качество моделей обучения с учителем;
- повышение качества модели регрессии для задачи тест Фридмана-1 с помощью выделения признаков, произведенного генетическим алгоритмом, написанным с использованием каркаса DEAP;
- повышение качества модели классификации для набора данных Zoo с помощью выделения признаков, произведенного генетическим алгоритмом.

Глава начинается кратким введением в машинное обучение с учителем. Если вы профессионально занимаетесь анализом и обработкой данных, можете пропустить вступительные разделы.

Технические требования

В этой главе нам понадобится Python 3 и следующие библиотеки:

- `deap`;
- `numpy`;
- `pandas`;
- `matplotlib`;
- `seaborn`;
- `sklearn` – описывается здесь же.

Кроме того, мы будем использовать набор данных *Zoo* (https://archive.ics.uci.edu/ml/datasets/zoo).

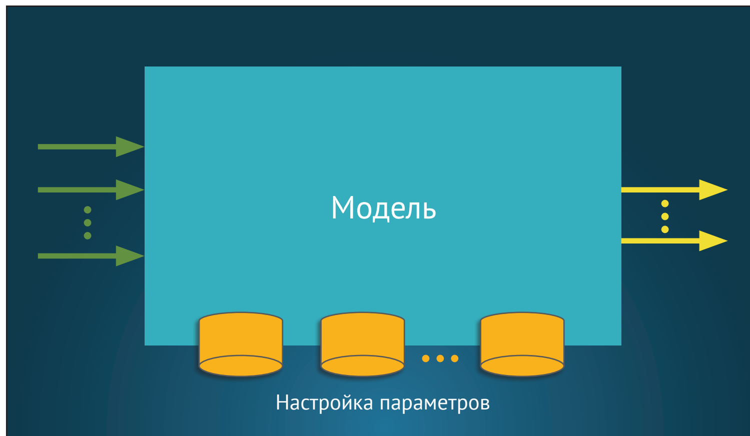
Код приведенных в этой главе программ можно найти в репозитории книги на GitHub по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter07. Чтобы увидеть код в действии, посмотрите видео по адресу http://bit.ly/37HCKyr.

Машинное обучение с учителем

Термин **машинное обучение** обычно относится к компьютерной программе, которая получает данные на входе и порождает данные на выходе. Наша цель – обучить эту программу, называемую также **моделью**, порождать правильные выходные данные по имеющимся входным, не программируя это соответствие явно.

В процессе обучения модель учится строить отображение между входами и выходами, корректируя свои внутренние параметры. Один из распространенных способов обучить модель – подать ей на вход данные, для которых известен правильный выход. Для каждого элемента входных данных мы сообщаем модели, каким должен быть выход, поэтому она может настроить себя так, чтобы порождать нужные выходы для всех входов. Вот этот процесс настройки и является сутью обучения.

С годами было разработано много типов моделей машинного обучения. У каждой модели свои внутренние параметры, которые влияют на отображение между входами и выходами, и значения этих параметров можно настраивать, как показано на рисунке ниже.



Настройка параметров модели машинного обучения

Например, модель, реализованная как решающее дерево, могла бы содержать несколько предложений IF-THEN и формулироваться следующим образом:

IF <входное значение> МЕНЬШЕ. ЧЕМ <некоторое пороговое значение>
THEN <перейти к некоторой ветви>

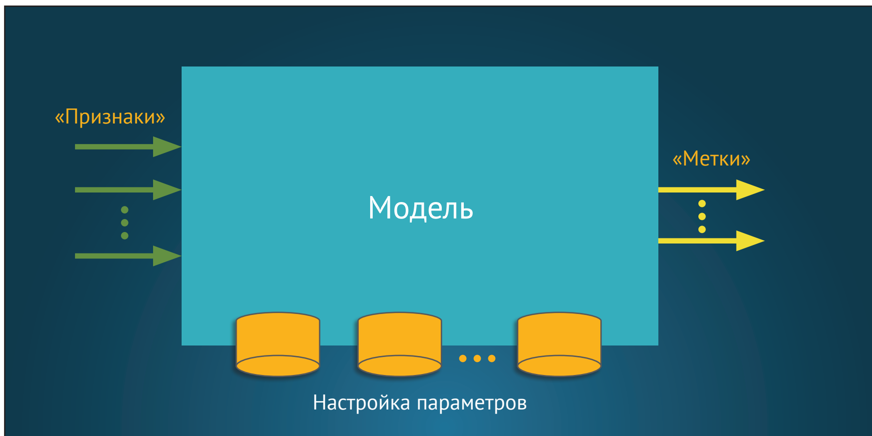
В этом случае пороговое значение и идентификатор целевой ветви – параметры, настраиваемые в процессе обучения.

Для настройки внутренних параметров с каждым типом моделей ассоциирован **алгоритм обучения**, который перебирает входные и выходные данные и стремится установить как можно более точное соответствие между входами и выходами. Для этого типичный алгоритм обучения вычисляет разность (ошибку) между фактическим и желаемым выходом, а затем пытается минимизировать эту ошибку, корректируя внутренние параметры модели.

Два основных типа машинного обучения с учителем – **классификация** и **регрессия** – будут описаны в следующих подразделах.

Классификация

В задаче классификации модель должна определить, к какой категории принадлежит вход. Каждая категория представлена одним выходом (**меткой**), а входные данные называются **признаками**.

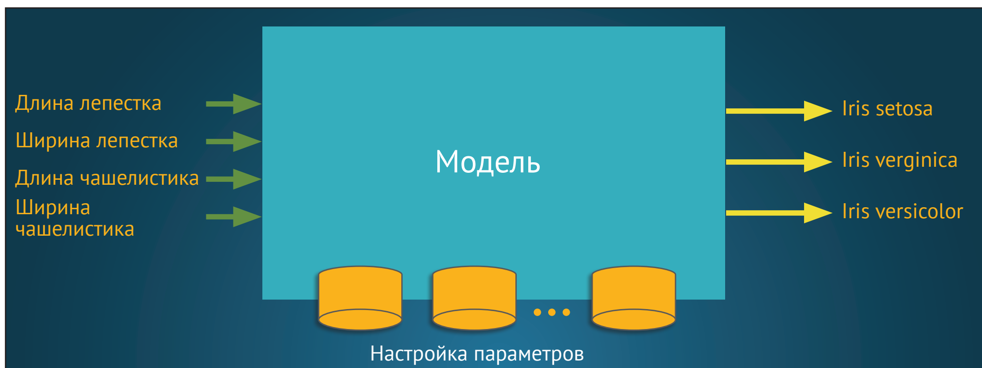


Модель классификации в машинном обучении

Например, в хорошо известном наборе данных об ирисах Iris (https://archive.ics.uci.edu/ml/datasets/Iris) признаков четыре: длина лепестка, ширина лепестка, длина чашелистика, ширина чашелистика. Эти характеристики были измерены для реальных цветков ириса.

На выходе возможна одна из трех меток, описывающих сорт ирисов: **Iris setosa**, **Iris virginica** или **Iris versicolor**.

Если на вход подаются данные, представляющие измерения некоторого конкретного цветка, то мы ожидаем, что для правильной метки будет выведено значение **high**, а для двух других – **low**:

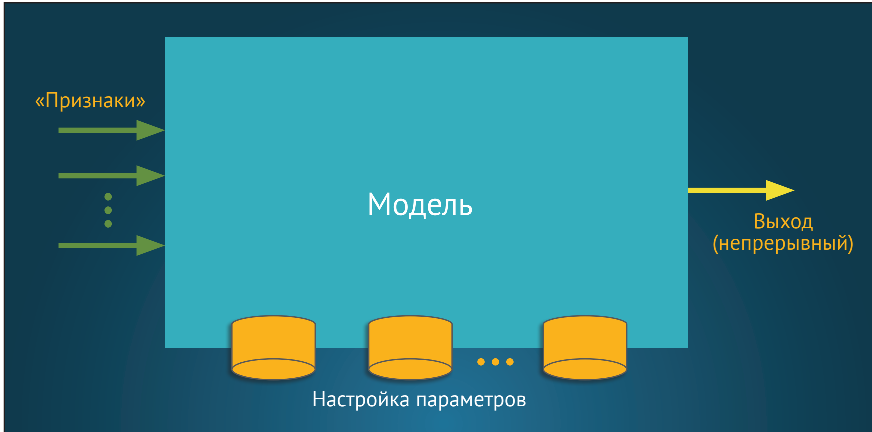


Классификатор ирисов

У задач классификации много практических применений, например одобрение банковских займов и максимального кредита по банковской карте, обнаружение почтового спама, распознавание рукописных цифр и лиц. Ниже в этой главе мы продемонстрируем классификацию животных с помощью набора данных Zoo.

Регрессия

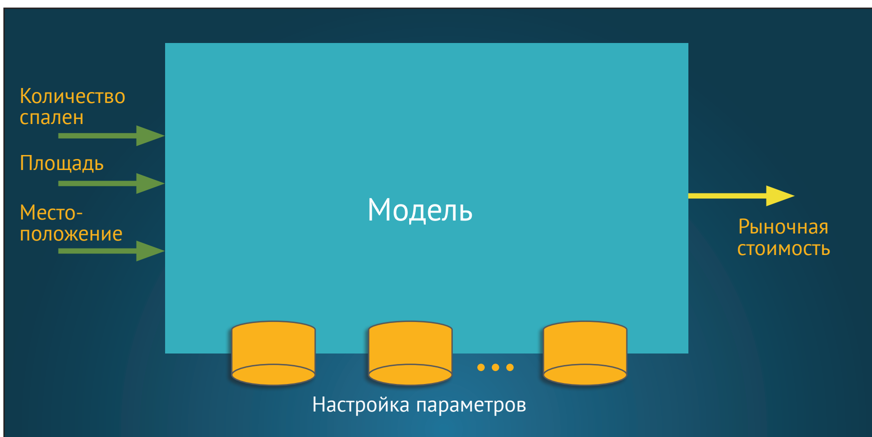
В отличие от классификации, модель регрессии отображает входные значения на единственное выходное значение с непрерывной областью значений.



Модель регрессии в машинном обучении

Ожидается, что модель будет предсказывать правильное выходное значение для поданных на вход данных.

Практические приложения регрессии включают предсказание стоимости ценных бумаг, качества вина или рыночной цены дома, как показано на рисунке ниже.



Регрессионная модель цены дома

Здесь на вход подаются признаки, содержащие информацию о доме, а выходом является предсказанная стоимость дома.

Существует много моделей классификации и регрессии – некоторые из них описаны в следующем подразделе.

Алгоритмы обучения с учителем

Как уже было сказано, у любой модели обучения с учителем имеется набор внутренних настраиваемых параметров и алгоритм, настраивающий их в попытке достичь желаемого результата.

Перечислим некоторые из распространенных моделей (алгоритмов) машинного обучения.

- **Решающие деревья:** семейство алгоритмов, в которых используется древовидный граф, вершины (узлы) которого представляют решения, а ветви – их последствия.
- **Случайные леса:** алгоритмы, которые создают много решающих деревьев на этапе обучения и используют их комбинацию для формирования выхода.
- **Метод опорных векторов:** входные данные рассматриваются как точки в пространстве, а алгоритм стремится разделить их плоскостью, так чтобы зазор между точками, принадлежащими разным категориям, был как можно больше.
- **Искусственные нейронные сети:** модели, состоящие из большого числа простых блоков, называемых нейронами, которые могут быть связаны между собой различными способами. С каждой связью ассоциируется вес, управляющий уровнем сигнала, передаваемого от одного нейрона другому.

Существуют методы, позволяющие повысить качество таких моделей. Один из них – выделение *признаков* – обсуждается в следующем разделе.

Выделение признаков в обучении с учителем

В предыдущем разделе мы видели, что модель обучения с учителем получает множество входов, называемых **признаками**, и отображает их на множество выходов. Предполагается, что информация, содержащаяся в признаках, полезна для определения значений выходов. На первый взгляд кажется, что чем больше информации подается на вход, тем выше шансы на правильное предсказание выходов. Но во многих случаях ситуация прямо противоположна; если некоторые признаки нерелевантны или избыточны, то верность моделей может снизиться (иногда значительно).

Выделение признаков – это процесс отбора самых важных и полезных признаков из общего их множества. Помимо повышения верности модели, успешное выделение признаков дает следующие преимущества:

- уменьшается время обучения модели;
- обученная модель проще, ее легче интерпретировать;

- обученная модель лучше обобщается, т. е. лучше работает на новых данных, которые не предъявлялись в процессе обучения.

Генетические алгоритмы естественно рассматривать при решении задачи о выделении признаков. В следующем разделе мы продемонстрируем их применение на примере искусственно сгенерированного набора данных.

Выделение признаков для задачи регрессии Фридмана-1

В задаче регрессии Фридмана-1, придуманной Фридманом и Брейманом, единственный выход y является следующей функцией пяти входных значений $x_0, \ldots, x_4$ и случайного шума:

$$y(x_0, x_1, x_2, x_3, x_4) = 10 \cdot \sin(\pi \cdot x_0 \cdot x_1) + 20(x_2 - 0.5)^2 + 10x_3 + 5x_4 + noise \cdot N(0,1).$$

Переменные $x_0, \ldots, x_4$ – независимые случайные величины, равномерно распределенные в интервале [0, 1]. Последнее слагаемое в формуле – случайно сгенерированный шум. Шум имеет нормальное распределение и умножен на константу *noise*, определяющую его уровень.

Библиотека scikit-learn (`sklearn`), написанная на Python, предоставляет функцию `make_friedman1()`, которая умеет генерировать набор данных, содержащий любое количество примеров. Каждый пример состоит из случайно сгенерированных значений $x_0, \ldots, x_4$ и соответствующего им значения y. Но вот что интересно – мы можем попросить функцию добавить произвольное число нерелевантных входных переменных к пяти исходным, задав параметр `n_features` больше 5. Так, если положить `n_features = 15`, то будет сгенерирован набор данных, содержащий пять исходных входных переменных (признаков), по которым вычислялось значение y, и еще 10 дополнительных признаков, вообще никак не связанных с выходом. Эту возможность можно использовать, например, для тестирования устойчивости различных моделей регрессии к шуму и присутствию нерелевантных признаков в наборе данных.

Мы можем воспользоваться данной функцией для проверки эффективности генетических алгоритмов как механизма выделения признаков. В нашем тесте функция `make_friedman1()` будет создавать набор данных с 15 признаками, а генетический алгоритм – искать подмножество признаков, при котором качество оказывается наилучшим. Ожидается, что генетический алгоритм отберет первые пять признаков и отбросит остальные в предположении, что верность модели действительно лучше, когда на вход подаются только релевантные признаки. Функция приспособленности генетического алгоритма будет использовать модель регрессии, которая для каждого потенциального решения – подмножества набора признаков – обучается на наборе данных, содержащем только выделенные признаки.

Как обычно, начнем с выбора подходящего представления решения.

Представление решения

Цель нашего алгоритма – найти подмножество признаков, при котором получается наилучшее качество модели. Поэтому в решении должно быть указано, какие признаки отобраны, а какие отброшены. Проще всего представлять индивидуума списком двоичных значений. Каждое значение соответствует одному признаку. Если значение равно 1, то соответствующий признак отобран, если 0 – то нет. Это очень напоминает подход, принятый в задаче о рюкзаке 0-1, описанной в главе 4.

Как мы увидим в следующем подразделе, присутствие 0 в решении означает выбрасывание соответствующего столбца признаков из набора данных.

Представление решения на Python

Для инкапсуляции задачи Фридмана-1 о выделении признаков мы создали Python-класс `Friedman1Test`, который находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/friedman.py.

Ниже описаны основные части этого класса.

1. Метод класса `__init__()` создает набор данных:

```
self.X, self.y = datasets.make_friedman1(n_samples=self.numSamples,
                                         n_features=self.numFeatures,
                                         noise=self.NOISE,
                                         random_state=self.randomSeed)
```

2. Затем он разбивает данные на два поднабора – обучающий и тестовый – с помощью метода `model_selection.train_test_split()` из библиотеки scikit-learn:

```
self.X_train, self.X_validation, self.y_train,
self.y_validation = \
    model_selection.train_test_split(self.X, self.y,
test_size=self.VALIDATION_SIZE, random_state=self.randomSeed)
```

   Разделение данных на обучающий и тестовый наборы позволяет обучить модель регрессии на обучающем наборе, в котором имеется правильная метка, а затем протестировать на отдельном тестовом наборе, не предоставляя модели правильных меток, а сравнивая с ними сделанные предсказания. Таким образом, можно проверить, действительно ли модель способна к обобщению, а не просто запомнила обучающие данные.

3. Затем мы создаем модель регрессии типа Gradient Boosting Regressor (регрессия с градиентным усилением – GBR). Эта модель создает ансамбль (или агрегат) решающих деревьев на этапе обучения:

```
self.regressor =
GradientBoostingRegressor(random_state=self.randomSeed)
```

Заметим, что конструктору передается начальное значение генератора случайных чисел, которое будет использоваться моделью. Тем самым мы обеспечиваем воспроизводимость результатов.

4. Метод класса `getMSE()` определяет качество нашей модели регрессии для набора выделенных признаков. Он принимает список двоичных значений – 1 означает, что соответствующий признак отобран, а 0 – что отброшен. Затем метод удаляет те столбцы обучающего и тестового наборов, которые соответствуют отброшенным признакам.

```
zeroIndices = [i for i, n in enumerate(zeroOneList) if n == 0]
currentX_train = np.delete(self.X_train, zeroIndices, 1)
currentX_validation = np.delete(self.X_validation, zeroIndices, 1)
```

5. После этого модифицированный обучающий набор, содержащий только выделенные признаки, используется для обучения модели регрессии, а модифицированный тестовый набор – для оценки ее предсказаний:

```
self.regressor.fit(currentX_train, self.y_train)
prediction = self.regressor.predict(currentX_validation)
return mean_squared_error(self.y_validation, prediction)
```

Для оценки модели регрессии будем использовать **среднеквадратическую ошибку** (**СКО**), т. е. среднее арифметическое разностей квадратов между фактическими и предсказанными моделью значениями. Чем меньше эта величина, тем лучше модель регрессии.

6. Метод `main()` создает экземпляр класса `Friedman1Test` с 15 признаками. Затем он в цикле вызывает метод `getMSE()`, чтобы оценить качество модели на первых *n* признаках, где *n* увеличивается от 1 до 15:

```
for n in range(1, len(test) + 1):
    nFirstFeatures = [1] * n + [0] * (len(test) - n)
    score = test.getMSE(nFirstFeatures)
```

Выполнение программы показывает, что по мере добавления первых пяти признаков качество повышается. Но каждый последующий признак ухудшает качество модели:

```
1 первых признаков: оценка = 47.553993
2 первых признаков: оценка = 26.121143
3 первых признаков: оценка = 18.509415
4 первых признаков: оценка = 7.322589
5 первых признаков: оценка = 6.702669
6 первых признаков: оценка = 7.677197
7 первых признаков: оценка = 11.614536
8 первых признаков: оценка = 11.294010
9 первых признаков: оценка = 10.858028
10 первых признаков: оценка = 11.602919
11 первых признаков: оценка = 15.017591
12 первых признаков: оценка = 14.258221
13 первых признаков: оценка = 15.274851
```

```
14 первых признаков: оценка = 15.726690
15 первых признаков: оценка = 17.187479
```

Это наглядно видно на графике, показывающем, что минимум СКО достигается, когда берутся первые пять признаков.

График среднеквадратической ошибки для теста Фридмана-1

В следующем разделе мы выясним, сможет ли генетический алгоритм выделить эти первые пять признаков.

Решение с помощью генетического алгоритма

Чтобы выделить лучший набор признаков в нашем тесте регрессии с помощью генетического алгоритма, мы написали на Python программу, которая находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/01-solve-friedman.py.

Напомним, что мы представляем хромосому списком нулей и единиц, в котором 1 означает, что список отобран, а 0 – что отброшен. С точки зрения генетического алгоритма, это делает нашу задачу похожей на задачу OneMax или задачу о рюкзаке 0-1, которые мы уже решали. Разница в том, что функция приспособленности теперь возвращает СКО модели регрессии, вычисляемую в классе `Friedman1Test`.

Ниже описаны основные шаги нашего решения.

1. Сначала создаем экземпляр класса `Friedman1Test` с нужными параметрами:

```
friedman = friedman.Friedman1Test(NUM_OF_FEATURES,
NUM_OF_SAMPLES, RANDOM_SEED)
```

2. Поскольку мы стремимся минимизировать СКО модели регрессии, определим единственную цель – минимизирующую стратегию приспособления:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. Так как решение представлено списком нулей и единиц, воспользуемся следующими определениями из инструментария для создания начальной популяции:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.zeroOrOne, len(friedman))
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. Затем просим генетический алгоритм использовать метод `getMSE()` объекта класса `Friedman1Test` для вычисления приспособленности:

```
def friedmanTestScore(individual):
    return friedman.getMSE(individual), # вернуть кортеж

toolbox.register("evaluate", friedmanTestScore)
```

5. Применяем турнирный отбор размера 2 и операторы скрещивания и мутации, специализированные для двоичного списка:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/len(friedman))
```

6. Как и раньше, используем элитистский подход, т. е. без изменения копируем лучших на данный момент индивидуумов из зала славы в следующее поколение:

```
population, logbook = elitism.eaSimpleWithElitism(population, toolbox,
                                                  cxpb=P_CROSSOVER,
                                                  mutpb=P_MUTATION,
                                                  ngen=MAX_GENERATIONS,
                                                  stats=stats,
                                                  halloffame=hof,
                                                  verbose=True)
```

После прогона алгоритма для 30 поколений с размером популяции 30 получается такой результат:

```
-- Лучший индивидуум = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
-- Лучшая приспособленность = 6.702668910463287
```

Это показывает, что для обеспечения лучшей СКО (приблизительно 6.7) были отобраны первые пять признаков. Заметим, что генетический алго-

ритм не делал никаких предположений о множестве исследуемых признаков, т. е. не знал, что ищет именно *первые n* признаков. Он просто искал наилучшее подмножество признаков.

В следующем разделе мы перейдем от искусственно сгенерированных данных к реальному набору данных и воспользуемся генетическим алгоритмом, чтобы выделить лучшие признаки для задачи классификации.

Выделение признаков для классификации набора данных Zoo

В репозитории машинного обучения Калифорнийского университета в Ирвайне (https://archive.ics.uci.edu/ml/index.php) хранится 350 наборов данных, доступных научно-исследовательскому сообществу. Эти наборы можно использовать для экспериментов с различными моделями и алгоритмами. Типичный набор, содержащий ряд признаков (входов) и желаемый выход, представлен в форме столбцов с описанием назначения каждого.

В этом разделе мы будем работать с набором данных Zoo (https://archive.ics.uci.edu/ml/datasets/zoo), в котором описано 101 животное с помощью следующих 18 признаков:

№	Признак	Тип данных
1	название животного	уникальное
2	шерсть	Boolean
3	перья	Boolean
4	яйцекладущее	Boolean
5	млекопитающее	Boolean
6	летающее	Boolean
7	водное	Boolean
8	хищник	Boolean
9	зубастое	Boolean
10	наличие позвоночника	Boolean
11	дышащее	Boolean
12	ядовитое	Boolean
13	плавники	Boolean
14	количество ног	Numeric (принимает значения из множества [0,2,4,5,6,8])
15	хвост	Boolean
16	домашнее	Boolean
17	размером с кошку	Boolean
18	тип	Numeric (целое в диапазоне [1,7])

Большинство признаков имеют тип Boolean (принимают значение 1 или 0), указывающий на наличие или отсутствие некоторого атрибута, например шерсти, плавников и т. д. Первый признак, **название животного**, чисто информационный, он не участвует в процессе обучения.

Этот набор данных используется для тестирования алгоритмов классификации, которые должны отобразить входные признаки в две или более категорий (меток). Последний признак, **тип**, как раз и представляет категорию и используется в качестве выхода. Всего в этом наборе семь категорий. Например, тип 5 соответствует категории, в которую входят лягушка, тритон и жаба. Итак, модель классификации, обучаемая на этом наборе, будет использовать признаки со 2 по 17 для предсказания признака 18 (тип животного).

Как и раньше, мы хотим использовать генетический алгоритм для выделения признаков, дающих лучшее предсказание. Начнем с создания Python-класса, представляющего классификатор, обученный на этом наборе данных.

Представление задачи на Python

Чтобы инкапсулировать процесс выделения признаков для классификации набора данных Zoo, мы создали Python-класс `Zoo`. Он находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/zoo.py.

Ниже описаны основные части этого класса.

1. Метод класса `__init__()` загружает набор данных Zoo из интернета и пропускает первый признак, название животного:

```
self.data = read_csv(self.DATASET_URL, header=None,
usecols=range(1, 18))
```

2. Затем он разделяет данные на входные признаки (первые 16 из оставшихся столбцов) и выходную категорию (последний столбец):

```
self.X = self.data.iloc[:, 0:16]
self.y = self.data.iloc[:, 16]
```

3. Вместо того чтобы разбивать данные на обучающий и тестовый наборы, как в предыдущем разделе, мы решили воспользоваться k-групповым перекрестным контролем. Это означает, что данные разбиваются на k равных частей и модель прогоняется k раз – в каждом прогоне $k - 1$ частей используются для обучения, а оставшаяся – для тестирования. На Python это легко сделать с помощью метода `model_selection.KFold()` из библиотеки scikit-learn:

```
self.kfold = model_selection.KFold(n_splits=self.NUM_FOLDS,
random_state=self.randomSeed)
```

4. Далее мы создаем модель классификации на основе решающего дерева. В классификаторе такого типа на этапе обучения строится древовидная структура, которая позволяет разбивать набор данных на меньшие части и в конечном итоге выдавать предсказание:

```
self.classifier = DecisionTreeClassifier(random_state=self.randomSeed)
```

Заметим, что конструктору передается начальное значение генератора случайных чисел, которое будет использоваться классификатором. Тем самым мы обеспечиваем воспроизводимость результатов.

5. Метод `getMeanAccuracy()` вычисляет качество классификатора для множества отобранных признаков. Как и метод `getMSE()` класса `Friedman1Test`, он принимает список двоичных значений, в котором 1 соответствует отобранному признаку, а 0 – отброшенному. Затем метод удаляет те столбцы обучающего и тестового наборов, которые соответствуют отброшенным признакам.

```
zeroIndices = [i for i, n in enumerate(zeroOneList) if n == 0]
currentX = self.X.drop(self.X.columns[zeroIndices], axis=1)
```

6. Модифицированный набор данных, содержащий только отобранные признаки, используется для *k*-группового перекрестного контроля и оценки качества классификатора на отдельных группах. В нашем классе значение *k* принято равным 5, поэтому каждый раз выполняется пять оцениваний:

```
cv_results = model_selection.cross_val_score(self.classifier,
currentX, self.y, cv=self.kfold, scoring='accuracy')
return cv_results.mean()
```

   Для оценки классификатора применяется метрика, называемая **верностью**, – доля правильно классифицированных примеров. Например, верность 0.85 означает, что 85 % всех примеров были классифицированы правильно. Поскольку мы обучаем и оцениваем классификатор *k* раз, то используем верность, усредненную по этим *k* вычислениям.

7. Метод `main()` создает экземпляр класса Zoo и оценивает классификатор, обученный на всех 16 признаках, присутствующих в представлении, содержащем все единицы:

```
allOnes = [1] * len(zoo)
print("-- Выделены все признаки: ", allOnes, ", accuracy = ",
zoo.getMeanAccuracy(allOnes))
```

Результат выполнения метода `main` показывает, что при обучении классификатора методом 5-группового перекрестного контроля на всех 16 признаках достигается верность 91 %:

```
-- Выделены все признаки: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
accuracy = 0.9099999999999999
```

В следующем подразделе мы попробуем улучшить верность классификатора, выделив только часть признаков. И, как вы догадались, воспользуемся для этого генетическим алгоритмом.

Решение с помощью генетического алгоритма

Для выделения наилучшего набора признаков в задаче классификации набора данных Zoo мы написали на Python программу, которая находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/02-solve-zoo.py.

Как было описано в предыдущем разделе, хромосома будет представлена списком нулей и единиц, соответствующих отброшенным и отобранным признакам.

Ниже описаны основные шаги программы.

1. Сначала создаем экземпляр класса Zoo и передаем конструктору начальное значение генератора случайных чисел, чтобы обеспечить воспроизводимость результатов:

   ```
   zoo = zoo.Zoo(RANDOM_SEED)
   ```

2. Поскольку мы стремимся минимизировать верность модели классификации, определим единственную цель – максимизирующую стратегию приспособления:

   ```
   creator.create("FitnessMax", base.Fitness, weights=(1.0,))
   ```

3. Как и в предыдущем разделе, воспользуемся следующими определениями из инструментария, чтобы создать начальную популяцию индивидуумов, описываемых списками нулей и единиц:

   ```
   toolbox.register("zeroOrOne", random.randint, 0, 1)
   toolbox.register("individualCreator", tools.initRepeat,
   creator.Individual, toolbox.zeroOrOne, len(zoo))
   toolbox.register("populationCreator", tools.initRepeat, list,
   toolbox.individualCreator)
   ```

4. Далее мы вызываем метод `getMeanAccuracy()` объекта `Zoo` для вычисления приспособленности. Для этого пришлось внести два изменения:
   - мы запретили алгоритму не выбирать ни одного признака (создавать индивидуума, представленного только нулями), поскольку в таком случае классификатор возбудил бы исключение;
   - мы добавили небольшой штраф за каждый отобранный признак, чтобы побудить алгоритм выделять меньше признаков. Штраф совсем невелик (0.001), поэтому будет использоваться, только чтобы разрешить неоднозначность в случае, когда качество двух классификаторов одинаково – тогда предпочтение будет отдано тому, который обошелся меньшим числом признаков.

   ```
   def zooClassificationAccuracy(individual):
       numFeaturesUsed = sum(individual)
       if numFeaturesUsed == 0:
           return 0.0,
       else:
   ```

```
        accuracy = zoo.getMeanAccuracy(individual)
        return accuracy - FEATURE_PENALTY_FACTOR *
numFeaturesUsed, # вернуть кортеж

toolbox.register("evaluate", zooClassificationAccuracy)
```

5. Применяем турнирный отбор размера 2 и операторы скрещивания и мутации, специализированные для двоичного списка:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit,indpb=1.0/len(zoo))
```

6. Как и раньше, используем элитистский подход, т. е. без изменения копируем лучших на данный момент индивидуумов из зала славы в следующее поколение:

```
population, logbook = elitism.eaSimpleWithElitism(population,
toolbox, cxpb=P_CROSSOVER, mutpb=P_MUTATION,
ngen=MAX_GENERATIONS, stats=stats, halloffame=hof, verbose=True)
```

7. В конце прогона мы печатаем все содержимое зала славы, чтобы увидеть лучшие найденные алгоритмом результаты. Печатается как приспособленность, включающая штраф за количество признаков, так и фактическая верность:

```
print("- Лучшие решения:")
for i in range(HALL_OF_FAME_SIZE):
    print(i, ": ", hof.items[i], ", приспособленность = ",
hof.items[i].fitness.values[0],
          ", верность = ", zoo.getMeanAccuracy(hof.items[i]),
          ", признаков = ", sum(hof.items[i]))
```

После выполнения алгоритма для 50 поколений с популяцией размера 50 и залом славы размера 5 мы получили:

```
- Лучшие решения:
0 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0] , приспособленность = 0.964 ,
верность = 0.97 , признаков = 6
1 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1] , приспособленность = 0.963 ,
верность = 0.97 , признаков = 7
2 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0] , приспособленность = 0.963 ,
верность = 0.97 , признаков = 7
3 : [1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0] , приспособленность = 0.963 ,
верность = 0.97 , признаков = 7
4 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0] , приспособленность = 0.963 ,
верность = 0.97 , признаков = 7
```

Эти результаты показывают, что все пять лучших решений достигли верности 97 %, используя шесть или семь признаков из 16. Благодаря штрафу за количество признаков самым лучшим является решение с шестью признаками:

- перья;
- млекопитающее;
- летающее;
- наличие позвоночника;
- плавники;
- хвост.

Итак, выделив именно эти шесть признаков из 16, мы смогли не только уменьшить размерность задачи, но и повысить верность модели с 91 % до 97 %. Если вам кажется, что это не такое уж большое достижение, взгляните на него по-другому – как на уменьшение частоты ошибок с 9 % до 3 %, что считается весьма значительным улучшением качества классификации.

Резюме

В этой главе мы познакомились с машинным обучением и двумя основными типами встречающихся в нем задач: регрессией и классификацией. Затем мы узнали о потенциальном положительном влиянии выделения признаков на качество моделей для решения этих задач. Основным содержанием данной главы стали два примера применения генетических алгоритмов к повышению качества моделей за счет выделения признаков. В первом случае нам удалось выделить истинные признаки в тестовой задаче регрессии Фридмана-1, а во втором мы выделили самые полезные признаки в наборе данных Zoo для классификации.

В следующей главе мы рассмотрим еще один способ повышения качества моделей машинного обучения с учителем – настройку гиперпараметров.

Для дальнейшего чтения

За дополнительными сведениями по вопросам, рассмотренным в этой главе, рекомендуем обратиться к следующим источникам:

- Benjamin Johnston and Ishita Mathur «Applied Supervised Learning with Python»;
- Sinan Ozdemir and Divya Susarla «Feature Engineering Made Easy»;
- M. Dash and H. Liu «Feature selection for classification», 1997: https://doi.org/10.1016/S1088-467X(97)00008-5;
- репозиторий машинного обучения UCI: https://archive.ics.uci.edu/ml/index.php.

Глава 8

Настройка гиперпараметров моделей машинного обучения

В этой главе описывается, как с помощью генетических алгоритмов улучшить качество моделей машинного обучения с учителем за счет настройки их гиперпараметров. Мы начнем с краткого введения в настройку гиперпараметров, а затем опишем идею поиска на сетке. После описания набора данных о винах Wine и классификатора на основе адаптивного усиления (то и другое используется в этой главе) мы продемонстрируем настройку гиперпараметров как с помощью традиционного поиска на сетке, так и путем применения к поиску генетического алгоритма. Наконец, попробуем улучшить результаты, применив генетический алгоритм напрямую.

В этой главе мы:

- узнаем о том, что такое настройка гиперпараметров в машинном обучении;
- познакомимся с набором данных Wine и классификатором на основе адаптивного усиления;
- улучшим качество классификатора с помощью поиска гиперпараметров на сетке;
- улучшим качество классификатора с помощью поиска на сетке, управляемого генетическим алгоритмом;
- улучшим качество классификатора, непосредственно применив генетический алгоритм к настройке гиперпараметров.

Глава начинается кратким обзором роли гиперпараметров в машинном обучении. Если вы профессионально занимаетесь анализом и обработкой данных, можете пропустить этот вступительный раздел.

Технические требования

В этой главе нам понадобится Python 3 и следующие библиотеки:

- `deap`;
- `numpy`;
- `pandas`;
- `matplotlib`;
- `seaborn`;
- `sklearn`;
- `sklearn-deap` – описывается здесь же.

Кроме того, мы будем использовать набор данных Wine Калифорнийского университета в Ирвайне (https://archive.ics.uci.edu/ml/datasets/Wine).

Код приведенных в этой главе программ можно найти в репозитории книги на GitHub по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter08. Чтобы увидеть код в действии, посмотрите видео по адресу http://bit.ly/37Q45id.

Гиперпараметры в машинном обучении

В главе 7 мы описывали машинное обучение как процесс программной корректировки (или настройки) внутренних параметров модели, так чтобы она порождала желаемые выходы в ответ на подаваемые входные данные. Для этого с каждой моделью обучения с учителем ассоциируется алгоритм обучения, который итеративно корректирует свои внутренние параметры на этапе обучения.

Однако у большинства моделей есть еще один набор параметров, которые задаются **до** начала обучения. Они называются **гиперпараметрами** и влияют на сам способ обучения. На рисунке ниже показаны оба типа параметров.

Настройка гиперпараметров модели машинного обучения

Обычно у гиперпараметров имеются значения по умолчанию, которые применяются, если никакие другие не заданы явно. Например, заглянув в реализацию решающего дерева в библиотеке `sklearn` (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html), мы увидим несколько гиперпараметров вместе с их значениями по умолчанию. Некоторые из них приведены в таблице ниже.

Имя	Тип	Описание	Значение по умолчанию
`max_depth`	int	Максимальная глубина дерева	Нет
`splitter`	string	Стратегия разделения в каждом узле (`best` или `random`)	`best`
`min_samples_split`	int или float	Минимальное количество примеров, при котором производится разделение	2

Каждый из этих параметров влияет на процесс построения дерева на этапе обучения, а их совокупное воздействие на результат обучения и, следовательно, на качество модели может быть весьма существенно.

Поскольку выбор гиперпараметров может оказать заметный эффект на качество моделей машинного обучения, специалисты по обработке данных часто тратят много времени на поиск лучших комбинаций гиперпараметров. Этот процесс и называется **настройкой гиперпараметров**. В следующем подразделе описаны некоторые методы настройки гиперпараметров.

Настройка гиперпараметров

Широко распространенный способ отыскания хороших комбинаций гиперпараметров – **поиск на сетке**. При использовании этого метода мы выбираем подмножество значений каждого гиперпараметра, подлежащего настройке. Например, в случае решающего дерева можно выбрать подмножество {2, 5, 10} значений гиперпараметра `max_depth`, а для `splitter` оставить оба допустимых значения – {`"best"`, `"random"`}. Тогда нужно будет исследовать все шесть комбинаций этих значений. Для каждой комбинации классификатор обучается и оценивается по некоторому критерию, например верности. В итоге мы выбираем ту комбинацию, которая дала наилучшее качество.

Основной недостаток поиска на сетке – необходимость полного перебора всех комбинаций, что может потребовать очень много времени. Сократить время позволяет **случайный поиск**, когда выбираются и проверяются случайные комбинации гиперпараметров.

Нам интереснее вариант поиска на сетке, при котором для нахождения наилучших комбинаций применяется генетический алгоритм. Он открывает возможность отыскать лучшую комбинацию за время меньшее, чем в случае исчерпывающего поиска на сетке.

Поиск на сетке и случайный поиск поддерживаются библиотекой `sklearn`, но для применения генетического алгоритма нужна библиотека `sklearn-deap`, построенная на базе каркаса DEAP. Для ее установки выполните команду

```
pip install sklearn-deap
```

В следующих разделах мы протестируем и сравним обе версии поиска на сетке – полным перебором и с применением генетического алгоритма. Но сначала познакомимся с набором данных Wine, на котором будем экспериментировать.

Набор данных Wine

Широко известен набор данных о винах Wine из репозитория машинного обучения Калифорнийского университета в Ирвайне. Он находится по адресу https://archive.ics.uci.edu/ml/datasets/Wine и содержит результаты химического анализа 178 разных вин, произведенных в одном и том же регионе Италии. Каждому вину присвоена одна из трех категорий.

Химический анализ производится по 13 показателям, представляющим следующие характеристики вина:

- процентное содержание алкоголя;
- содержание яблочной кислоты;
- содержание золы;
- содержание щелочи;
- содержание магния;
- общее количество фенолов;
- флаваноидные фенолы;
- нефлаваноидные фенолы;
- проантоцианиды;
- интенсивность цвета;
- оттенок вина;
- разбавленность вина;
- сорт вина.

В столбцах 2–4 набора данных содержатся результаты измерений, а результат классификации – сам тип вина (1, 2 или 3) – находится в первом столбце.

Теперь рассмотрим классификатор, который мы собираемся использовать.

Классификатор на основе адаптивного усиления

Алгоритм адаптивного усиления, или **AdaBoost**, – эффективная модель машинного обучения, которая объединяет результаты нескольких простых алгоритмов обучения (слабых обучаемых), вычисляя их взвешенную сумму. AdaBoost складывает экземпляры слабых обучаемых в процессе обучения, корректируя их с целью улучшить результаты для входных примеров, которые до того были классифицированы неверно.

В реализации этой модели в библиотеке `sklearn`, AdaboostClassifier (https://scikitlearn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html), имеется несколько гиперпараметров, некоторые приведены в следующей таблице.

Имя	Тип	Описание	Значение по умолчанию
n_estimators	int	Максимальное число оценивателей	50
learning_rate	float	Позволяет уменьшить вклад каждого классификатора	1
algorithm	{'SAMME', 'SAMME.R'}	'SAMME' – используется вещественный алгоритм усиления; 'SAMME.R' – используется дискретный алгоритм усиления	2

Интересно, что типы всех трех гиперпараметров различны – `int`, `float` и перечисление. Ниже мы узнаем, как методы настройки обращаются с этими типами. В следующем разделе описываются две формы поиска на сетке.

Настройка гиперпараметров с помощью генетического поиска на сетке

Чтобы инкапсулировать настройку гиперпараметров классификатора AdaBoost для набора данных Wine с помощью поиска на сетке – традиционного и управляемого генетическим алгоритмом, – мы создали Python-класс `HyperparameterTuningGrid`, который находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter08/01-hyperparameter-tuning-grid.py.

Ниже описаны основные части этого класса.

1. Метод класса `__init__()` инициализирует набор данных, классификатор AdaBoost, *k*-групповой перекрестный контроль и параметры сетки:

```
self.initWineDataset()
self.initClassifier()
self.initKfold()
self.initGridParams()
```

2. Метод `initGridParams()` инициализирует поиск на сетке, задавая множество значений трех гиперпараметров, упомянутых в предыдущем разделе:
   - для параметра `n_estimators` проверяется 10 значений, равномерно распределенных между 10 и 100;
   - для параметра `learning_rate` проверяется 10 значений, логарифмически равномерно распределенных между 0.1 (10^{-2}) и 1 (10^{0});
   - для параметра `algorithm` проверяются оба допустимых значения: `'SAMME'` и `'SAMME.R'`.

   Таким образом, нам предстоит проверить 200 (10×10×2) различных комбинаций параметров:

```
self.gridParams = {
    'n_estimators': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100],
```

```
    'learning_rate': np.logspace(-2, 0, num=10, base=10),
    'algorithm': ['SAMME', 'SAMME.R'],
}
```

3. Метод `getDefaultAccuracy()` оценивает верность классификатора с подразумеваемыми по умолчанию значениями гиперпараметров:

```
cv_results = model_selection.cross_val_score(self.classifier,
                                             self.X,
                                             self.y,
                                             cv=self.kfold,
                                             scoring='accuracy')
return cv_results.mean()
```

4. Метод `gridTest()` выполняет традиционный поиск на определенной ранее сетке значений гиперпараметров. Применяя *k*-групповой перекрестный контроль и вычисляя среднюю верность, он находит лучшую комбинацию значений:

```
gridSearch = GridSearchCV(estimator=self.classifier,
                          param_grid=self.gridParams,
                          cv=self.kfold,
                          scoring='accuracy')

gridSearch.fit(self.X, self.y)
```

5. Метод `geneticGridTest()` выполняет поиск на сетке, управляемый генетическим алгоритмом. Для этого используется метод `EvolutionaryAlgorithmSearchCV()` из библиотеки `sklearn-deap`, специально разработанный так, чтобы вызов был похож на традиционный поиск на сетке. Нужно только добавить несколько параметров генетического алгоритма: размер популяции, вероятность мутации, размер турнира и количество поколений:

```
gridSearch =
EvolutionaryAlgorithmSearchCV(estimator=self.classifier,
                              params=self.gridParams,
                              cv=self.kfold,
                              scoring='accuracy',
                              verbose=True,
                              population_size=20,
                              gene_mutation_prob=0.30,
                              tournament_size=2,
                              generations_number=5)
                              gridSearch.fit(self.X, self.y)
```

6. Наконец, метод `main()` начинает с того, что оценивает качество классификатора с подразумеваемыми по умолчанию значениями гиперпараметров. После этого он прогоняет сначала традиционный полный поиск на сетке, а затем генетический поиск на сетке, замеряя в каждом случае время.

Результаты выполнения метода main этого класса описаны в следующем подразделе.

Тестирование качества классификатора с параметрами по умолчанию

Результаты прогона показывают, что при подразумеваемых по умолчанию значениях n_estimators = 50, learning_rate = 1.0 и algorithm = 'SAMME.R' верность классификации равна приблизительно 65 %:

Значения гиперпараметров классификатора по умолчанию:

```
{'algorithm': 'SAMME.R', 'base_estimator': None, 'learning_rate': 1.0,
'n_estimators': 50, 'random_state': 42}
Оценка для значений по умолчанию = 0.6457142857142857
```

Эту верность хорошей не назовешь. Будем надеяться, что поиск на сетке позволит улучшить ее, отыскав более подходящую комбинацию гиперпараметров.

Результаты традиционного поиска на сетке

Далее запускается традиционный исчерпывающий поиск на сетке с перебором всех 200 возможных комбинаций. Выясняется, что лучшей является комбинация n_estimators = 70, learning_rate ≈ 0.359, algorithm = 'SAMME.R'.

При этом достигается верность классификации приблизительно 93 % – гораздо лучше, чем первоначальные 65 %. Время поиска составило около 32 секунд:

```
производится поиск на сетке...
лучшие параметры: {'algorithm': 'SAMME.R', 'learning_rate':
0.3593813663804626, 'n_estimators': 70}
лучшая оценка: 0.9325842696629213
затраченное время = 32.180874824523926
```

Далее у нас по плану генетический поиск на сетке. Удастся ли улучшить результаты? Посмотрим.

Результаты генетического поиска на сетке

Последняя часть прогона – генетический поиск на сетке с теми же параметрами. Распечатка начинается с загадочного сообщения:

```
выполняется генетический поиск на сетке...
обнаружены типы [1, 2, 1] и maxint [9, 9, 1]
```

Эти сообщения относятся к сетке, на которой производится поиск: список из 10 целых чисел (значений n_estimators), массив ndarray из 10 элементов (значений learning_rate) и список из двух строк (значений algorithm), а именно:

- `'типы [1, 2, 1]'` описывает типы значений: [список, ndarray, список];
- `'maxint [9, 9, 1]'` описывает размеры списков и массивов: [10, 10, 2].

Следующая строка говорит об общем количестве комбинаций параметров (10×10×2):

```
--- Эволюция на 200 возможных комбинациях ---
```

Остаток распечатки нам хорошо знаком, потому что используются те самые реализации генетических алгоритмов в DEAP, с которыми мы работали всю дорогу. Точнее, печатается статистика каждого поколения:

```
gen nevals avg      min      max      std
0   20     0.642135 0.117978 0.904494 0.304928
1   14     0.807865 0.123596 0.91573  0.20498
2   15     0.829775 0.123596 0.921348 0.172647
3   12     0.885393 0.679775 0.921348 0.0506055
4   13     0.903652 0.865169 0.926966 0.0176117
5   11     0.905618 0.797753 0.932584 0.027728
```

В конце работы печатается лучшая найденная комбинация, а также оценка и затраченное время:

```
Лучший индивидуум: {'n_estimators': 70, 'learning_rate':
0.3593813663804626, 'algorithm': 'SAMME.R'}
приспособленность: 0.9325842696629213
затраченное время = 10.997037649154663
```

Эти результаты показывают, что генетический поиск на сетке смог найти тот же лучший результат, что и исчерпывающий поиск, но за меньшее время – примерно 11 секунд.

Отметим, что это простой пример, в котором алгоритм работает очень быстро. На практике часто встречаются куда большие наборы данных, а также сложные модели и многомерные сетки гиперпараметров. В таких случаях исчерпывающий поиск на сетке может занять недопустимо много времени, тогда как генетический поиск способен дать хорошие результаты за приемлемое время.

Тем не менее любой поиск на сетке ограничен определенным нами подмножеством гиперпараметров. А что, если мы хотим произвести поиск вне сетки, не ограничиваясь заранее заданным подмножеством? Возможное решение описано в следующем разделе.

Прямой генетический подход к настройке гиперпараметров

Генетические алгоритмы можно использовать не только как эффективный способ поиска на сетке, но и для прямого поиска во всем пространстве параметров – так же, как мы делали во многих задачах, рассмотренных в этой

книге. Каждый гиперпараметр можно представить переменной, участвующей в поиске, а хромосомой тогда будет комбинация всех таких переменных.

Поскольку типы гиперпараметров могут быть различны, например float, int и перечисление, как в классификаторе AdaBoost, возможно, придется кодировать их по-разному, а в качестве генетических операций использовать какую-то комбинацию отдельных операторов, адаптированных к конкретным типам. Но можно и не «заморачиваться», а представить все параметры числами с плавающей точкой и тем самым упростить алгоритм, что мы и сделаем далее.

Представление гиперпараметров

В главе 6 мы использовали генетические алгоритмы для оптимизации функций вещественных параметров. Параметры были представлены списками чисел с плавающей точкой, например:

```
[1.23, 7.2134, -25.309]
```

Поэтому и генетические операторы были специализированы для работы с такими списками.

Чтобы адаптировать этот подход к настройке гиперпараметров, мы будем представлять каждый гиперпараметр числом с плавающей точкой независимо от его истинного типа. Нужно только придумать, как преобразовать параметр в число с плавающей точкой, а затем вернуться к исходному представлению. Мы поступим следующим образом:

- `n_estimators`, целое число, будет представлено числом типа float в некотором диапазоне, например [1, 100]. Для преобразования float в int мы воспользуемся функцией Python `round()`, которая округляет вещественное число до ближайшего целого;
- параметр `learning_rate` уже имеет тип float, так что никакого преобразования не нужно. Он находится в диапазоне [0.01, 1.0];
- параметр `algorithm` может принимать одно из двух значений: `'SAMME'` и `'SAMME.R'`. Мы будем представлять его числом типа float в диапазоне [0, 1]. Для преобразования обратно в целое округлим до ближайшего целого – 0 или 1, а затем заменим 0 на `'SAMME'`, а 1 – на `'SAMME.R'`.

Код этих преобразований находится в двух Python-файлах, которые описаны в следующих подразделах.

Оценка верности классификатора

Оценка верности классификатора инкапсулирована в Python-классе `HyperparameterTuningGenetic`, находящемся в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter08/hyperparameter_tuning_genetic_test.py.

Ниже описаны основные функции этого класса.

1. Метод класса convertParam() принимает список params, содержащий значения гиперпараметров типа float, и преобразует их в значения истинных типов, как было описано в предыдущем подразделе:

```
n_estimators = round(params[0])
learning_rate = params[1]
algorithm = ['SAMME', 'SAMME.R'][round(params[2])]
```

2. Метод getAccuracy() принимает список чисел типа float, представляющих значения гиперпараметров, вызывает метод convertParam() для преобразования их в истинные значения и инициализирует классификатор ADABooost этими значениями:

```
n_estimators, learning_rate, algorithm = self.convertParams(params)
self.classifier = AdaBoostClassifier(n_estimators=n_estimators,
                                     learning_rate=learning_rate,
                                     algorithm=algorithm)
```

3. Затем он вычисляет верность классификатора, используя *k*-групповой перекрестный контроль, как в наборе данных Wine:

```
cv_results = model_selection.cross_val_score(self.classifier,
                                             self.X,
                                             self.y,
                                             cv=self.kfold,
                                             scoring='accuracy')
return cv_results.mean()
```

Этот класс применяется в программе, реализующей генетический алгоритм настройки гиперпараметров, которая описана в следующем подразделе.

Настройка гиперпараметров с помощью генетического алгоритма

Генетический поиск лучших гиперпараметров реализован в программе, находящейся в файле https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter08/02-hyperparameter-tuning-genetic.py.

Ниже описаны основные части программы.

1. Задаем нижние и верхние границы всех чисел типа float, представляющих гиперпараметры: [1, 100] для n_estimators, [0.01, 1] для learning_rate и [0, 1] для algorithm:

```
# [n_estimators, learning_rate, algorithm]:
BOUNDS_LOW = [ 1, 0.01, 0]
BOUNDS_HIGH = [100, 1.00, 1]
```

2. Создаем экземпляр класса HyperparameterTuningGenetic, который позволит нам проверить различные комбинации гиперпараметров:

```
test = hyperparameter_tuning_genetic.HyperparameterTuningGenetic(RANDOM_SEED)
```

3. Поскольку мы стремимся максимизировать верность классификатора, определим единственную цель – максимизирующую стратегию приспособления:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

4. Вот теперь мы подошли к интересной части: т. к. решение представлено списком чисел типа float, принадлежащих различным диапазонам, мы в цикле обойдем все пары, состоящие из нижней и верхней границ, и для каждого гиперпараметра создадим в инструментарии свой оператор, который будет генерировать случайное число в соответствующем диапазоне:

```
for i in range(NUM_OF_PARAMS):
    # "hyperparameter_0", "hyperparameter_1", ...
    toolbox.register("hyperparameter_" + str(i),
                     random.uniform,
                     BOUNDS_LOW[i],
                     BOUNDS_HIGH[i])
```

5. Затем создаем кортеж `hyperparameters`, содержащий только что созданные генераторы случайных чисел для отдельных гиперпараметров:

```
hyperparameters = ()
for i in range(NUM_OF_PARAMS):
    hyperparameters = hyperparameters + \
(toolbox.__getattribute__("hyperparameter_" + str(i)),)
```

6. Теперь мы можем использовать этот кортеж в сочетании со встроенным в DEAP оператором `initCycle()`, чтобы создать новый оператор `individualCreator`, который инициализирует индивидуум комбинацией случайных значений гиперпараметров:

```
toolbox.register("individualCreator",
tools.initCycle,
creator.Individual,
hyperparameters,
n=1)
```

7. Далее мы просим генетический алгоритм использовать метод `getAccuracy()` экземпляра `HyperparameterTuningGenetic` для вычисления приспособленности. Напомним, что метод `getAccuracy()`, описанный в предыдущем подразделе, преобразует переданного ему индивидуума – список трех чисел типа float – в значения соответствующих гиперпараметров, обучает классификатор с этими значениями и вычисляет верность, пользуясь *k*-групповым перекрестным контролем:

```
def classificationAccuracy(individual):
    return test.getAccuracy(individual),

toolbox.register("evaluate", classificationAccuracy)
```

8. Осталось определить генетические операторы. В качестве оператора отбора используем турнир размера 2, а для скрещивания и мутации возьмем операторы, специализированные для хромосом в виде списков чисел с плавающей точкой, ограниченных предельными значениями гиперпараметров:

```
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0 / NUM_OF_PARAMS)
```

9. Как и раньше, будем использовать элитистский подход, т. е. без изменения копировать лучших на данный момент индивидуумов из зала славы в следующее поколение:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                  toolbox,
                                                  cxpb=P_CROSSOVER,
                                                  mutpb=P_MUTATION,
                                                  ngen=MAX_GENERATIONS,
                                                  stats=stats,
                                                  halloffame=hof,
                                                  verbose=True)
```

После прогона алгоритма для 5 поколений с размером популяции 20 получается такой результат:

```
gen nevals max      avg
0   20     0.92127  0.841024
1   14     0.943651 0.900603
2   13     0.943651 0.912841
3   14     0.943651 0.922476
4   15     0.949206 0.929754
5   13     0.949206 0.938563
- Лучшее решение:
params = 'n_estimators'= 69, 'learning_rate'=0.628, 'algorithm'=SAMME.R
Верность = 0.94921
```

Таким образом, лучшая найденная комбинация имеет вид `n_estimators = 69`, `learning_rate = 0.628`, `algorithm = 'SAMME.R'`. Достигнутая при таких значениях верность равна приблизительно 94.9 % – заметное улучшение по сравнению с верностью, достигнутой поиском на сетке. Интересно, что лучшие значе-

ния n_estimators и learning_rate оказались вне той сетки, на которой мы их искали.

Резюме

В этой главе мы занимались настройкой гиперпараметров в машинном обучении. Познакомившись с набором данных Wine и классификатором с адаптивным усилением AdaBoost, мы затем представили методы настройки гиперпараметров посредством исчерпывающего поиска на сетке и его генетического аналога. Мы сравнили оба метода и напоследок испытали подход на основе прямого применения генетического алгоритма, когда все гиперпараметры были представлены числами с плавающей точкой. Этот подход позволил улучшить результаты поиска на сетке.

В следующей главе мы рассмотрим чарующие модели машинного обучения на основе нейронных сетей и глубокого обучения и применим генетические алгоритмы, чтобы улучшить их качество.

Для дальнейшего чтения

За дополнительными сведениями по вопросам, рассмотренным в этой главе, рекомендуем обратиться к следующим источникам:

- введение в настройку гиперпараметров: книга Alan Fontaine «Mastering Predictive Analytics with scikit-learn and TensorFlow», https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781789617740/2/ch02lvl1sec16/introduction-to-hyperparameter-tuning;
- библиотека sklearn-deap на GitHub: https://github.com/rsteca/sklearn-deap;
- сравнение EvolutionaryAlgorithmSearchCV с GridSearchCV и RandomizedSearchCV: https://github.com/rsteca/sklearn-deap/blob/master/test.ipynb;
- классификатор ADABoost в библиотеке sklearn: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html;
- репозиторий машинного обучения Калифорнийского университета в Ирвайне: https://archive.ics.uci.edu/ml/index.php.

Глава 9

Оптимизация архитектуры сетей глубокого обучения

В этой главе описывается, как с помощью генетических алгоритмов улучшить качество моделей машинного обучения на основе искусственных нейронных сетей за счет оптимизации их архитектуры. Мы начнем с краткого введения в нейронные сети и глубокое обучение. Затем познакомимся с набором данных об ирисах Iris и многослойным перцептроном, после чего продемонстрируем подход к оптимизации архитектуры сети с помощью генетического алгоритма. Впоследствии мы обобщим этот подход, объединив оптимизацию архитектуры с настройкой гиперпараметров в одном решении на основе генетического алгоритма.

Будут рассмотрены следующие вопросы:

- основные понятия искусственных нейронных сетей и глубокого обучения;
- набор данных Iris и классификатор на основе **многослойного перцептрона** (**МСП**);
- повышение качества классификатора посредством оптимизации архитектуры сети;
- дальнейшее повышение качества классификатора посредством объединения оптимизации архитектуры сети с настройкой гиперпараметров.

Мы начнем эту главу с обзора искусственных нейронных сетей. Если вы профессионально занимаетесь анализом и обработкой данных, можете пропустить вступительные разделы.

Технические требования

В этой главе нам понадобится Python 3 и следующие библиотеки:

- `deap`;
- `numpy`;

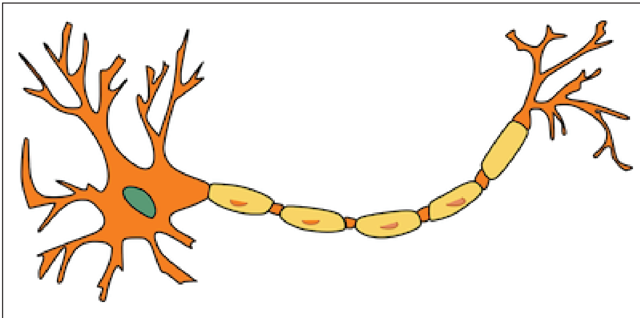
- `sklearn`.

Кроме того, мы будем использовать набор данных Iris Калифорнийского университета в Ирвайне (https://archive.ics.uci.edu/ml/datasets/Iris).

Код приведенных в этой главе программ можно найти в репозитории книги на GitHub по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter09. Чтобы увидеть код в действии, посмотрите видео по адресу http://bit.ly/317KCXA.

Искусственные нейронные сети и глубокое обучение

Нейронные сети – одна из самых широко используемых моделей в машинном обучении. В их основе лежат результаты исследований человеческого мозга. Главными структурными элементами являются нейроны, моделирующие биологические нейроны, изображенные на рисунке ниже.

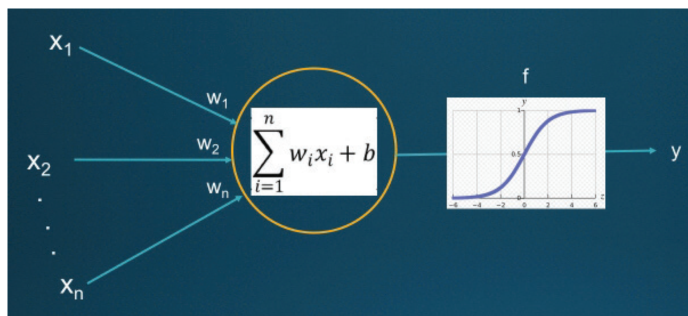


Модель биологического нейрона

Источник: https://pixabay.com/vectors/neuron-nerve-cell-axon-dendrite-296581/

Дендриты, окружающие тело клетки и показанные в левой части рисунка, являются проводниками входных сигналов от похожих клеток, а длинный аксон, исходящий из тела клетки, играет роль выхода и может соединяться с другими клетками.

Эту структуру пытается имитировать показанная ниже искусственная модель, называемая перцептроном.



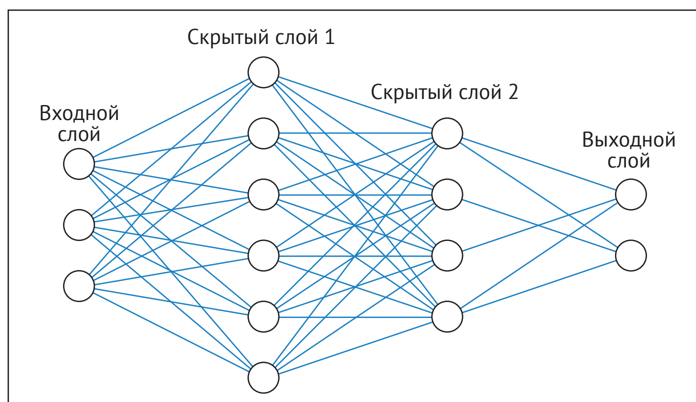
Модель искусственного нейрона – перцептрон

Перцептрон вычисляет выход, умножая каждое входное значение на некоторый вес; результаты складываются, и к сумме прибавляется смещение. Затем к результату применяется нелинейная функция активации, которая возвращает выход. Эта модель повторяет поведение биологического нейрона, который возбуждается (посылает серию импульсов через свой выходной канал), если взвешенная сумма входных сигналов оказывается больше некоторого порога.

Модель перцептрона можно использовать для простых задач классификации и регрессии, если настроить веса и смещение, так чтобы входы отображались на желаемые выходы. Но если соединить несколько перцептронов в так называемый **многослойный перцептрон**, то получается гораздо более мощная модель, которую мы опишем в следующем подразделе.

Многослойный перцептрон

Многослойный перцептрон (МСП) – это обобщение идеи перцептрона: объединение нескольких блоков, каждый из которых реализует перцептрон. Блоки в МСП собраны в слои, и каждый слой соединен со следующим. Базовая структура МСП показана на рисунке ниже.



Базовая структура многослойного перцептрона

Многослойный перцептрон состоит из трех основных частей:

- **входной слой:** получает входные значения и соединяет каждое из них с каждым нейроном следующего слоя;
- **выходной слой:** возвращает результаты, вычисленные МСП. Если МСП используется в качестве классификатора, то каждый выход представляет один класс. Если же МСП используется для регрессии, то выходной блок только один и возвращает непрерывное значение;
- **скрытые слои:** именно они составляют истинную мощь модели и придают ей сложность. На рисунке выше показано только два скрытых слоя, но их может быть и больше, разных размеров; все они располагаются между входным и выходным слоем. По мере увеличения количества скрытых слоев сеть становится глубже и может выполнять все более сложное нелинейное отображение входов на выходы.

Обучение этой модели сводится к корректировке весов и смещений каждого блока. Обычно для этого применяются алгоритмы **обратного распространения**. Их идея состоит в том, чтобы минимизировать расхождение между фактическими и желательными результатами, распространяя ошибку от выходного слоя к входному (в обратном направлении). Веса отдельных блоков – нейронов – изменяются так, что чем больше вклад нейрона в ошибку, тем сильнее корректируется его вес.

На протяжении многих лет ограничения вычислительных возможностей оборудования позволяли включать не более двух-трех скрытых слоев, но новейшие достижения полностью изменили картину. Мы объясним, что произошло, в следующем подразделе.

Глубокое обучение и сверточные нейронные сети

Сравнительно недавно алгоритмы обратного распространения совершили гигантский скачок, в результате чего стало возможно увеличить количество скрытых слоев сети. В таких глубоких нейронных сетях каждый слой может интерпретировать сравнительно простые абстрактные концепции, которым обучились предыдущие слои, и выдавать на выходе концепции более высокого уровня. Например, в задаче распознавания лиц первый слой обрабатывает пиксели изображения и обучается обнаруживать границы в разных направлениях. Следующий слой собирает границы в линии, углы и т. д., следующий за ним выделяет черты лица, например нос и губы, и, наконец, последний объединяет их и строит лицо.

Впоследствии была предложена идея **сверточной нейронной сети**, которая позволяет уменьшить количество блоков глубокой нейронной сети, обрабатывающей двумерную информацию (например, изображения), поскольку рассматривает близлежащие участки иначе, чем отдаленные. Такие модели оказываются особенно успешными при обработке изображений и видео. Помимо **полносвязных слоев**, аналогичных скрытым слоям многослойного перцептрона, в них используются **пулинговые слои** (слои **понижающей передискретизации**), которые агрегируют выходы нейронов предыдущих слоев, и **сверточные слои**, которые используются как фильтры

для обнаружения определенных признаков (например, границ определенной ориентации).

Обучение глубоких нейронных сетей требует большого объема вычислений и часто производится с применением **графических процессоров** (GPU), которые эффективнее обычных процессоров реализуют алгоритмы обратного распространения. Специализированные библиотеки глубокого обучения, например TensorFlow, умеют работать с вычислительными платформами на базе GPU. Но в этой главе мы для простоты будем пользоваться реализацией МСП, предлагаемой библиотекой `sklearn`, и простым набором данных. Однако все принципы применимы и к более сложным сетям и наборам данных.

В следующем разделе мы посмотрим, как архитектуру МСП можно оптимизировать с помощью генетического алгоритма.

Оптимизация архитектуры классификатора на основе глубокой сети

При создании модели глубокой нейронной сети для задачи машинного обучения одно из важнейших решений, которые предстоит принять, связано с выбором архитектуры сети. В случае многослойного перцептрона количество блоков во входном и выходном слоях определяется характеристиками поставленной задачи. Поэтому все решения относятся к скрытым слоям – их количеству и числу нейронов в каждом слое. Существуют эвристические соображения, упрощающие принятие таких решений, но чаще всего выбор архитектуры сводится к утомительному методу проб и ошибок.

Один из подходов к выбору параметров архитектуры сети состоит в том, чтобы рассматривать их как гиперпараметры модели, поскольку они задаются до начала обучения. В этом разделе мы примем данный подход и воспользуемся генетическим алгоритмом для поиска наилучшей комбинации скрытых слоев точно так же, как поступали при выборе лучших значений гиперпараметров в предыдущей главе. Начнем с постановки задачи – классификации ирисов.

Набор данных Iris

Один из наиболее изученных наборов данных, Iris (https://archive.ics.uci.edu/ml/datasets/Iris), содержит результаты измерений лепестков и чашелистиков трех видов ирисов (Iris setosa, Iris virginica и Iris versicolor), проделанных биологами в 1936 году.

Набор данных содержит по 50 примеров каждого из трех видов и включает следующие признаки:

- sepal_length – длина чашелистика (см);
- sepal_width – ширина чашелистика (см);
- petal_length – длина лепестка (см);
- petal_width – ширина лепестка (см).

Этот набор данных входит в состав библиотеки `sklearn`, для его инициализации нужно выполнить следующие действия:

```
from sklearn import datasets

data = datasets.load_iris()
X = data['data']
y = data['target']
```

В своих экспериментах мы будем использовать МСП-классификатор в сочетании с этим набором и воспользуемся возможностями генетических алгоритмов для нахождения архитектуры сети – количества скрытых слоев и числа нейронов в каждом слое, – при которой достигается наилучшая верность классификации.

Коль скоро мы работаем с генетическим алгоритмом, нужно придумать, как представить архитектуру в виде хромосомы.

Представление конфигурации скрытого слоя

Поскольку архитектура МСП определяется конфигурацией скрытого слоя, подумаем, как можно представить эту конфигурацию в нашем решении. Конфигурация скрытого слоя в модели многослойного перцептрона из библиотеки `sklearn` (https://scikit-learn.org/stable/modules/neural_networks_supervised.html) определяется кортежем `hidden_layer_sizes`, передаваемым конструктору в качестве параметра. По умолчанию этот кортеж равен (100,), т. е. имеется один скрытый слой, состоящий из 100 блоков. Чтобы сконфигурировать МСП с тремя скрытыми слоями по 20 блоков в каждом, нужно было бы задать кортеж (20, 20, 20). Прежде чем приступать к реализации оптимизатора конфигурации скрытых слоев на базе генетического алгоритма, нужно определить подходящий формат хромосомы.

Нам нужна хромосома, которая кодировала бы как количество слоев, так и число блоков в каждом слое. Один из вариантов – хромосома переменной длины, которая прямо транслируется в кортеж переменной длины, передаваемый в параметре модели `hidden_layer_sizes`. Однако при этом пришлось бы придумывать специальные, возможно, громоздкие генетические операторы. Чтобы можно было воспользоваться стандартными генетическими операторами, мы остановимся на представлении фиксированной длины. Тогда максимальное количество слоев задается заранее; все эти слои всегда представлены в хромосоме, но необязательно присутствуют в решении. Например, если мы решим ограничиться четырьмя скрытыми слоями, то хромосома будет выглядеть так:

$[n1, n2, n3, n4]$,

где n_i – число блоков в i-м слое.

Но для управления числом скрытых слоев сети некоторые из этих значений могут быть нулевыми или отрицательными. Это означает, что в сеть не нужно добавлять больше слоев. Проиллюстрируем на примерах.

- Хромосома [10, 20, –5, 15] транслируется в кортеж (10, 20), поскольку –5 служит признаком завершения.
- Хромосома [10, 0, –5, 15] транслируется в кортеж (10,), поскольку 0 служит признаком завершения.
- Хромосома [10, 20, 5, –15] транслируется в кортеж (10, 20, 5), поскольку –15 служит признаком завершения.
- Хромосома[10, 20, 5, 15] транслируется в кортеж (10, 20, 5, 15).

Чтобы гарантировать наличие хотя бы одного скрытого слоя, мы можем потребовать, чтобы первый параметр всегда был больше нуля. Остальные параметры могут иметь различные распределения вокруг нуля, чтобы мы могли управлять вероятностью того, что они станут признаками завершения.

Кроме того, хотя эта хромосома состоит только из целых чисел, мы решили все же работать с числами с плавающей точкой, как в предыдущей главе. Это удобно, потому что позволяет использовать существующие генетические операторы, оставив себе возможность обобщить хромосому, так чтобы она могла включать параметры других типов. Позже мы так и поступим. Числа типа float можно преобразовать в целые с помощью функции `round()`. Проиллюстрируем этот обобщенный подход на примерах.

- Хромосома [9.35, 10.71, –2.51, 17.99] транслируется в кортеж (9, 11).
- Хромосома [9.35, 10.71, 2.51, –17.99] транслируется в кортеж (9, 11, 3).

Чтобы оценить хромосому, представляющую архитектуру, мы должны преобразовать ее обратно в кортеж слоев, создать МСП-классификатор с такими слоями, обучить его и оценить верность. Как это делается, показано в следующем подразделе.

Оценка верности классификатора

Класс, инкапсулирующий оценку верности МСП-классификатора для набора данных Iris, называется `MlpLayersTest` и находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter09/mlp_layers_test.py.

Ниже описаны его основные функции.

1. Метод класса `convertParam()` принимает список `params`. Это хромосома, описанная в предыдущем разделе, она содержит числа с плавающей точкой, представляющие не более четырех скрытых слоев. Метод преобразует этот список в кортеж `hidden_layer_sizes`:

```
if round(params[1]) <= 0:
    hiddenLayerSizes = round(params[0]),
elif round(params[2]) <= 0:
    hiddenLayerSizes = (round(params[0]), round(params[1]))
elif round(params[3]) <= 0:
    hiddenLayerSizes = (round(params[0]), round(params[1]), round(params[2]))
else:
    hiddenLayerSizes = (round(params[0]), round(params[1]), round(params[2]),
round(params[3]))
```

2. Метод `getAccuracy()` принимает список `params`, представляющий конфигурацию скрытых слоев, с помощью метода `convertParam()` преобразует его в кортеж `hidden_layer_sizes` и инициализирует МСП-классификатор этим кортежем:

```
hiddenLayerSizes = self.convertParams(params)
self.classifier = MLPClassifier(hidden_layer_sizes=hiddenLayerSizes)
```

Затем он определяет верность классификатора с помощью *k*-группового перекрестного контроля, как при работе с набором данных Wine:

```
cv_results = model_selection.cross_val_score(self.classifier,
                                             self.X,
                                             self.y,
                                             cv=self.kfold,
                                             scoring='accuracy')
return cv_results.mean()
```

Класс `MlpLayersTest` используется в оптимизаторе на базе генетического алгоритма, как описано в следующем подразделе.

Оптимизация архитектуры МСП с помощью генетического алгоритма

Теперь, умея представлять конфигурацию МСП для классификации данных об ирисах и зная, как определить верность классификатора для любой конфигурации, мы можем перейти к созданию генетического алгоритма для поиска оптимальной конфигурации – количества скрытых слоев (не более 4) и числа блоков в каждом слое, – при которой достигается наилучшая верность. Наше решение реализовано Python-программой, находящейся в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter09/01-optimize-mlp-layers.py.

Ниже описаны основные части этого решения.

1. Сначала зададим нижнюю и верхнюю границы чисел с плавающей точкой, представляющих скрытый слой. Для первого скрытого слоя возьмем диапазон [5, 15], а для остальных нижние границы будут монотонно уменьшающимися отрицательными значениями, так чтобы шансы слоя оказаться последним увеличивались.

```
# [layer_layer_1_size, hidden_layer_2_size,
hidden_layer_3_size, hidden_layer_4_size]
BOUNDS_LOW = [ 5, -5, -10, -20]
BOUNDS_HIGH = [15, 10, 10, 10]
```

2. Затем создадим экземпляр класса `MlpLayersTest`, с помощью которого будем проверять различные комбинации параметров архитектуры:

```
test = mlp_layers_test.MlpLayersTest(RANDOM_SEED)
```

3. Поскольку мы стремимся максимизировать верность классификатора, определим единственную цель – максимизирующую стратегию приспособления:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

4. Теперь применим тот же подход, что в предыдущей главе: поскольку решение представлено списком чисел типа float, каждое в своем диапазоне, в цикле обойдем все пары, состоящие из нижней и верхней границ, и для каждого диапазона создадим в инструментарии свой оператор (`layer_size_attribute`), который впоследствии будет генерировать случайное число в этом диапазоне:

```
for i in range(NUM_OF_PARAMS):
    # "layer_size_attribute_0", "layer_size_attribute_1", ...
    toolbox.register("layer_size_attribute_" + str(i),
                     random.uniform,
                     BOUNDS_LOW[i],
                     BOUNDS_HIGH[i])
```

5. Далее создадим кортеж `layer_size_attributes`, содержащий только что созданные генераторы случайных чисел с плавающей точкой для каждого скрытого слоя:

```
layer_size_attributes = ()
for i in range(NUM_OF_PARAMS):
    layer_size_attributes = layer_size_attributes + \
(toolbox.__getattribute__("layer_size_attribute_" + str(i)),)
```

6. Теперь можно воспользоваться этим кортежем в сочетании со встроенным в DEAP оператором `initCycle()`, чтобы создать оператор `individualCreator`, который порождает экземпляр индивидуума со слоями случайно сгенерированных размеров:

```
toolbox.register("individualCreator",
                 tools.initCycle,
                 creator.Individual,
                 layer_size_attributes,
                 n=1)
```

7. Далее мы просим генетический алгоритм использовать метод `getAccuracy()` экземпляра `MlpLayersTest` для вычисления приспособленности. Напомним, что метод `getAccuracy()`, описанный в предыдущем подразделе, преобразует переданного ему индивидуума – список четырех чисел типа float – в кортеж размеров скрытых слоев. Затем этот кортеж используется для конфигурирования МСП-классификатора, после чего вычисляется его верность с помощью k-группового перекрестного контроля:

```
def classificationAccuracy(individual):
    return test.getAccuracy(individual),

toolbox.register("evaluate", classificationAccuracy)
```

8. Что касается генетических операторов, то мы повторяем конфигурацию, описанную в предыдущей главе. В качестве оператора отбора, как обычно, используем турнир размера 2, а для скрещивания и мутации берем операторы, специализированные для хромосом в виде списков чисел с плавающей точкой и передаем им границы, определенные нами для каждого скрытого слоя:

```
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0 / NUM_OF_PARAMS)
```

9. Как и раньше, будем использовать элитистский подход, т. е. без изменения копировать лучших на данный момент индивидуумов из зала славы в следующее поколение:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                  toolbox,
                                                  cxpb=P_CROSSOVER,
                                                  mutpb=P_MUTATION,
                                                  ngen=MAX_GENERATIONS,
                                                  stats=stats,
                                                  halloffame=hof,
                                                  verbose=True)
```

После прогона алгоритма для 10 поколений с размером популяции 20 получается такой результат:

```
gen nevals max      avg
0   20     0.666667 0.416333
1   17     0.693333 0.487
2   15     0.76     0.537333
3   14     0.76     0.550667
4   17     0.76     0.568333
5   17     0.76     0.653667
6   14     0.76     0.589333
7   15     0.76     0.618
8   16     0.866667 0.616667
9   16     0.866667 0.666333
10  16     0.866667 0.722667
- Лучшее решение: 'hidden_layer_sizes'=(15, 5, 8), accuracy = 0.8666666666666666
```

Как видим, в рамках заданных нами ограничений лучшей оказалась комбинация трех скрытых слоев с размерами 15, 5, 8. При таких параметрах достигается верность классификации 86.7 %.

Для рассматриваемой задачи подобная верность выглядит приемлемо. Однако можно добиться большего.

Объединение оптимизации архитектуры с настройкой гиперпараметров

Занимаясь оптимизацией архитектуры сети – параметров скрытых слоев, – мы оставили для самого МСП-классификатора параметры по умолчанию. Однако в предыдущей главе мы видели, что с помощью настройки гиперпараметров можно повысить качество классификатора. Можно ли включить настройку в наш процесс оптимизации? Как вы, наверное, догадались, ответ утвердительный. Но сначала разберемся, какие именно гиперпараметры мы собираемся оптимизировать.

В реализации МСП-классификатора из библиотеки `sklearn` есть много настраиваемых гиперпараметров. Мы ограничимся следующими:

Имя	Тип	Описание	Значение по умолчанию
`activation`	`{'tanh', 'relu', 'logistic'}`	Функция активации в скрытых слоях	`'relu'`
`solver`	`{'sgd', 'adam', 'lbfgs'}`	Алгоритм оптимизации весов	`'adam'`
`alpha`	`float`	Параметр регуляризации	`0.0001`
`learning_rate`	`{'constant', 'invscaling', 'adaptive'}`	Характер изменения скорости обучения при обновлении весов	`'constant'`

В предыдущей главе мы видели, что представление хромосомы с числами типа float позволяет включать различные типы гиперпараметров в процесс оптимизации на базе генетического алгоритма. Поскольку конфигурация скрытых слоев представлена именно такой хромосомой, мы теперь можем включить в оптимизацию и другие параметры, дополнив уже имеющуюся хромосому. Ниже описано, как это сделать.

Представление решения

К уже имеющейся конфигурации сети вида [*n1, n2, n3, n4*] мы добавим еще четыре гиперпараметра:

- `activation` может принимать одно из трех значений: `'tanh'`, `'relu'` или `'logistic'`. Поэтому представим этот параметр числом типа float в диапазоне `[0, 2.99]`. Для преобразования числа типа float в одно из выше-

перечисленных значений применим к нему функцию `floor()`, которая возвращает значение 0, 1 или 2. Затем заменим 0 на `tanh`, 1 – на `relu`, а 2 – на `logistic`;
- `solver` может принимать одно из трех значений: `'sgd'`, `'adam'` или `'lbfgs'`. Как и параметр `activation`, его можно представить числом типа float в диапазоне `[0, 2.99]`;
- `alpha` уже является числом с плавающей точкой, так что никакого преобразования не нужно. Оно будет ограничено диапазоном `[0.0001, 2.0]`;
- `learning_rate` может принимать одно из трех значений: `'constant'`, `'invscaling'` или `'adaptive'`. И снова для его представления можно использовать число типа float в диапазоне `[0, 2.99]`.

Вычисление верности классификатора

Оценка верности МСП-классификатора с заданной комбинацией скрытых слоев и гиперпараметров реализована в Python-классе `MlpHyperparametersTest`, находящемся в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter09/mlp_hyperparameters_test.py.

В основе этого класса лежит класс `MlpLayersTest`, которым мы пользовались, когда оптимизировали конфигурацию скрытых слоев, но с несколькими модификациями.

1. Метод `convertParam()` теперь обрабатывает список `params`, в котором первые четыре элемента (`params[0]`, ... , `params[3]`), как и раньше, представляют размеры скрытых слоев, а элементы `params[4]`, ... , `params[7]` – добавленные для оценки гиперпараметры. Поэтому в метод добавлены следующие строчки для преобразования дополнительных параметров в их истинные значения, которые затем можно будет подать на вход МСП-классификатора:

```
activation = ['tanh', 'relu', 'logistic'][floor(params[4])]
solver = ['sgd', 'adam', 'lbfgs'][floor(params[5])]
alpha = params[6]
learning_rate = ['constant', 'invscaling', 'adaptive'][floor(params[7])]
```

2. Аналогично метод `getAccuracy()` теперь обрабатывает пополненный список `params`. Он конфигурирует МСП-классификатор преобразованными значениями всех параметров, а не только задает конфигурацию скрытых слоев.

```
hiddenLayerSizes, activation, solver, alpha, learning_rate =
self.convertParams(params)

self.classifier = MLPClassifier(random_state=self.randomSeed,
                                hidden_layer_sizes=hiddenLayerSizes,
                                activation=activation,
                                solver=solver,
                                alpha=alpha,
                                learning_rate=learning_rate)
```

Этот класс MlpHyperparametersTest используется в генетическом оптимизаторе, как описано в следующем подразделе.

Оптимизация объединенной конфигурации МСП с помощью генетического алгоритма

Генетический поиск лучшей комбинации скрытых слоев и гиперпараметров реализован в программе 02-optimize-mlphyperparameters.py, которая находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter09/02-optimize-mlphyperparameters.py.

Благодаря унифицированному представлению всех параметров числами с плавающей точкой эта программа почти идентична программе из предыдущего раздела. Главное отличие – определение списков BOUNDS_LOW и BOUNDS_HIGH, которые содержат диапазоны параметров. К четырем прежним диапазонам – по одному для каждого скрытого слоя – мы добавили еще четыре, соответствующих четырем рассмотренным выше гиперпараметрам:

```
# 'hidden_layer_sizes': первые четыре значения
# 'activation': 0..2.99
# 'solver': 0..2.99
# 'alpha': 0.0001..2.0
# 'learning_rate': 0..2.99
BOUNDS_LOW = [ 5, -5, -10, -20, 0, 0, 0.0001, 0 ]
BOUNDS_HIGH = [15, 10, 10, 10, 2.999, 2.999, 2.0, 2.999]
```

И это все – теперь программа может обработать дополнительные параметры без каких-либо новых изменений.

После прогона алгоритма для 5 поколений с размером популяции 20 получается такой результат:

```
gen nevals max      avg
0   20     0.933333 0.447333
1   16     0.933333 0.631667
2   15     0.94     0.736667
3   16     0.94     0.849
4   15     0.94     0.889667
5   17     0.946667 0.937
- Лучшее решение:
'hidden_layer_sizes'=(8, 8)
'activation'='relu'
'solver'='lbfgs'
'alpha'=0.572971105096338
'learning_rate'='invscaling'
=> верность = 0.9466666666666667
```

Из-за различий между операционными системами результаты в вашей системе могут немного отличаться от показанных выше.

Отсюда следует, что в пределах определенных нами диапазонов лучшей комбинацией скрытых слоев и гиперпараметров является следующая:

- два скрытых слоя по 8 блоков в каждом;
- параметр `activation` равен `'relu'` – это значение по умолчанию;
- параметр `solver` равен `'lbfgs'`, а не `'adam'`, как по умолчанию;
- параметр `learning_rate` равен `'invscaling'`, а не `'constant'`, как по умолчанию;
- параметр `alpha` равен 0.572 – значительно больше, чем значение по умолчанию 0.0001.

При такой комбинированной оптимизации точность классификации равна приблизительно 94.7 % – существенное улучшение предыдущих результатов, достигнутое при меньшем числе скрытых слоев и блоков в каждом слое.

Резюме

В этой главе мы познакомились с основными понятиями искусственных нейронных сетей и глубокого обучения. Мы рассмотрели набор данных Iris и классификатор на основе **многослойного перцептрона** (**МСП**), а затем определили, что понимается под оптимизацией архитектуры сети. Далее продемонстрировали оптимизацию архитектуры МСП-классификатора с помощью генетического алгоритма. Наконец, мы объединили оптимизацию архитектуры сети и гиперпараметров модели в одном генетическом алгоритме и сумели еще улучшить качество классификатора.

До сих пор мы говорили только об обучении с учителем. В следующей главе рассмотрим применение генетических алгоритмов к обучению с подкреплением – увлекательной и быстро развивающейся отрасли машинного обучения.

Для дальнейшего чтения

За дополнительными сведениями по вопросам, рассмотренным в этой главе, рекомендуем обратиться к следующим источникам:

- Gianmario Spacagna, Daniel Slater et al. «Python Deep Learning», второе издание;
- James Loy «Neural Network Projects with Python»;
- МСП-классификатор в библиотеке scikit-learn: https://scikit-learn.org/stable/modules/neural_networks_supervised.html;
- репозиторий машинного обучения Калифорнийского университета в Ирвайне: https://archive.ics.uci.edu/ml/index.php.

Глава 10

Генетические алгоритмы и обучение с подкреплением

В этой главе мы продемонстрируем применение генетических алгоритмов к обучению с подкреплением – быстро развивающейся дисциплине машинного обучения, способной решать сложные задачи. Для этого мы рассмотрим две тестовые окружающие среды из библиотеки OpenAI Gym. Мы начнем с краткого введения в обучение с подкреплением, а затем дадим обзор OpenAI Gym – комплекта инструментов для сравнения разработки алгоритмов обучения с подкреплением, – а также его интерфейса, написанного на Python. Затем перейдем к двум окружающим средам Gym, *MountainCar* (машина на горе) и *CartPole* (балансирование стержня), и разработаем программы для их решения на базе генетических алгоритмов.

Будут рассмотрены следующие вопросы:

- основные понятия обучения с подкреплением;
- знакомство с проектом OpenAI Gym и его интерфейсом;
- применение генетического алгоритма к окружающей среде *MountainCar*;
- применение генетического алгоритма в сочетании с нейронной сетью к окружающей среде *CartPole*.

Мы начнем эту главу с обзора основных понятий обучения с подкреплением. Если вы профессионально занимаетесь анализом и обработкой данных, можете пропустить этот вступительный раздел.

Технические требования

В этой главе нам понадобится Python 3 и следующие библиотеки:

- `deap`;
- `numpy`;
- `sklearn`;
- `gym` – будет описана здесь же.

В этой главе используются окружающие среды Gym *MountainCar-v0* (https://gym.openai.com/envs/MountainCar-v0/) и *CartPole-V1* (https://gym.openai.com/envs/CartPole-v1/).

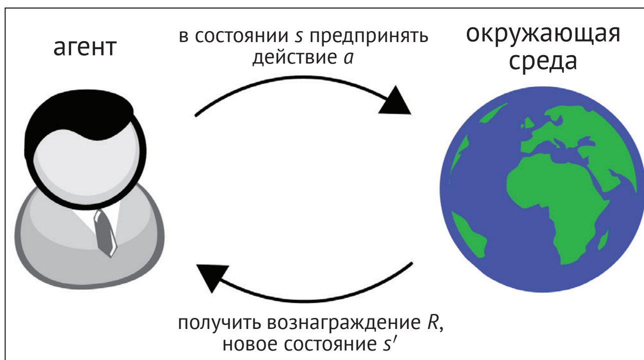
Код приведенных в данной главе программ можно найти в репозитории книги на GitHub по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter10. Чтобы увидеть код в действии, посмотрите видео по адресу http://bit.ly/2Oey7V0.

Обучение с подкреплением

В предыдущих главах мы рассмотрели несколько тем, относящихся к машинному обучению, уделив особое внимание задачам обучения с учителем. Конечно, обучение с учителем крайне важно и имеет множество практических применений, но в настоящее время обучение с подкреплением, похоже, становится самой увлекательной и многообещающей отраслью машинного обучения. Причины – в разнообразии сложных повседневных задач, которые потенциально можно решить средствами обучения с подкреплением. В марте 2016 года программа *AlphaGo*, основанная на обучении с подкреплением и предназначенная для игры в очень сложную игру го, сумела победить сильнейшего в мире игрока в матче, который получил широкое освещение в СМИ.

Если для обучения с подкреплением требуются **размеченные** обучающие данные, т. е. пары, состоящие из входа и соответствующего ему выхода, то для обучения с подкреплением непосредственной обратной связи **правильно–неправильно** не нужно. Ее место занимает окружающая среда, от которой требуется получить максимальное вознаграждение за длительный период. Это означает, что иногда алгоритму приходится отступать **назад**, чтобы в итоге достичь долгосрочной цели, – и мы продемонстрируем это в первом примере в данной главе.

Два основных компонента задачи обучения с подкреплением – *окружающая среда* и *агент* – показаны на рисунке ниже.



Обучение с подкреплением как взаимодействие между агентом и окружающей средой

Агент представляет алгоритм, который взаимодействует с окружающей средой и стремится решить поставленную задачу путем максимизации полного вознаграждения.

Взаимодействие между агентом и окружающей средой можно выразить в виде последовательности шагов. На каждом шаге среда сообщает агенту некоторое состояние (*s*), называемое также **наблюдением**. В свою очередь, агент предпринимает некоторое **действие** (*a*). Среда реагирует переходом в новое состояние (*s'*), а также промежуточным **вознаграждением** (*R*). Это взаимодействие повторяется, пока не будет выполнено условие остановки. Цель агента – максимизировать сумму вознаграждений, полученных на всем протяжении взаимодействия.

Несмотря на простоту формулировки, это описание подходит для чрезвычайно сложных задач и ситуаций, благодаря чему обучение с подкреплением применимо к широкому кругу приложений, в т. ч. теории игр, здравоохранению, системам управления, автоматизации цепочек поставок и исследованию операций.

В этой главе мы еще раз продемонстрируем гибкость генетических алгоритмов, применив их как вспомогательное средство в задачах обучения с подкреплением.

Генетические алгоритмы и обучение с подкреплением

Для задач обучения с подкреплением разработано много специализированных алгоритмов: Q-обучение, SARSA, DQN и др. Но поскольку в этих задачах фигурирует максимизация долгосрочного вознаграждения, мы можем рассматривать их как задачи оптимизации разных типов. Поэтому в обучении с подкреплением есть место для генетических алгоритмов, и даже не одно – два из них мы продемонстрируем в этой главе. В первом случае генетический алгоритм прямо даст оптимальную последовательность действий агента, а во втором он найдет оптимальные параметры нейросетевого контроллера, вырабатывающего такие действия.

Но прежде чем применять генетические алгоритмы к задачам обучения с подкреплением, познакомимся с комплектом инструментов для тестирования – OpenAI Gym.

OpenAI Gym

OpenAI Gym (https://github.com/openai/gym) – библиотека с открытым исходным кодом, предлагающая стандартизованный набор задач обучения с подкреплением, а также комплект инструментов для разработки и сравнения алгоритмов их решения.

OpenAI Gym содержит набор окружающих сред с общим интерфейсом `env`. Этот интерфейс разрывает связь между окружающими средами и агентами,

которых можно реализовать как угодно, лишь бы агент мог взаимодействовать со средой через интерфейс **env**. Как это выглядит, мы опишем в следующем подразделе.

Базовый пакет gym предоставляет доступ к нескольким окружающим средам. Для его установки выполните команду

```
pip install gym
```

Существуют и другие пакеты, например 'Atari', 'Box2D' и 'MuJoCo', которые предоставляют доступ к дополнительным окружающим средам самого разного характера. Некоторые пакеты зависят от системы и, стало быть, доступны не на всех платформах. Дополнительные сведения см. на странице по адресу https://github.com/openai/gym#installation.

В следующем подразделе описывается взаимодействие с интерфейсом **env**.

Интерфейс env

Для создания окружающей среды нужно вызвать метод make(), передав ему имя среды:

```
env = gym.make('MountainCar-v0')
```

Подробные сведения об имеющихся окружающих средах см. по ссылкам:

- https://github.com/openai/gym/blob/master/docs/environments.md;
- https://gym.openai.com/envs/.

Созданную среду следует инициализировать методом reset():

```
observation = env.reset()
```

Этот метод возвращает объект наблюдения observation, описывающий начальное состояние среды. Внутреннее содержимое наблюдения зависит от среды.

В соответствии с описанием цикла обучения с подкреплением (см. предыдущий раздел) взаимодействие с окружающей средой состоит из двух шагов: отправка ей действия и получение в ответ промежуточного вознаграждения и нового состояния. Это реализуется методом step():

```
observation, reward, done, info = env.step(action)
```

Помимо объекта observation, описывающего новое состояние, и промежуточного вознаграждения reward (число с плавающей точкой), этот метод возвращает следующие значения:

- done: булев признак, равный True, если текущий прогон (называемый также **эпизодом**) завершился, например если агент истратил жизнь или успешно справился с задачей;
- info: словарь, содержащий дополнительную информацию, полезную для отладки. Но на этапе обучения агент не должен пользоваться этой информацией.

В любой момент времени можно получить визуальное представление окружающей среды:

```
env.render()
```

Это представление зависит от среды.

Наконец, среду можно закрыть, в результате чего выполняется необходимая очистка:

```
env.close()
```

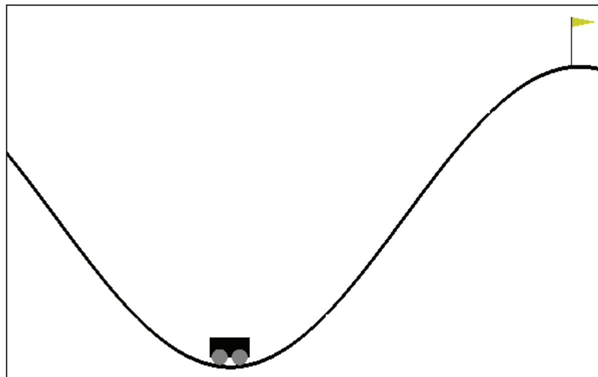
Если этот метод не был вызван, то среда автоматически закроется, когда Python в следующий раз запустит сборщик мусора (процесс нахождения и освобождения памяти, которая уже не используется программой) или в момент выхода из программы.

> Подробные сведения об интерфейсе **env** имеются по адресу http://gym.openai.com/docs/.

В следующем разделе мы продемонстрируем весь цикл взаимодействия на примере окружающей среды `MountainCar` из комплекта gym.

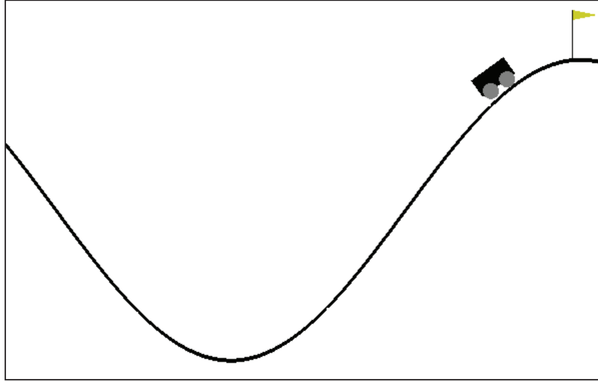
Решение окружающей среды MountainCar

Окружающая среда MountainCar-v0 моделирует поведение автомобиля на одномерной трассе между двумя холмами. Моделирование начинается, когда машина находится в долине, как показано на рисунке ниже.



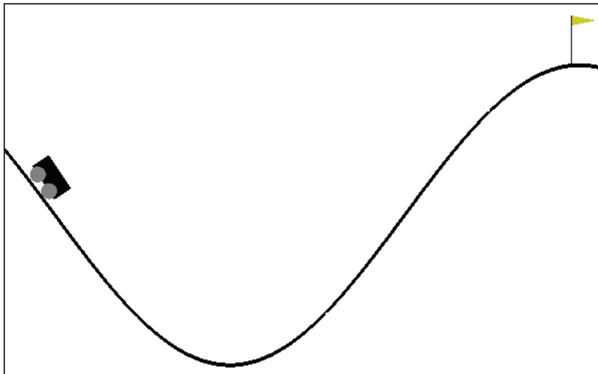
Окружающая среда MountainCar – начальная точка

Наша цель – заехать на более высокий холм справа, где находится флажок.



Окружающая среда MountainCar –
машина поднимается на правый холм

Но двигатель машины слишком слаб, и просто въехать на холм машина не может. Единственный способ добиться цели – разгонять машину, двигаясь взад-вперед, набрав достаточно инерции для подъема. Поможет подъем на левый холм, поскольку набранной при спуске инерции может хватить, чтобы добраться до вершины правого. См. рисунок ниже.



Окружающая среда MountainCar – скатывание с левого холма

Эта окружающая среда – прекрасный пример того, как временное отступление (движение влево) может помочь в достижении конечной цели (проделать весь путь вправо).

В этом взаимодействии ожидаемое действие – целое число, принимающее одно из трех значений:

- 0: разгон влево;
- 1: движение по инерции;
- 2: разгон вправо.

Объект `observation` содержит два числа с плавающей точкой, описывающих положение и скорость машины, например:

```
[-1.0260268, -0.03201975]
```

На каждом шаге начисляется вознаграждение –1 до тех пор, пока не будет достигнута цель (находящаяся в точке с координатой 0.5). Эпизод заканчивается через 200 шагов, если цель не была достигнута раньше.

На момент написания книги еще не было определено, когда задача о машине на горе считается решенной, поэтому мы просто попытаемся достичь флажка за 200 шагов или менее из заданной начальной позиции. Чтобы найти последовательность действий, приводящую к подъему на вершину холма, мы построим решение на основе генетического алгоритма. Как обычно, начнем с представления потенциального решения.

Дополнительные сведения об окружающей среде *MountainCar-v0* можно найти по ссылкам:

- **MountainCar-v0**: https://gym.openai.com/envs/MountainCar-v0/;
- **MountainCar-v0**: https://github.com/openai/gym/wiki/MountainCar-v0.

Представление решения

Поскольку окружающая среда `MountainCar` управляется последовательностью действий, принимающих значения 0, 1 или 2, а всего в эпизоде может быть не более 200 действий, напрашивается очевидное представление решения списком длины 200, содержащим числа 0, 1, 2, например:

```
[0, 1, 2, 0, 0, 1, 2, 2, 1, ... , 0, 2, 1, 1]
```

Элементы списка интерпретируются как действия по управлению машиной с целью добраться до флажка. Если машина справилась с задачей меньше, чем за 200 шагов, то оставшиеся элементы списка не используются.

Теперь надо определить, как оценивать решение такого вида.

Оценивание решения

Очевидно, что одно лишь вознаграждение не дает достаточно информации для оценки данного решения или сравнения двух решений. Вознаграждение определено так, что в конце любого эпизода, в котором машина не достигла флажка, оно составит –200. Для сравнения двух решений, не достигших цели, нам все-таки нужно знать, какое лучше. Поэтому в дополнение к величине вознаграждения мы будем использовать для оценки решения конечное положение машины. Если машина не добралась до вершины, то в роли оценки выступает расстояние до флажка. Если же добралась, то базовая оценка равна нулю, а из нее вычитается число неиспользованных шагов, так что итоговая оценка будет отрицательной. Поскольку мы стремимся к наименьшей возможной оценке, то такое определение отдает предпочтение решениям, при которых машина добралась до флажка, совершив наименьшее количество действий.

Эта процедура оценивания реализована в классе `MountainCar`, описанном в следующем подразделе.

Представление задачи на Python

Для инкапсуляции задачи `MountainCar` мы написали на Python класс `MountainCar`, код которого находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter10/mountain_car.py.

Конструктору класса передается начальное значение генератора случайных чисел. Класс предоставляет следующие методы:

- `getScore(actions)`: вычисляет оценку решения, представленного списком действий. Для этого инициализируется эпизод взаимодействия со средой `MountainCar` и в нем выполняются указанные действия. Чем меньше оценка, тем лучше;
- `saveActions(actions)`: сохраняет список действий в файле в формате pickle (формат сериализации и десериализации в Python);
- `replaySavedActions()`: десериализует сохраненный список действий и воспроизводит его методом `replay`;
- `replay(actions)`: рисует окружающую среду и воспроизводит в ней заданный список действий, чтобы наглядно представить решение.

После того как решение сериализовано и сохранено методом `saveActions()`, можно вызывать метод `main()`. Он инициализирует класс и вызывает метод `replaySavedActions()` для анимации последнего сохраненного решения.

Обычно метод `main()` используется для демонстрации лучшего решения, найденного генетическим алгоритмом, который описан в следующем подразделе.

Решение с помощью генетического алгоритма

Программа для решения задачи `MountainCar` с помощью генетического алгоритма находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter10/01-solve-mountain-car.py.

Поскольку для представления решения мы выбрали список, содержащий целые числа 0, 1, 2, эта программа напоминает ту, с помощью которой мы решили задачу о рюкзаке 0-1 в главе 4. Там решение представлялось в виде списка нулей и единиц.

Ниже описаны основные части программы.

1. Начинаем с создания экземпляра класса `MountainCar`, который позволит нам оценивать решения задачи `MountainCar`:

```
car = mountain_car.MountainCar(RANDOM_SEED)
```

2. Наша задача – минимизировать оценку, т. е. добраться до флажка за наименьшее количество шагов или же подобраться к флажку как можно ближе, поэтому определяем единственную цель – минимизирующую стратегию приспособления:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. Далее создаем в инструментарии оператор, порождающий одно из трех допустимых действий – 0, 1, 2:

```
toolbox.register("zeroOneOrTwo", random.randint, 0, 2)
```

4. Затем – оператор, инициализирующий экземпляр индивидуума такими значениями:

```
toolbox.register("individualCreator",
                 tools.initRepeat,
                 creator.Individual,
                 toolbox.zeroOneOrTwo,
                 len(car))
```

5. После этого просим генетический алгоритм вызвать метод `getScore()` для вычисления приспособленности экземпляра класса `MountainCar`. Напомним, что описанный выше метод `getScore()` инициирует эпизод взаимодействия со средой `MountainCar` и использует данного индивидуума – список действий – как входные данные для среды, пока эпизод не завершится. Затем он вычисляет оценку – чем меньше, тем лучше – по конечному положению машины. Если она добралась до флажка, то оценка может дополнительно улучшиться в зависимости от количества неиспользованных шагов.

```
def carScore(individual):
    return car.getScore(individual),

toolbox.register("evaluate", carScore)
```

6. В качестве оператора отбора, как обычно, берем турнир размера 2. Поскольку решение представлено списком целых чисел 0, 1, 2, то можно использовать двухточечный оператор скрещивания, как и в случае, когда список состоит только из нулей и единиц. Но вот для мутации нужно брать не оператор `FlipBit`, подходящий только для двоичного случая, а `UniformInt`, используемый для произвольного целочисленного диапазона. Этот оператор мы сконфигурируем для диапазона 0.2:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, low=0, up=2, indpb=1.0/len(car))
```

7. Как и раньше, будем использовать элитистский подход, т. е. без изменения копировать лучших на данный момент индивидуумов из зала славы в следующее поколение:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                  toolbox,
                                                  cxpb=P_CROSSOVER,
                                                  mutpb=P_MUTATION,
                                                  ngen=MAX_GENERATIONS,
```

```
                                              stats=stats,
                                              halloffame=hof,
                                              verbose=True)
```

8. После прогона печатаем и сохраняем лучшее найденное решение, чтобы впоследствии можно было его анимировать средствами воспроизведения, встроенными в класс `MountainCar`:

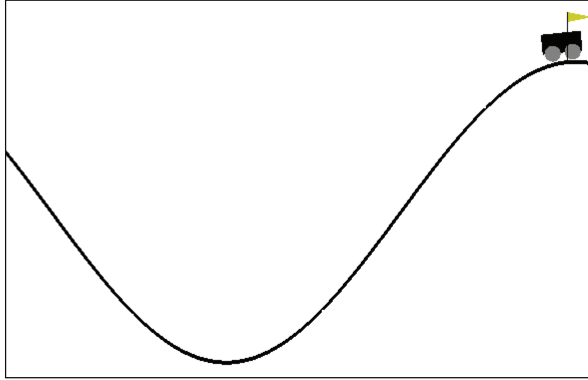
```
best = hof.items[0]
print("Лучшее решение = ", best)
print("Лучшая приспособленность = ", best.fitness.values[0])
car.saveActions(best)
```

После прогона алгоритма для 80 поколений с размером популяции 100 получается такой результат:

```
gen nevals  max         avg
0   100     0.659205    1.02616
1   78      0.659205    0.970209
...
60  75      0.00367593 0.100664
61  73      0.00367593 0.0997352
62  77      -0.005     0.100359
63  73      -0.005     0.103559
...
67  78      -0.015     0.0679005
68  80      -0.015     0.0793169
...
79  76      -0.02      0.020927
80  76      -0.02      0.0175934

Лучшее решение = [0, 1, 2, 0, 0, 1, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2, 0, ... ,
1, 0, 2, 2, 0, 2, 1]
Лучшая приспособленность = -0.02
```

Отсюда видно, что после примерно 60 поколений лучшие решения стали добираться до флажка, поскольку их оценки отрицательны. Начиная с этого момента лучшим решениям требуется все меньше и меньше шагов, оценки становятся все более отрицательными.

Как мы помним, лучшее решение было сохранено в конце прогона, поэтому теперь мы можем воспроизвести его, запустив программу `mountain_car`. Анимация показывает, что машина движется взад-вперед между двумя холмами, с каждым разом забираясь все выше, пока, наконец, не оказывается на вершине левого холма. Затем она скатывается вниз и разгоняется достаточно сильно, чтобы подняться на вершину правого холма и поравняться с флажком.



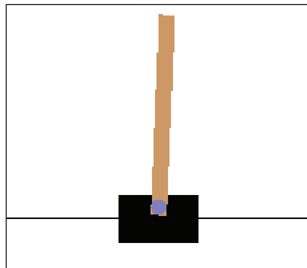
Окружающая среда MountainCar – машина добралась до цели

Решение, конечно, доставило удовольствие, но нам вообще не пришлось динамически взаимодействовать со средой. Мы сумели подняться на вершину, воспользовавшись последовательностью действий, которую наш алгоритм нашел, исходя из начального положения машины. Но вот при решении следующей задачи – окружающей среды CartPole – нам нужно будет динамически вычислять действие на каждом временном шаге в зависимости от последнего наблюдения.

Решение окружающей среды CartPole

Окружающая среда CartPole-v1 моделирует балансирование стержня, шарнирно закрепленного на тележке, которая движется влево и вправо по дорожке. Для удержания стержня в вертикальном положении необходимо прикладывать к тележке силу, направленную влево или вправо.

Стержень ведет себя как обратный маятник, в начальный момент он отклоняется от вертикального положения на небольшой угол, как показано на рисунке ниже.



Моделирование CartPole – начальное положение

Наша цель – не дать стержню упасть как можно дольше, а точнее на протяжении 500 временных шагов. За каждый шаг, на котором стержень не падает, начисляется вознаграждение +1, так что максимальное полное вознаграждение равно 500. Эпизод заканчивается досрочно в одном из двух случаев:

- стержень отклонился от вертикального положения на угол более 15°;
- тележка отъехала от центра более чем на 2.4 единицы длины.

В этих случаях вознаграждение будет меньше 500.

В данной окружающей среде возможно два действия:

- 0: толкнуть тележку влево;
- 1: толкнуть тележку вправо.

Объект `observation` содержит четыре числа с плавающей точкой, сообщающих следующую информацию:

- положение тележки (от –2.4 до 2.4);
- скорость тележки (от минус бесконечности до бесконечности);
- угол наклона шеста (от –41.8° до 41.8°);
- скорость кончика шеста (от минус бесконечности до бесконечности).

Например, мы могли бы получить такое наблюдение [0.33676587, 0.3786464, –0.00170739, –0.36586074].

В нашем решении мы используем эти значения как входные данные на каждом временном шаге, на основании которых определяется действие. Для этого воспользуемся нейросетевым контроллером, как описано в следующем подразделе.

Дополнительные сведения об окружающей среде CartPole-v1 см. по следующим ссылкам:

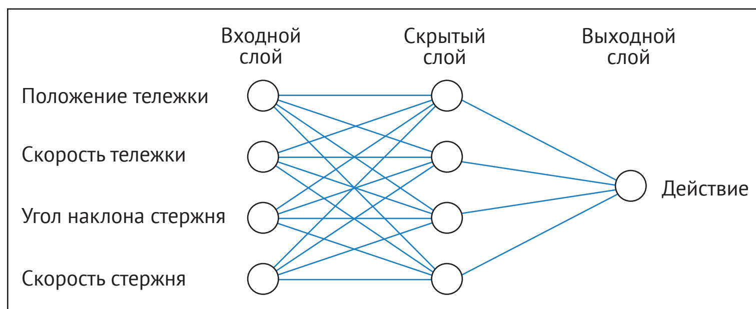
- https://gym.openai.com/envs/CartPole-v1/;
- https://github.com/openai/gym/wiki/CartPole-v0.

Управление средой CartPole с помощью нейронной сети

Чтобы решить задачу CartPole, мы должны динамически реагировать на изменения в окружающей среде. Например, если стержень начинает клониться в одном направлении, то, вероятно, нам следует толкнуть тележку в этом направлении, но перестать толкать, как только стержень начнет выправляться. Таким образом, в данном случае задачу обучения с подкреплением можно рассматривать как обучение контроллера балансированию стержня посредством отображения четырех входных переменных – положения тележки, скорости тележки, угла наклона стержня и скорости стержня – в действие на каждом шаге. Как реализовать такое отображение?

Один из способов дает нейронная сеть. В главе 9 мы видели, что нейронная сеть, например многослойный перцептрон (МСП), позволяет реализовать сложные отображения между входами и выходами. Для этого подстраиваются внутренние параметры сети – веса и смещения активных нейронов, – а также выбираются функции активации нейронов. Мы будем использовать сеть с од-

ним скрытым слоем, состоящим из четырех нейронов – блоков. Входной слой также содержит четыре блока, по одному для каждого входного значения, получаемого от окружающей среды, а выходной слой содержит всего один блок, поскольку имеется лишь одно выходное значение – действие, которое следует выполнить. Структура этой сети показана на рисунке ниже.



Структура нейронной сети для управления тележкой

Как мы уже видели, значения весов и смещений нейронной сети обычно устанавливаются в процессе ее обучения. Интересно, что до сих пор мы видели только, как такие сети обучаются с помощью алгоритма обратного распространения в процессе обучения с учителем. То есть у нас всегда был обучающий набор, содержащий входы и правильные выходы для них, а сеть обучалась так, чтобы вход отображался на соответствующий ему выход. Но в случае обучения с подкреплением таких обучающих данных нет. Насколько хорошо сработала сеть, мы узнаем только в конце эпизода. Это означает, что вместо использования традиционных алгоритмов обучения нам нужен метод, который позволит найти оптимальные параметры сети – веса и смещения, – располагая лишь результатами эпизодов взаимодействия с окружающей средой. Но именно для такого рода оптимизации генетические алгоритмы особенно хороши: найти набор параметров, дающий лучший результат, при условии что мы умеем оценивать и сравнивать результаты. Но сначала нужно решить, как представлять параметры сети и как оценивать набор параметров. Оба вопроса рассматриваются в следующем подразделе.

Представление и оценивание решения

Поскольку мы решили управлять тележкой в окружающей среде CartPole с помощью многослойного перцептрона, набор подлежащих оптимизации параметров состоит из весов и смещений сети.

- **Входной слой:** этот слой не участвует в отображении входов на выходы сетью; он просто получает входные данные и передает их каждому нейрону следующего слоя. Поэтому никаких параметров здесь нет.
- **Скрытый слой:** каждый блок этого слоя связан с каждым входом сети, поэтому требуется четыре веса и одно смещение.

- **Выходной слой:** единственный блок этого слоя связан с каждым блоком скрытого слоя, поэтому требуется четыре веса и одно смещение.

Итого, нам нужно найти 20 весов и пять смещений, все типа float. Поэтому потенциальное решение можно представить списком из 25 чисел типа float, например:

```
[0.9505049282421143, -0.8068797228337171, -0.45488246459260073, -0.7598208314027836,
... , 0.4039043861825575, -0.874433212682847, 0.9461075409693256,
0.6720551701599038]
```

Для оценки решения нужно создать МСП правильной формы – четыре входа, скрытый слой с четырьмя блоками, один выход – и сопоставить каждому узлу веса и смещения из переданного списка. Затем этот МСП используется как контроллер тележки в одном эпизоде. Результирующее полное вознаграждение становится оценкой решения. В отличие от предыдущей задачи, здесь мы стремимся *максимизировать* оценку. Вся процедура оценивания реализована в классе `CartPole`, описанном в следующем подразделе.

Представление задачи на Python

Для инкапсуляции задачи `CartPole` мы написали на Python класс `MountainCar`, код которого находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter10/cart_pole.py.

Конструктору класса передается необязательное начальное значение генератора случайных чисел. Класс предоставляет следующие методы:

- `initMlp()`: инициализирует многослойный перцептрон с требуемой архитектурой (слоями) и параметрами сети (весами и смещениями), полученными из списка чисел типа float, представляющего потенциальное решение;
- `getScore(actions)`: вычисляет оценку решения, представленного списком параметров сети. Для этого создается соответствующий МСП, после чего эпизод взаимодействия со средой CartPole инициируется и прогоняется под управлением этого МСП, выбирающего действия; при этом наблюдения играют роль входов сети. Чем больше оценка, тем лучше;
- `saveParams()`: сериализует и сохраняет список параметров сети в формате pickle;
- `replayWithSavedParams()`: десериализует последний сохраненный список параметров сети и воспроизводит эпизод методом `replay`;
- `replay()`: рисует окружающую среду и использует заданные параметры сети для воспроизведения эпизода, чтобы наглядно представить решение.

Метод `main()` следует вызывать после того, как решение сериализовано и сохранено методом `saveParams()`. Он инициализирует класс и вызывает метод `replayWithSavedParams()` для анимации последнего сохраненного решения.

Обычно метод `main()` используется для демонстрации лучшего решения, найденного генетическим алгоритмом, который описан в следующем подразделе.

Решение с помощью генетического алгоритма

Программа для решения задачи `CartPole` с помощью генетического алгоритма находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter10/02-solve-cart-pole.py.

Поскольку для представления решения мы выбрали список, содержащий числа с плавающей точкой, эта программа напоминает программы оптимизации функций, которые мы рассматривали в главе 6, например функции EggHolder.

Ниже описаны основные части программы.

1. Начинаем с создания экземпляра класса `CartPole`, который позволит нам тестировать различные решения задачи `CartPole`:

```
car = cart_pole.CartPole(RANDOM_SEED)
```

2. Далее задаем верхнюю и нижнюю границы искомых значений типа float. Поскольку значения представляют веса и смещения блоков нейронной сети, нужно выбирать их из диапазона между –1.0 и 1.0:

```
BOUNDS_LOW, BOUNDS_HIGH = -1.0, 1.0
```

3. Напомним, что наша задача состоит в том, чтобы максимизировать оценку – время, в течение которого мы сможем удерживать стержень в равновесии. Поэтому определяем единственную цель – максимизирующую стратегию приспособления:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

4. Теперь нужно написать вспомогательную функцию, которая будет порождать случайные вещественные числа, равномерно распределенные в заданном диапазоне. Эта функция предполагает, что диапазон одинаков в каждом направлении – в нашем случае так оно и есть:

```
def randomFloat(low, up):
    return [random.uniform(l, u) for l, u in zip([low] *
NUM_OF_PARAMS, [up] * NUM_OF_PARAMS)]
```

5. Воспользуемся этой функцией, чтобы создать оператор, который возвращает список случайных чисел типа float в определенном ранее диапазоне:

```
toolbox.register("attrFloat", randomFloat, BOUNDS_LOW, BOUNDS_HIGH)
```

6. Далее определим оператор, который заполняет экземпляр индивидуума с помощью предыдущего оператора:

```
toolbox.register("individualCreator",
                 tools.initIterate,
                 creator.Individual,
                 toolbox.attrFloat)
```

7. Попросим генетический алгоритм вызывать метод `getScore()` экземпляра `CartPole` для вычисления приспособленности.
   Напомним, что метод `getScore()`, описанный в предыдущем подразделе, инициирует эпизод взаимодействия с окружающей средой CartPole. В течение этого эпизода тележка управляется МСП с одним скрытым слоем. Веса и смещения этого МСП задаются списком чисел типа float, представляющим текущее решение. На протяжении этого эпизода МСП динамически отображает наблюдения за окружающей средой в перемещения: влево или вправо. По завершении эпизода оценка приспособленности равна полному вознаграждению, т. е. количеству шагов, на которых удавалось удерживать стержень в равновесии. Чем оценка выше, тем лучше.

```
def score(individual):
    return cartPole.getScore(individual),

toolbox.register("evaluate", score)
```

8. Пришло время выбрать генетические операторы. Мы снова будем использовать турнирный отбор размера 2. Поскольку решение представлено списком чисел с плавающей точкой в заданном диапазоне, то будем использовать непрерывные ограниченные операторы скрещивания и мутации из каркаса DEAP: `cxSimulatedBinaryBounded` и `mutPolynomialBounded` соответственно:

```
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0/NUM_OF_PARAMS)
```

9. Как обычно, применим элитистский подход, т. е. будем без изменения копировать лучших на данный момент индивидуумов из зала славы в следующее поколение:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                  toolbox,
                                                  cxpb=P_CROSSOVER,
                                                  mutpb=P_MUTATION,
                                                  ngen=MAX_GENERATIONS,
                                                  stats=stats,
                                                  halloffame=hof,
                                                  verbose=True)
```

10. После прогона печатаем и сохраняем лучшее решение, чтобы впоследствии его можно было анимировать с помощью средств, встроенных в класс `CartPole`:

```
best = hof.items[0]
print("Лучшее решение = ", best)
print("Лучшая оценка = ", best.fitness.values[0])
cartPole.saveParams(best)
```

11. Для окружающей среды CartPole определены критерии, по которым задача считается решенной, поэтому проверим, удовлетворяет ли им наше решение. На момент написания книги задача считалась решенной, если среднее вознаграждение больше или равно 195.0 в 100 последовательных испытаниях (см. https://github.com/openai/gym/wiki/CartPole-v0#solved-requirements).
По-видимому, это определение подразумевает, что максимальное количество шагов в одном эпизоде равно 200. Однако с тех пор длина эпизода, похоже, увеличилась до 500 шагов. Поэтому будем ориентироваться на среднее вознаграждение 490.0 в 100 последовательных эпизодах. Проверим, выполняется ли это требование, прогнав 100 тестов с использованием нашего лучшего индивидуума и усреднив результаты:

```
scores = []
for test in range(100):
    scores.append(cart_pole.CartPole().getScore(best))
print("оценки = ", scores)
print("Средняя оценка = ", sum(scores) / len(scores))
```

Заметим, что во время прогона тестов среда CartPole каждый раз инициализируется случайным образом, поэтому начальные условия в разных эпизодах несколько отличаются, потому и результаты могут быть другими.

Итак, пора посмотреть, хорошо ли мы справились с задачей. После прогона алгоритма для 10 поколений с размером популяции 20 получается такой результат:

```
gen nevals  max   avg
0   20      41    13.7
1   15      54    17.3
```

```
...
5   16      157   63.9
6   17      500   87.2
...
9   15      500   270.9
10  13      500   420.3
Лучшее решение = [0.733351790484474, -0.8068797228337171,
-0.45488246459260073, ...
Лучшая оценка = 500.0
```

Как видим, после смены всего шести поколений лучшее решение достигло максимальной оценки 500, т. е. стержень удерживался в равновесии на протяжении всего эпизода. Результаты следующего теста показывают, что во всех 100 эпизодах было получено максимальное вознаграждение 500:

```
Прогон 100 эпизодов с использованием лучшего решения...
оценки = [500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0,
500.0, 500.0, 500.0,
...
500.0, 500.0, 500.0, 500.0]
Средняя оценка = 500.0
```

Как уже было сказано, в каждом из 100 эпизодов начальные условия несколько различались. Но контроллеру неизменно удавалось удерживать стержень в равновесии. Чтобы понаблюдать за контроллером в действии, мы можем прогнать один или несколько эпизодов CartPole, воспользовавшись ранее сохраненными результатами, для чего следует запустить программу `cart_pole`. Анимация показывает, как контроллер динамически реагирует на движения стержня, применяя действия, удерживающие его в равновесии на протяжении всего эпизода.

Если хотите сравнить этот результат с неидеальным, то повторите эксперимент, заведя скрытый слой с тремя (или даже двумя) блоками вместо четырех – достаточно изменить значение константы `HIDDEN_LAYER` в классе `CartPole`.

Резюме

В этой главе мы узнали о началах обучения с подкреплением. Познакомившись с библиотекой OpenAI Gym, мы рассмотрели окружающую среду MountainCar, в которой требуется управлять автомобилем, так чтобы он поднялся на вершину холма. Мы решили эту задачу с помощью генетического алгоритма и перешли к следующей окружающей среде, CartPole, в которой требовалось удерживать в вертикальном положении стержень, шарнирно закрепленный на движущейся тележке. Мы смогли решить ее, воспользовавшись нейросетевым контроллером, который обучался под управлением генетического алгоритма.

В следующей главе мы направим стопы в мир искусства и посмотрим, как можно использовать генетические алгоритмы для реконструкции знаменитых картин из полупрозрачных перекрывающихся фигур.

Для дальнейшего чтения

За дополнительными сведениями по вопросам, рассмотренным в этой главе, рекомендуем обратиться к следующим источникам:

- Rajalingappaa Shanmugamani, Sudharsan Ravichandiran, et al. «Python Reinforcement Learning»;
- Maksim Lapan «Deep Reinforcement Learning Hands-On»;
- документация по OpenAI Gym: http://gym.openai.com/docs/;
- Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, Wojciech Zaremba «OpenAI Gym (White Paper)»: https://arxiv.org/abs/1606.01540.

ЧАСТЬ IV

РОДСТВЕННЫЕ ТЕХНОЛОГИИ

В этой части описано несколько методов оптимизации, связанных с генетическими алгоритмами и с другими бионическими комбинаторными алгоритмами

Глава 11

Генетическая реконструкция изображений

В этой главе мы будем экспериментировать с одним из самых популярных способов применения генетических алгоритмов к обработке изображений – реконструкцией изображения с помощью набора полупрозрачных многоугольников. Попутно мы приобретем полезный опыт обработки изображений и наглядно увидим, как протекает эволюционный процесс.

Мы начнем с обзора обработки изображений в Python и познакомимся с тремя полезными библиотеками: `Pillow`, `scikit-image` и `opencv-python`. Затем узнаем, как нарисовать изображение с нуля, пользуясь многоугольниками, и как вычислить степень различия двух изображений. Далее разработаем программу на базе генетического алгоритма для реконструкции участка знаменитой картины с помощью многоугольников и оценим результаты.

Будут рассмотрены следующие вопросы:

- знакомство с библиотеками обработки изображений на Python;
- как программно нарисовать изображение с помощью многоугольников;
- сравнение двух изображений;
- применение генетического алгоритма в сочетании с библиотеками обработки изображений к реконструкции изображений из многоугольников.

Мы начнем эту главу с краткого введения в задачу реконструкции изображений.

Технические требования

В этой главе нам понадобится Python 3 и следующие библиотеки:

- `deap`;
- `numpy`;
- `matplotlib`;

- `seaborn`;
- `Pillow` (ветвь PIL) – будет описана здесь же;
- `scikit-image` (`skimage`) – будет описана здесь же;
- `OpenCV-Python` (`cv2`) – будет описана здесь же.

Код приведенных в этой главе программ можно найти в репозитории книги на GitHub по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter11. Чтобы увидеть код в действии, посмотрите видео по адресу http://bit.ly/2u1ytHz.

Реконструкция изображений из многоугольников

Один из самых популярных примеров применения генетических алгоритмов в обработке изображений – реконструкция изображения с помощью набора полупрозрачных перекрывающихся фигур. Эти эксперименты не только забавны и дают возможность освоить обработку изображений, но и позволяют заглянуть внутрь эволюционного процесса и лучше понять визуальные искусства, а также последние достижения в области анализа и сжатия изображения.

В экспериментах по реконструкции – а в интернете можно найти несколько вариаций на эту тему – знакомое изображение (зачастую знаменитая картина или ее фрагмент) используется в качестве образца. Цель – составить похожее изображение из набора перекрывающихся фигур, обычно многоугольников, разного цвета и степени прозрачности.

Мы решим эту задачу с помощью генетического алгоритма и библиотеки `deap`, как уже поступали при решении многих других задач в данной книге. Но поскольку нам придется рисовать изображения и сравнивать их с образцом, давайте сначала познакомимся с основами обработки изображений на Python.

Обработка изображений на Python

Для достижения цели нужно будет выполнять различные операции с изображениями, например создавать изображение с нуля, рисовать на изображении фигуры, выводить изображение на печать, открывать файл изображения, сохранять изображение в файле, сравнивать два изображения и иногда изменять размер изображения. В следующих разделах мы посмотрим, как такие операции выполняются на Python.

Библиотеки обработки изображений на Python

Из всего богатства библиотек обработки изображений, доступных программистам на Python, мы выбрали три самые известные.

Библиотека Pillow

`Pillow` – сопровождаемая в настоящее время ветвь оригинальной библиотеки **Python Imaging Library** (**PIL**). Она позволяет открывать файлы изображений разного формата, манипулировать ими и сохранять в файлах. Поскольку она умеет рисовать фигуры, задавать степень прозрачности и манипулировать пикселями, мы выбрали ее в качестве основного инструмента создания реконструированного изображения.

Адрес домашней страницы этой библиотеки – https://python-pillow.org/.

Обычно для установки `Pillow` используется команда `pip`:

```
pip install Pillow
```

В `Pillow` применяется пространство имен `PIL`. Если на вашем компьютере уже установлена оригинальная библиотека PIL, то сначала ее следует удалить. Дополнительные сведения можно найти в документации по адресу https://pillow.readthedocs.io/en/stable/index.html.

Библиотека scikit-image

Библиотека `scikit-image`, разработанная сообществом SciPy, расширяет возможности `scipy.image` и предлагает различные алгоритмы обработки изображений, в т. ч. ввод-вывод, фильтрацию, работу с цветами и распознавание объектов. Нам здесь понадобится только модуль `metrics`, содержащий средства сравнения двух изображений.

Адрес домашней страницы этой библиотеки – https://scikit-image.org/.

Библиотека `scikit-image` входит в состав нескольких дистрибутивов Python, в частности *Anaconda* и *winPython*. Для ее установки в общем случае можно воспользоваться командой

```
pip install scikit-image
```

Но если вы пользуетесь дистрибутивом Anaconda или miniconda, то выполните такую команду:

```
conda install -c conda-forge scikit-image
```

Дополнительные сведения можно найти в документации по адресу https://scikit-image.org/docs/stable/index.html.

Библиотека opencv-python

OpenCV – весьма развитая библиотека, содержащая различные алгоритмы, относящиеся к компьютерному зрению и машинному обучению. Ее версии существуют для многих языков программирования и платформ. Библиотека `opencv-python` – интерфейс между OpenCV и Python. Она сочетает скорость C++ API с удобством языка Python. Здесь мы будем пользоваться ей в основном для вычисления разности двух изображений.

Адрес домашней страницы библиотеки `opencv-python` – https://pypi.org/project/opencv-python/.

Библиотека состоит из четырех пакетов с общим пространством имен cv2. В одной среде следует устанавливать лишь один из этих пакетов. Для наших целей достаточно следующей команды, которая устанавливает только основные модули:

```
pip install opencv-python
```

Дополнительные сведения можно найти в документации по адресу https://docs.opencv.org/master/.

Рисование с помощью многоугольников

Чтобы нарисовать изображение с нуля, можно воспользоваться классами Pillow `Image` и `ImageDraw`, например:

```
image = Image.new('RGB', (width, height))
draw = ImageDraw.Draw(image, 'RGBA')
```

Здесь `'RGB'` и `'RGBA'` – значения аргумента **mode**. `'RGB'` означает, что каждый пиксель представляется тремя 8-битовыми значениями – по одному для основных цветов: красный (‘R’), зеленый (‘G’) и синий (‘B’). `'RGBA'` добавляет еще одно 8-битовое значение ‘A’, представляющее альфа-канал (уровень непрозрачности). Комбинация базового RGB-изображения и рисования в формате RGBA позволяет рисовать многоугольники разной степени прозрачности на черном фоне.

Мы можем добавить в базовое изображение многоугольник, вызвав метод `polygon` класса `ImageDraw`, как показано в следующем примере, где рисуется треугольник:

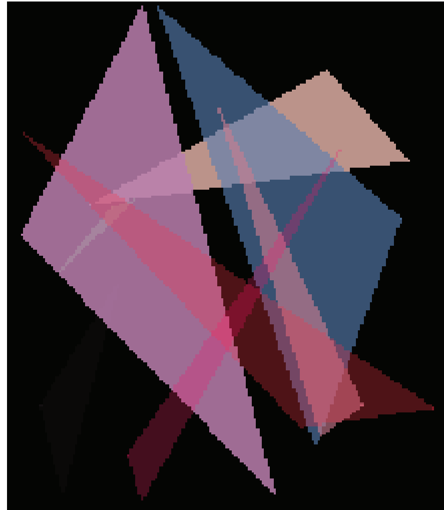
```
draw.polygon([(x1, y1), (x2, y2), (x3, y3)], (red, green, blue, alpha))
```

Ниже этот пример описывается подробно.

- Кортежи `(x1, y1)`, `(x2, y2)` и `(x3, y3)` представляют три вершины треугольника. Каждый кортеж содержит координаты *x*, *y* одной вершины.
- `red`, `green` и `blue` – целые числа в диапазоне [0, 255], представляющие яркость соответствующей компоненты цвета многоугольника.
- `alpha` – целое число в диапазоне [0, 255], представляющее степень непрозрачности многоугольника (чем меньше значение, тем прозрачнее цвет).

Чтобы нарисовать многоугольник с большим числом вершин, нужно добавить дополнительные кортежи (x_i, y_i) в список.

Можно было бы добавить и больше многоугольников, частично перекрывающихся, как показано на рисунке ниже.



Перекрывающиеся многоугольники
разного цвета и прозрачности

Нарисовав изображение, составленное из многоугольников, мы должны сравнить его с образцом. Как это делается, описано в следующем подразделе.

Измерение степени различия двух изображений

Мы хотим построить изображение, похожее на оригинальное, поэтому необходим способ вычисления степени сходства или различия двух изображений. Существует два таких способа:

- попиксельная среднеквадратическая ошибка (СКО);
- структурное сходство (Structural Similarity – SSIM).

Ниже рассматриваются оба.

Попиксельная среднеквадратическая ошибка

Самый распространенный способ оценки сходства двух изображений – попиксельное сравнение. Разумеется, для этого изображения должны быть одинакового размера.

Метрика СКО вычисляется следующим образом.

1. Вычислить квадрат разности между каждой парой соответственных пикселей изображений. Поскольку пиксель представлен значениями трех компонент – красной, зеленой и синей, – то разности считаются по каждому измерению в отдельности.
2. Сложить все квадраты.
3. Разделить сумму на общее число пикселей.

Если оба изображения представлены средствами библиотеки OpenCV (cv2), то это вычисление выполняется следующим образом:

```
MSE = np.sum((cv2Image1.astype("float") - cv2Image2.astype("float")) ** 2)/
float(numPixels)
```

Ели оба изображения одинаковы, то СКО равна нулю. Поэтому в качестве целевой функции алгоритма можно принять минимизацию этой метрики.

Структурное сходство (SSIM)

Индекс структурного сходства используется для предсказания качества изображения, созданного данным алгоритмом сжатия, посредством его сравнения с исходным. В отличие от СКО, метод SSIM вычисляет не абсолютную величину ошибки, а основан на восприятии и учитывает изменения структурной информации, а также таких свойств изображения, как яркость и текстура.

В модуле `metrics` из библиотеки `scikit-image` имеется функция, вычисляющая индекс структурного сходства двух изображений. Если оба изображения представлены в формате библиотеки OpenCV (cv2), то эту функцию можно вызвать напрямую:

```
SSIM = structural_similarity(cv2Image1, cv2Image2)
```

Она возвращает число с плавающей точкой в диапазоне [–1, 1], это и есть индекс структурного сходства. Значение 1 означает, что оба изображения одинаковы.

По умолчанию функция сравнивает полутоновые изображения. Для сравнения цветных изображений нужно задать необязательный аргумент `multichannel` равным `True`.

Применение генетических алгоритмов для реконструкции изображений

Как уже было сказано, цель нашего эксперимента – взять знакомое изображение в качестве образца и создать другое изображение, максимально похожее на оригинал, составив его из перекрывающихся многоугольников разного цвета и прозрачности. При использовании генетического алгоритма каждое потенциальное решение представляет собой набор таких многоугольников, а для оценки решения нужно составить из них изображение и сравнить его с образцом. Как обычно, первым делом нужно определиться с представлением решения.

Представление и оценивание решения

Итак, наше решение состоит из набора многоугольников в границах изображения. У каждого многоугольника есть цвет и степень прозрачности. Для рисования многоугольника средствами библиотеки `Pillow` нужно задать следующие аргументы:

- список кортежей $[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]$, представляющий вершины многоугольника. Каждый кортеж содержит координаты x, y одной вершины в системе координат изображения. Поэтому значения x принадлежат диапазону [0, ширина изображения – 1], а значения y – диапазону [0, высота изображения – 1];
- три целых числа в диапазоне [0, 255], представляющих красную, зеленую и синюю компоненты цвета многоугольника;
- еще одно целое число в диапазоне [0, 255], представляющее альфа-канал, или степень непрозрачности, многоугольника.

Это значит, что для каждого многоугольника в наборе нужно [2×(количество вершин) + 4] параметров. Например, для треугольника понадобится 10 параметров, а для шестиугольника – 16 параметров. Поэтому набор треугольников будет представлен списком следующего формата, в котором под каждый треугольник отведено 10 параметров:

$[x_{11}, y_{11}, x_{12}, y, x_{13}, y_{13}, r_1, g_1, b_1, alpha_1, x_{21}, y_{21}, x_{22}, y_{22}, x_{23}, y_{23}, r2, g_2, b_2, alpha_2, \ldots]$

Чтобы упростить это представление, мы будем использовать для каждого параметра числа с плавающей точкой в диапазоне [0, 1]. Но перед тем как рисовать многоугольник, мы приведем каждый параметр к нужному диапазону.

При таком представлении набор из 50 треугольников будет описываться списком из 500 чисел с плавающей точкой от 0 до 1, например:

```
[0.1499488467959301, 0.3812631075049196, 0.0004394580562993053,
0.9988170920722447, 0.9975357316889601, 0.9997461395379549,
0.6338072268312986, 0.379170095245514, 0.29280945382368373,
0.20126488596803083,
...
0.4551462922205506, 0.9912529573649455, 0.7882252614083617,
0.01684396868069853, 0.2353587486989349, 0.003221988752732261,
0.9998952203500615, 0.48148512088979356, 0.11555604920908047,
0.08328550982740457]
```

Для оценивания такого решения нужно разбить этот длинный список на части, соответствующие отдельным многоугольникам; в случае треугольников длина каждой части равна 10. Затем мы создадим новое изображение и нарисуем на нем все многоугольники один за другим. И наконец, вычислим степень различия между получившимся изображением и оригиналом. В пре-

дыдущем разделе мы описали два способа вычисления разности изображений – попиксельная СКО и индекс структурного сходства. Эта (довольно длинная) процедура вычисления оценки реализована классом Python, который мы опишем в следующем подразделе.

Представление задачи на Python

Класс, инкапсулирующий реконструкцию изображения, называется `ImageTest` и находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter11/image_test.py.

Конструктору класса передается два параметра: путь к файлу, содержащему изображение-образец, и количество вершин многоугольников, из которых составляется изображение-реконструкция. Класс предоставляет следующие открытые методы:

- `polygonDataToImage()`: принимает список, содержащий данные многоугольников, рассмотренный в предыдущем подразделе, разбивает этот список на части, представляющие отдельные многоугольники, и создает изображение, рисуя эти многоугольники на чистом холсте;
- `getDifference()`: принимает данные многоугольников, создает из них изображение и вычисляет различие между ним и образцом одним из двух методов: СКО или SSIM;
- `plotImages()`: для сравнения размещает образец и построенное изображение на одном рисунке;
- `saveImage()`: принимает данные многоугольников, создает из них изображение, создает рисунок, содержащий образец и построенное изображение, и сохраняет этот рисунок в файле.

В процессе выполнения генетического алгоритма метод `saveImage()` будет вызываться через каждые 100 поколений, чтобы можно было следить за ходом реконструкции. Для вызова метода служит функция обратного вызова, описанная в следующем подразделе.

Реализация генетического алгоритма

Для применения генетического алгоритма к реконструкции изображения с помощью набора полупрозрачных перекрывающихся прямоугольников мы написали на Python программу, которая находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter11/reconstruct-with-polygons.py.

Поскольку мы представляем решение списком чисел с плавающей точкой, эта программа очень похожа на программы для оптимизации функций из главы 6, например функции Eggholder.

Ниже описаны основные части программы.

1. Сначала определим константы. `POLYGON_SIZE` задает число вершин многоугольников, а `NUM_OF_POLYGONS` – общее число многоугольников, составляющих реконструированное изображение:

```
POLYGON_SIZE = 3
NUM_OF_POLYGONS = 100
```

2. Создадим экземпляр класса `ImageTest`, с помощью которого будем создавать изображения из многоугольников, сравнивать их с образцом и сохранять ход процесса:

```
imageTest = image_test.ImageTest("images/Mona_Lisa_Head.png", POLYGON_SIZE)
```

3. Зададим верхнюю и нижнюю границы диапазона чисел с плавающей точкой. Как было сказано выше, мы будем для удобства представлять все параметры числами из одного и того же диапазона от 0.0 до 1.0. Перед оценкой решения эти значения будут приведены к истинным диапазонам параметров и преобразованы в целые числа:

```
BOUNDS_LOW, BOUNDS_HIGH = 0.0, 1.0
```

4. Поскольку мы стремимся минимизировать разность изображений – образца и построенного из многоугольников, – определим единственную цель – минимизирующую стратегию приспособления:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

5. Теперь напишем вспомогательную функцию, которая порождает случайные вещественные числа, равномерно распределенные в заданном диапазоне. Предполагается, что диапазон один и тот же по каждому измерению, как в нашем случае:

```
def randomFloat(low, up):
    return [random.uniform(l, u) for l, u in zip([low] *
NUM_OF_PARAMS, [up] * NUM_OF_PARAMS)]
```

6. Воспользуемся этой функцией для создания оператора, возвращающего список случайных чисел типа float, каждое в диапазоне [0, 1]:

```
toolbox.register("attrFloat", randomFloat, BOUNDS_LOW, BOUNDS_HIGH)
```

7. Далее определим оператор, который заполняет экземпляр индивидуума, вызывая определенный выше оператор:

```
toolbox.register("individualCreator",
                 tools.initIterate,
                 creator.Individual,
                 toolbox.attrFloat)
```

8. Затем просим генетический алгоритм вызвать метод `getDifference()` экземпляра `ImageTest` для вычисления приспособленности.
Напомним, что определенный в предыдущем подразделе метод `getDifference()` принимает индивидуума, представляющего список многоугольников, создает из этих многоугольников изображение и вычисляет степень различия между ним и образцом, применяя один из двух методов: СКО или SSIM. Сначала воспользуемся методом СКО:

```
def getDiff(individual):
    return imageTest.getDifference(individual, "MSE"),

toolbox.register("evaluate", getDiff)
```

9. Пора заняться генетическими операторами. В качестве оператора отбора используем турнир размера 2. В главе 4 мы видели, что эта схема хорошо работает в сочетании с элитистским подходом, который мы планируем применить и здесь.

```
toolbox.register("select", tools.selTournament, tournsize=2)
```

10. Что же касается операторов скрещивания и мутации, то поскольку наше решение представлено списком чисел с плавающей точкой в ограниченном диапазоне, будем использовать специализированные для такого случая операторы из каркаса DEAP – `cxSimulatedBinaryBounded` и `mutPolynomialBounded` соответственно, с которыми впервые встретились в главе 6.

```
toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0/NUM_OF_PARAMS)
```

11. Как и много раз прежде, будем использовать элитистский подход, т. е. без изменения копировать лучших на данный момент индивидуумов из зала славы в следующее поколение. Но на этот раз добавим еще один нюанс – функцию обратного вызова, которую будем вызывать через каждые 100 поколений. Смысл этой функции мы обсудим в следующем подразделе:

```
population, logbook =
elitism_callback.eaSimpleWithElitismAndCallback(population,
                                                toolbox,
                                                cxpb=P_CROSSOVER,
                                                mutpb=P_MUTATION,
                                                ngen=MAX_GENERATIONS,
                                                callback=saveImage,
                                                stats=stats,
                                                halloffame=hof,
                                                verbose=True)
```

12. В конце программы печатаем лучшее решение и рисуем созданное ей изображение бок о бок с образцом:

```
best = hof.items[0]
print("Лучшее решение = ", best)
print("Лучшая оценка = ", best.fitness.values[0])
imageTest.plotImages(imageTest.polygonDataToImage(best))
```

Прежде чем знакомиться с результатами, обсудим реализацию функции обратного вызова.

Добавление функции обратного вызова в код генетического алгоритма

Чтобы получить возможность сохранять текущее изображение через каждые 100 поколений, главный цикл генетического алгоритма нужно немного модифицировать. Как вы помните, в конце главы 4 мы уже внесли в главный цикл одно изменение с целью реализовать механизм элитизма. Для этого был написан метод `eaSimpleWithElitism()` и помещен в файл `elitism.py`. За основу мы взяли метод `eaSimple()` из каркаса DEAP, который находится в файле `algorithms.py`, и добавили в него функциональность элитизма – выбрать лучших на текущий момент индивидуумов из зала славы и без изменения копировать их в следующее поколение на каждой итерации цикла. Теперь, чтобы реализовать обратный вызов, мы внесем еще одно небольшое изменение и назовем новый метод `eaSimpleWithElitismAndCallback()`. И поместим его в файл `elitism_callback.py`.

Модификация состоит из двух частей.

1. Первым делом добавим новый аргумент метода – `callback`:

   ```
   def eaSimpleWithElitismAndCallback(population, toolbox, cxpb,
                                      mutpb, ngen, callback=None,
                                      stats=None, halloffame=None,
                                      verbose=__debug__):
   ```

   Он представляет внешнюю функцию, вызываемую после каждой итерации.
2. Далее изменим сам главный цикл. Будем вызывать функцию обратного вызова после создания и оценки нового поколения. В качестве аргументов ей передаются номер текущего поколения и лучший на данный момент индивидуум:

   ```
   if callback:
       callback(gen, halloffame.items[0])
   ```

Определение функции обратного вызова, вызываемой после каждого поколения, может оказаться полезным в разных ситуациях. В данном случае мы определим функцию `saveImage()` в программе `01-reconstruct-with-polygons.py` и воспользуемся ей для сохранения рисунка, содержащего лучшее на данный момент изображение и образец, через каждые 100 поколений:

1. Выполнить некоторое действие через каждые 100 поколений нам поможет оператор вычисления остатка (%):

   ```
   if gen  % 100 == 0:
   ```

2. Если это условие выполнено, то мы создаем папку для хранения изображений, если она еще не существует. В имени папки участвуют число вершин многоугольника и число многоугольников, например `run-3-100` или `run-6-50`, а сама она создается внутри папки `images/results/`:

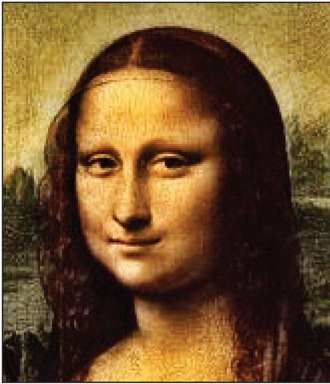
```
folder = "images/results/run-{}-{}".format(POLYGON_SIZE, NUM_OF_POLYGONS)
if not os.path.exists(folder):
    os.makedirs(folder)
```

3. Сохраним изображение, соответствующее лучшему индивидууму, в этой папке. Имя изображения включает количество уже сменившихся поколений, например `after-300-generations.png`:

```
imageTest.saveImage(polygonData,
                    "{}/after-{}-gen.png".format(folder, gen),
                    "После {} поколений".format(gen))
```

Вот теперь мы готовы выполнить алгоритм, задав изображение-образец, и посмотреть на результат.

Результаты реконструкции изображения



Голова Моны Лизы

Источник: https://commons.wikimedia.org/wiki/File:Mona_Lisa_headcrop.jpg. Художник: Леонардо да Винчи. Публикуется по лицензии Creative Commons CC0 1.0: https://creativecommons.org/publicdomain/zero/1.0/

Для тестирования программы возьмем знаменитый портрет Моны Лизы кисти Леонардо да Винчи, считающийся самой известной картиной в мире.

Для реконструкции изображения возьмем 100 треугольников:

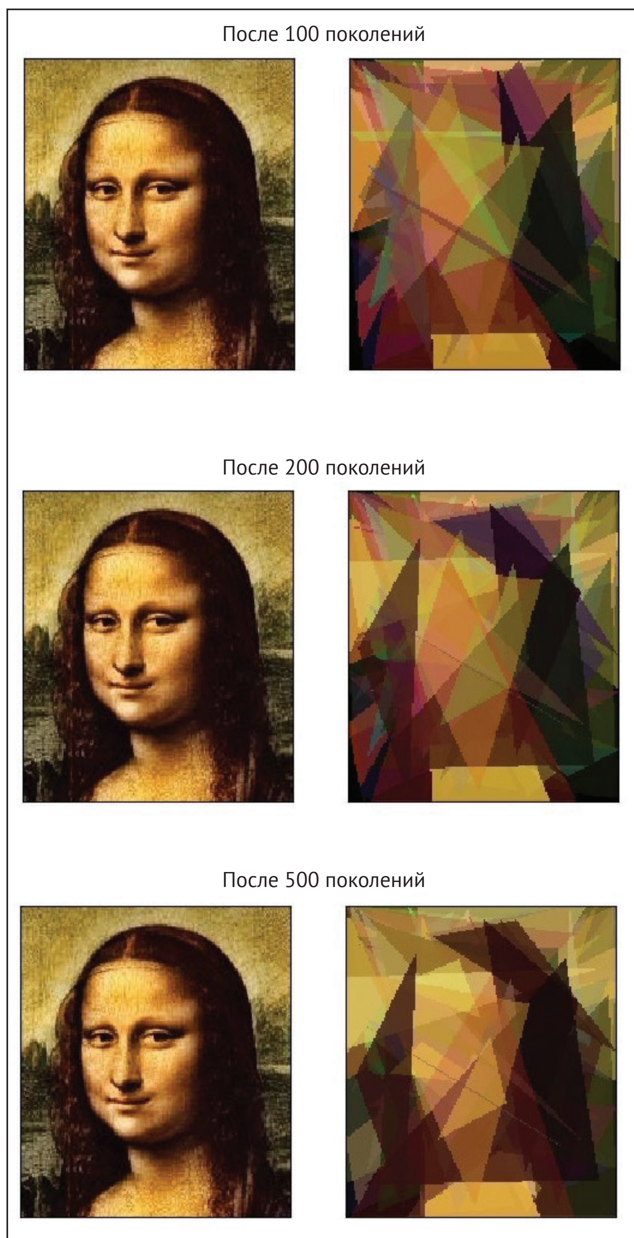
```
POLYGON_SIZE = 3
NUM_OF_POLYGONS = 100
```

Прогоним алгоритм на 5000 поколений в популяции размера 200. Результаты сравнения будем сохранять после каждых 100 поколений. По завершении прогона просмотрим сохраненные изображения, чтобы понаблюдать за эволюцией реконструируемого изображения.

Хотим предупредить, что из-за объема данных о многоугольниках и сложности операций обработки изображений время работы этого генетического алгоритма гораздо больше, чем для других программ, встречавшихся в данной книге, и в типичном случае составляет несколько часов. Результаты эксперимента описаны в следующих подразделах.

Применение попиксельной среднеквадратической ошибки

Начнем с применения попиксельной СКО для измерения разности между образцом и реконструированным изображением. Вот несколько промежуточных результатов:



Промежуточные результаты реконструкции портрета Моны Лизы с применением попиксельной среднеквадратической ошибки – часть 1

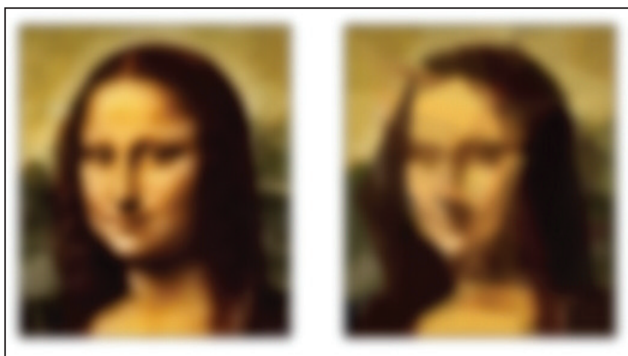
А вот еще несколько результатов:

Промежуточные результаты реконструкции портрета Моны Лизы
с применением попиксельной среднеквадратической ошибки – часть 2

Конечный результат похож на оригинал, хотя содержит острые углы и прямые линии – но этого и следовало ожидать от изображения, составленного из многоугольников. Если смотреть на изображения, прищурившись, то эти дефекты сглаживаются. Аналогичного эффекта можно достичь и программно, воспользовавшись фильтром `GaussianBlur` из библиотеки OpenCV:

```
origImage = cv2.imread('path/to/image')
blurredImage = cv2.GaussianBlur(origImage, (45, 45), cv2.BORDER_DEFAULT)
```

Вот как выглядят размытые версии обоих изображений, помещенные бок о бок:



Размытые версии оригинала и реконструированного изображения
(с применением попиксельной СКО)

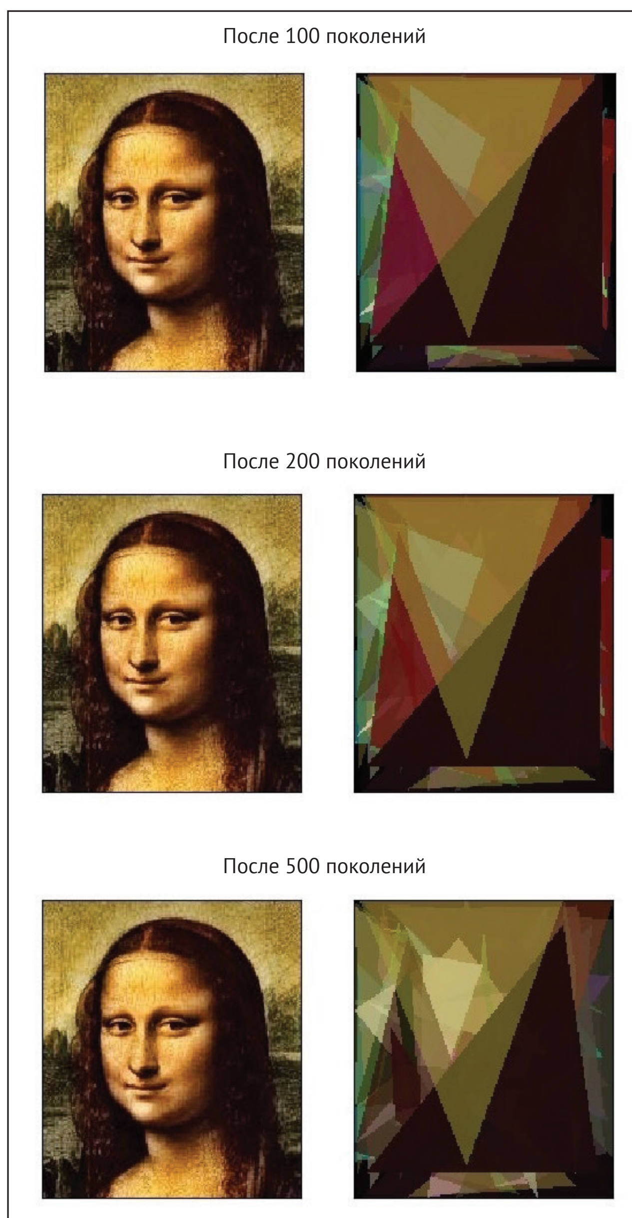
Далее испытаем другой метод измерения разности двух изображений – индекс структурного сходства.

Применение индекса структурного сходства

Повторим эксперимент, но на этот раз воспользуемся индексом структурного сходства (SSIM) для измерения степени различия между образцом и реконструированным изображением. Для этого модифицируем определение `getDiff()` следующим образом:

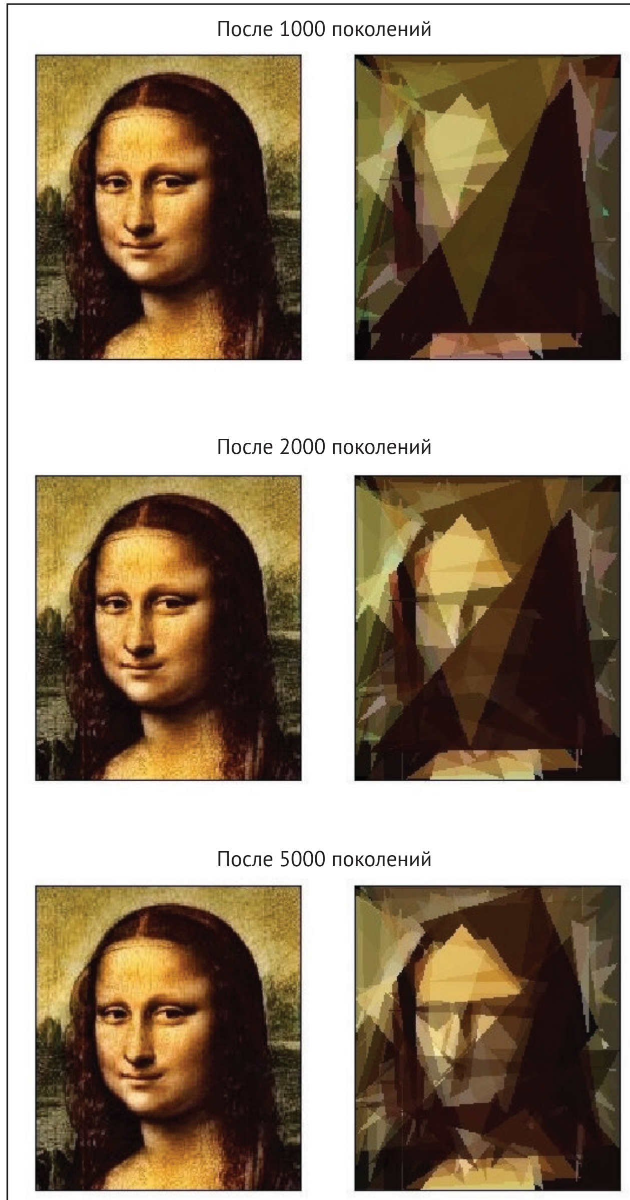
```
def getDiff(individual):
    return imageTest.getDifference(individual, "SSIM"),
```

Этот эксперимент принес такие промежуточные результаты.



Промежуточные результаты реконструкции портрета Моны Лизы с применением индекса структурного сходства – часть 1

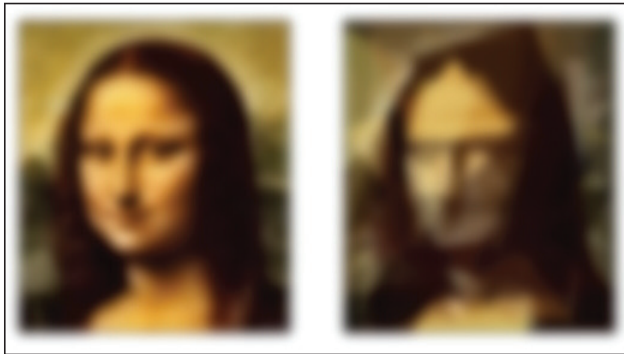
А вот еще результаты для следующих поколений:



Промежуточные результаты реконструкции портрета Моны Лизы
с применением индекса структурного сходства – часть 2

Результат любопытный – алгоритм уловил структуру изображения, но не так точно, как в случае СКО. Цвета немного искажены, потому что метод SSIM

уделяет больше внимания структуре и текстуре. Вот как выглядят размытые изображения, помещенные рядом:



Размытые версии оригинала и реконструированного изображения
(с применением индекса структурного сходства)

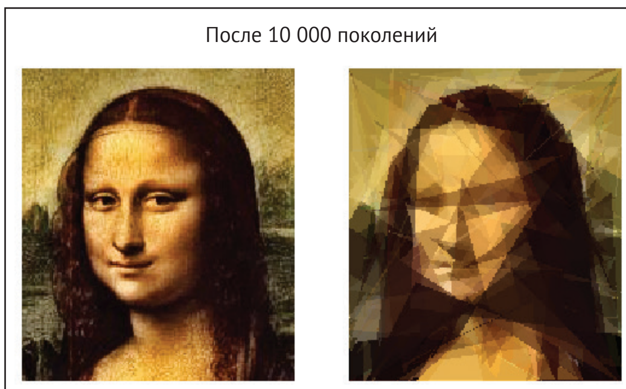
Интересно было бы скомбинировать оба метода измерения различия – сделайте это самостоятельно. Другие идеи для экспериментов описаны в следующем подразделе.

Другие эксперименты

Есть масса иных предложений. Простая и очевидная вариация – увеличить число вершин многоугольников. Можно ожидать, что результат получится точнее, потому что чем больше вершин, тем больше гибкость. Попробуем задать шесть вершин:

```
POLYGON_SIZE = 6
```

Повторив эксперимент с 10 000 поколений, получим такой результат:



Результаты реконструкции
с применением попиксельной СКО и шестиугольников

Как и следовало ожидать, изображение несколько точнее, чем в случае треугольников. Ниже приведены размытые изображения.

Размытые версии оригинала и реконструированного изображения
(с применением попиксельной СКО и шестиугольников)

Помимо изменения числа вершин, можно поэкспериментировать с другими возможностями и комбинациями, например:

- изменить общее число фигур;
- изменить размер популяции и количество поколений;
- использовать фигуры, отличные от многоугольников (круги, эллипсы) или только правильные фигуры (квадраты, прямоугольники);
- использовать разные типы исходных изображений (картины, рисунки, фотографии, логотипы);
- использовать не цветные, а полутоновые изображения.

Удачи в творческих экспериментах!

Резюме

В этой главе мы познакомились с популярной идеей реконструкции изображений с помощью перекрывающихся полупрозрачных многоугольников. Затем обсудили несколько библиотек обработки изображений на Python и узнали, как можно нарисовать изображение, составленное из многоугольников, и как вычислить степень различия двух изображений. Далее мы разработали генетический алгоритм для реконструкции участка знаменитой картины с помощью многоугольников. Были упомянуты многочисленные направления дальнейших экспериментов.

В следующей главе мы опишем и продемонстрируем несколько подходов к решению задач, родственных генетическим алгоритмам, а также некоторые другие бионические вычислительные алгоритмы.

Для дальнейшего чтения

За дополнительными сведениями по вопросам, рассмотренным в этой главе, рекомендуем обратиться к следующим источникам:

- Grow Your Own Picture: https://chriscummins.cc/s/genetics/;
- Genetic Programming: Evolution of Mona Lisa: https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa/;
- Sandipan Dey «Hands-On Image Processing with Python»
- Wang Z., Bovik A. C., Sheikh H. R. & Simoncelli E. P. (2004) «Image quality assessment: From error visibility to structural similarity», *IEEE Transactions on Image Processing*, 13, 600-612: https://ece.uwaterloo.ca/~z70wang/publications/.

Глава 12

Другие эволюционные и бионические методы вычислений

В этой главе мы расширим горизонты и откроем еще несколько методов решения задач и оптимизации, родственных генетическим алгоритмам. Два представителя этого расширенного семейства – генетическое программирование и оптимизация методом роя частиц – будут продемонстрированы конкретными программами на Python. И напоследок мы приведем краткий обзор ряда других вычислительных парадигм.

Будут рассмотрены следующие вопросы:

- семейство эволюционных вычислительных алгоритмов;
- идеи генетического программирования и его отличие от генетических алгоритмов;
- применение генетического программирования к решению задачи о контроле по четности;
- применение метода роя частиц к оптимизации функции Химмельблау;
- принципы некоторых других эволюционных и бионических алгоритмов.

Мы начнем эту главу с описания расширенного семейства эволюционных вычислений и обсуждения основных характеристик, присущих всем его членам.

Технические требования

В этой главе нам понадобится Python 3 и следующие библиотеки:

- `deap`;
- `numpy`;
- `networkx`.

Код приведенных в этой главе программ можно найти в репозитории книги на GitHub по адресу https://github.com/PacktPublishing/Hands-On-Genetic-

Algorithms-with-Python/tree/master/Chapter12. Чтобы увидеть код в действии, посмотрите видео по адресу http://bit.ly/2Gx4KsL.

Эволюционные и бионические вычисления

В этой книге мы рассмотрели методы решения задач, получившие названия генетических алгоритмов, и применили их к таким задачам, как комбинаторная оптимизация, оптимизация с ограничениями и оптимизация непрерывных функций, а также к машинному обучению и искусственному интеллекту. Однако, как было отмечено в главе 1, генетические алгоритмы – лишь часть гораздо более широкого семейства алгоритмов *эволюционных вычислений*, включающего различные методы решения задач и оптимизации, истоки которых следует искать в теории естественной эволюции Чарльза Дарвина.

Перечислим основные черты, общие для всех этих методов:

- отправной точкой является начальный набор потенциальных решений (популяция);
- потенциальные решения (индивидуумы) итеративно обновляются с целью создания новых поколений;
- при создании нового поколения менее успешные индивидуумы отсеиваются (отбор), а некоторые индивидуумы подвергаются небольшим случайным изменениям (мутациям). Могут также применяться иные операторы, например взаимодействие с другими индивидуумами (скрещивание);
- в результате смены поколений возрастает **приспособленность** популяции. Иными словами, потенциальные решения лучше проявляют себя в решении поставленной задачи.

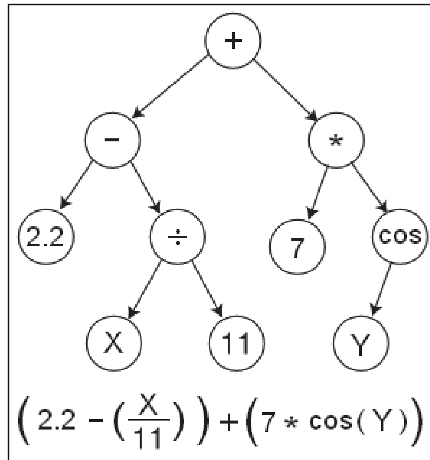
Говоря шире, поскольку эволюционные вычисления основаны на поведении различных биологических систем, некоторые из них пересекаются с семейством алгоритмов, известным под названием *бионические вычисления*.

В следующих разделах мы рассмотрим некоторые из наиболее часто используемых представителей эволюционных и бионических вычислений – одни более детально, другие лишь кратко упомянем. Начнем с подробного рассказа об удивительной технике эволюции самих компьютерных программ – *генетическом программировании*.

Генетическое программирование

Генетическое программирование (**ГП**) – это специальный вид генетического алгоритма – метода, который мы рассматривали на протяжении всей книги. В этом частном случае потенциальными решениями, или индивидуумами, эволюционирующими с целью нахождения лучшего решения, являются сами компьютерные программы, отсюда и название. Иными словами, применяя ГП, мы изменяем программы, стремясь найти идеально подходящую для решения данной задачи.

Как вы помните, в генетических алгоритмах используется представление потенциальных решений, называемое хромосомой. К этому представлению применяются генетические операторы отбора, скрещивания и мутации. Применение этих операторов к текущему поколению решений дает новое поколение, которое предположительно должно быть лучше предыдущего. В большинстве рассмотренных выше задач представлением был список (или массив) значений определенного типа: целых, булевых или с плавающей точкой. Но для представления программы обычно используется древовидная структура, показанная на рисунке ниже.



Древовидное представление простой программы

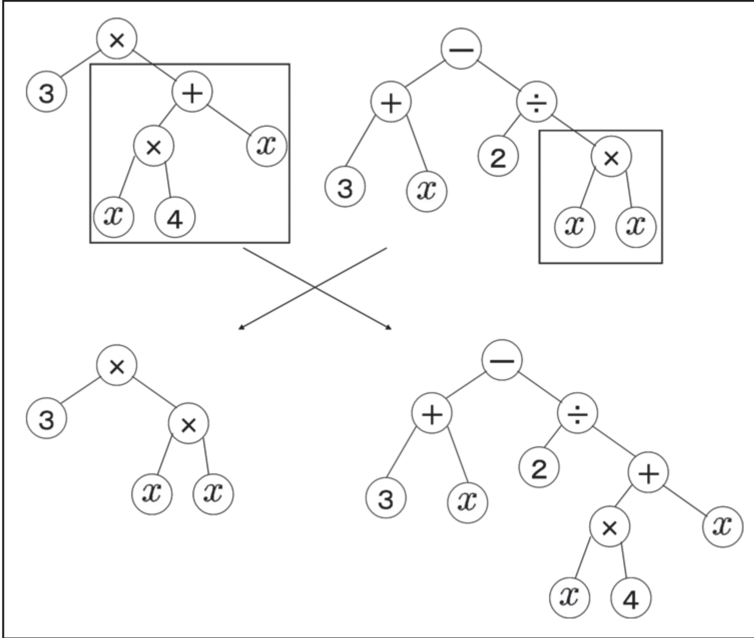
Источник: https://commons.wikimedia.org/wiki/File:Genetic_Program_Tree.png.
Автор Baxelrod. Является общественным достоянием

Эта конкретная древовидная структура представляет вычисление, приведенное под деревом. Оно эквивалентно короткой программе (или функции), принимающей два аргумента, X и Y, и возвращающей результат вычислений с ними. Чтобы создать и подвергнуть эволюции такие древовидные структуры, нам придется определить два множества:

- **терминальные символы**, или листья дерева. Это аргументы функции и константы, встречающиеся в дереве. В нашем примере аргументами являются X и Y, а константами – 2.2, 11 и 7. Константы можно также генерировать случайно, выбирая из некоторого диапазона, в процессе создания дерева;
- **примитивы**, или внутренние узлы дерева. Это функции (или операторы), принимающие один или несколько аргументов и порождающие один результат. В нашем примере +, - и × – примитивы, принимающие два аргумента, а *cos* – примитив с одним аргументом.

В главе 2 мы показывали, как генетический оператор одноточечного скрещивания применяется к спискам двоичных значений. Он создает двух потомков двух родителей, вырезая из каждого родителя часть и переставляя

эти части местами. Аналогично оператор скрещивания для деревьев может отделить некоторое поддерево (ветвь или группу ветвей) каждого родителя и поменять эти поддеревья местами, как показано на рисунке ниже.



Операция скрещивания двух древовидных структур, представляющих программы

Источник: https://commons.wikimedia.org/wiki/File:GP_crossover.png.
Автор: U-ichi, публикуется по лицензии Creative Commons CC SA 1.0: https://creativecommons.org/licenses/sa/1.0/

В этом примере два родителя в верхней строке обменялись частями, в результате чего получились два потомка в нижней строке. Переставленные части обведены прямоугольной рамкой.

Аналогично оператор мутации, который вносит случайные изменения в индивидуума, можно реализовать, выбрав какое-нибудь поддерево потенциального решения и заменив его случайно сгенерированным.

Библиотека `deap`, которой мы пользовались всю дорогу, предоставляет поддержку генетического программирования. В следующем разделе мы реализуем простой пример такого рода.

Пример генетического программирования – контроль по четности

Мы воспользуемся генетическим программированием, чтобы создать программу контроля по четности. В этой задаче на вход могут подаваться значе-

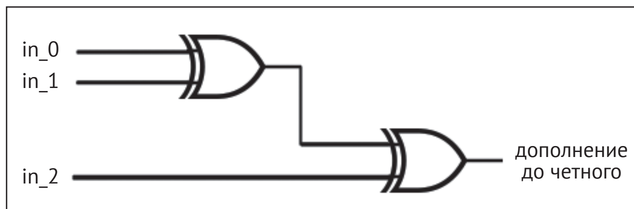
ния 0 и 1. На выходе должна выдаваться 1, если количество единиц на входе было нечетным (таким образом, общее количество единиц станет четным), в противном случае 0. В таблице ниже показаны различные комбинации входных значений и результат для каждой из них.

in_0	in_1	in_2	Бит четности
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Контроль по четности можно также представить с помощью логических вентилей: AND, OR, NOT, XOR (исключающее OR). Вентиль NOT принимает один вход и инвертирует его, а все остальные вентили принимают два входа. Чтобы на выходе получилась 1, на входы вентиля AND нужно подать две единицы, на входы OR – хотя бы одну единицу, а на входы XOR – ровно одну единицу. Это иллюстрируется в следующей таблице.

in_0	in_1	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Существует много способов реализовать контроль по четности для трех входов с помощью логических вентилей. Проще всего взять два вентиля XOR, как показано на следующем рисунке:



Контроль по четности для трех входов,
реализованный двумя вентилями XOR

В следующем разделе мы применим генетическое программирование для создания небольшой программы, которая реализует контроль по четности с помощью логических операций AND, OR, NOT и XOR.

Реализация с помощью генетического программирования

Для эволюции программы, реализующей логику контроля по четности, мы написали на Python программу, которая находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter12/01-gp-even-parity.py.

Поскольку генетическое программирование – частный случай генетического алгоритма, многое в этой программе покажется вам знакомым, если вы прорабатывали программы из предыдущих глав.

Ниже описаны основные части программы.

1. Для начала зададим значения констант. NUM_INPUTS – количество входов программы контроля по четности. Для простоты возьмем значение 3, хотя никто не мешает увеличить его. NUM_COMBINATIONS – количество возможных комбинаций значений входов, т. е. число строк в показанной выше таблице истинности:

```
NUM_INPUTS = 3
NUM_COMBINATIONS = 2 ** NUM_INPUTS
```

2. Далее следуют уже хорошо знакомые константы генетического алгоритма:

```
POPULATION_SIZE = 60
P_CROSSOVER = 0.9
P_MUTATION = 0.5
MAX_GENERATIONS = 20
HALL_OF_FAME_SIZE = 10
```

3. Однако для генетического программирования нужно еще несколько констант, относящихся к представлению потенциальных решений в виде дерева. Они определены ниже, а как используются, мы увидим далее:

```
MIN_TREE_HEIGHT = 3
MAX_TREE_HEIGHT = 5
MUT_MIN_TREE_HEIGHT = 0
MUT_MAX_TREE_HEIGHT = 2
LIMIT_TREE_HEIGHT = 17
```

4. Затем вычисляется таблица истинности для контроля по четности; мы будем использовать ее как образец при проверке правильности потенциального решения. Матрица `parityIn` представляет входные столбцы таблицы истинности, а вектор `parityOut` – выходной столбец. Функция Python `itertools.product()` служит элегантной заменой вложенным циклам `for` и позволяет обойти все входные значения:

```
parityIn = list(itertools.product([0, 1], repeat=NUM_INPUTS))
parityOut = []
```

```
for row in parityIn:
    parityOut.append(sum(row)  % 2)
```

5. Приспело время создать набор примитивов, т. е. операторов, которые будут использоваться в эволюционирующей программе. В первом объявлении создается пустой набор функций, для чего указываются следующие три аргумента:
   - имя программы, генерируемой с помощью примитивов из этого набора (мы назвали ее `main`);
   - количество входных аргументов программы;
   - префикс имен входных аргументов (необязательно).

   И вот как создается набор примитивов:

```
primitiveSet = gp.PrimitiveSet("main", NUM_INPUTS, "in_")
```

6. Теперь заполним набор примитивов различными функциями (или операторами), из которых будет строиться программа. Для каждого оператора задается ссылка на функцию, которую мы будем использовать, и количество передаваемых ей аргументов. Хотя мы могли бы определить собственные функции, в данном случае обойдемся теми, что имеются в модуле Python `operator`, где среди прочего есть и нужные нам логические операторы:

```
primitiveSet.addPrimitive(operator.and_, 2)
primitiveSet.addPrimitive(operator.or_, 2)
primitiveSet.addPrimitive(operator.xor, 2)
primitiveSet.addPrimitive(operator.not_, 1)
```

7. В следующих определениях задаются терминальные символы. Мы уже говорили, что это константы, которые можно использовать как входные данные для дерева. В нашем случае имеет смысл взять значения 0 и 1:

```
primitiveSet.addTerminal(1)
primitiveSet.addTerminal(0)
```

8. Поскольку мы стремимся создать программу, которая реализует таблицу истинности для контроля по четности, то надо будет минимизировать разность между выходом программы и известными значениями. Для этого определим единственную цель – минимизирующую стратегию приспособления:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

9. Далее создадим класс `Individual` на основе класса `PrimitiveTree` из библиотеки `deap`:

```
creator.create("Individual", gp.PrimitiveTree,
fitness=creator.FitnessMin)
```

10. Для конструирования индивидуумов напишем вспомогательную функцию, которая будет генерировать случайные деревья, пользуясь при-

митивами из ранее определенного набора. В данном случае мы воспользуемся функцией genFull() из библиотеки deap, передав ей набор примитивов, а также значения минимальной и максимальной высоты генерируемых деревьев:

```
toolbox.register("expr", gp.genFull, pset=primitiveSet,
min_=MIN_TREE_HEIGHT, max_=MAX_TREE_HEIGHT)
```

11. Затем определим два оператора: один создает индивидуума с помощью только что определенной вспомогательной функции, а другой порождает список таких индивидуумов:

```
toolbox.register("individualCreator", tools.initIterate,
creator.Individual, toolbox.expr)
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

12. Следующий оператор компилирует дерево примитивов в Python-код, вызывая функцию compile() из библиотеки deap. Впоследствии мы будем использовать этот оператор компиляции в созданной нами функции parityError(), подсчитывающей количество строк в таблице истинности, для которых результат вычислений отличается от ожидаемого:

```
toolbox.register("compile", gp.compile, pset=primitiveSet)

def parityError(individual):
    func = toolbox.compile(expr=individual)
    return sum(func(*pIn) != pOut for pIn, pOut in zip(parityIn, parityOut))
```

13. Сообщаем алгоритму генетического программирования, что для вычисления приспособленности следует использовать функцию getCost(). Эта функция возвращает ошибку четности в форме кортежа, необходимого эволюционному алгоритму:

```
def getCost(individual):
    return parityError(individual),

toolbox.register("evaluate", getCost)
```

14. Пора уже выбрать генетические операторы. В качестве оператора отбора (с псевдонимом select) в генетическом программировании обычно используется турнирный отбор, с которым мы много раз встречались в этой книге. В данном случае зададим размер турнира 2:

```
toolbox.register("select", tools.selTournament, tournsize=2)
```

15. В качестве оператора скрещивания (с псевдонимом mate) возьмем специализированный для генетического программирования оператор cxOnePoint(), предоставляемый библиотекой deap. Поскольку эволюционирующие программы представлены в виде деревьев, этот оператор принимает два родительских дерева и переставляет местами их участки, создавая два дерева-потомка:

```
toolbox.register("mate", gp.cxOnePoint)
```

16. Оператор мутации вносит случайные изменения в существующее дерево. Он определяется в два этапа. Сначала определим вспомогательный оператор, пользующийся специальной функцией генетического программирования `genGrow()` из библиотеки `deap`. Этот оператор создает поддерево в рамках ограничений, заданных двумя константами. Затем определим сам оператор мутации, который заменяет случайно выбранное поддерево случайным же деревом, сгенерированным вспомогательным оператором:

```
toolbox.register("expr_mut", gp.genGrow,
min_=MUT_MIN_TREE_HEIGHT, max_=MUT_MAX_TREE_HEIGHT)
toolbox.register("mutate", gp.mutUniform,
expr=toolbox.expr_mut, pset=primitiveSet)
```

17. Чтобы предотвратить разрастание индивидуумов в деревья с чрезмерным числом примитивом, нужны меры контроля разбухания. Для этого служит функция DEAP `staticLimit()`, которая налагает ограничение на высоту дерева, получающегося в результате скрещивания и мутации:

```
toolbox.decorate("mate",
gp.staticLimit(key=operator.attrgetter("height"),
max_value=LIMIT_TREE_HEIGHT))
toolbox.decorate("mutate",
gp.staticLimit(key=operator.attrgetter("height"),
max_value=LIMIT_TREE_HEIGHT))
```

18. Главный цикл программы очень похож на те, что встречались нам в предыдущих главах. После создания начальной популяции, определения статистических показателей и создания зала славы мы вызываем эволюционный алгоритм. Как и прежде, применяем элитистский подход, т. е. без изменения копируем лучших на данный момент индивидуумов из зала славы в следующее поколение:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                  toolbox,
                                                  cxpb=P_CROSSOVER,
                                                  mutpb=P_MUTATION,
                                                  ngen=MAX_GENERATIONS,
                                                  stats=stats,
                                                  halloffame=hof,
                                                  verbose=True)
```

19. В конце работы печатаем лучшее решение, а также высоту представляющего его дерева и его длину – общее количество операторов в дереве:

```
best = hof.items[0]
print("--Лучший индивидуум = ", best)
print("-- длина={}, высота={}".format(len(best), best.height))
print("--Лучшая приспособленность = ", best.fitness.values[0])
```

20. Осталось только нарисовать дерево, представляющее лучшее решение. Для этого мы воспользуемся библиотекой для работы с графами и сетями **networkx** (**nx**), с которой познакомились в главе 5. Сначала вызовем функцию `graph()` из библиотеки `deap`, которая расчленяет дерево индивидуума на вершины, ребра и метки, необходимые для создания графа, а затем создадим этот граф, вызывая соответствующие функции из библиотеки `networkx`:

```
nodes, edges, labels = gp.graph(best)
g = nx.Graph()
g.add_nodes_from(nodes)
g.add_edges_from(edges)
pos = nx.spring_layout(g)
```

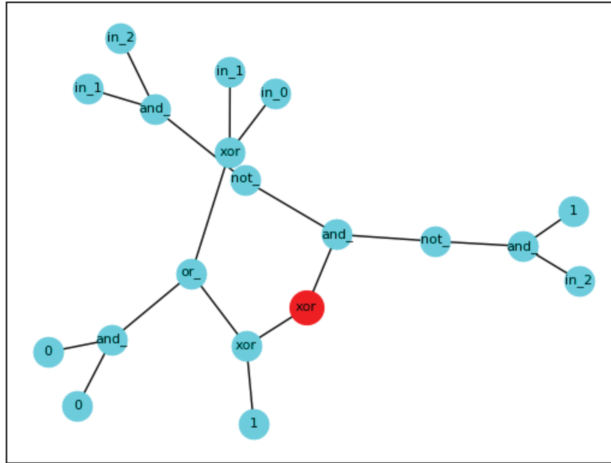
21. Далее мы рисуем вершины, ребра и метки. Поскольку внешне граф не похож на классическое дерево иерархии, выделяем корень красным цветом и увеличенным размером:

```
nx.draw_networkx_nodes(g, pos, node_color='cyan')
nx.draw_networkx_nodes(g, pos, nodelist=[0], node_color='red',
node_size=400)
nx.draw_networkx_edges(g, pos)
nx.draw_networkx_labels(g, pos, **{"labels": labels,
"font_size": 8})
```

В результате работы программы получаем:

```
gen nevals min avg
0   60     2   3.91667
1   50     1   3.75
2   47     1   3.45
...
5   47     0   3.15
...
20  48     0   1.68333
-- Лучший индивидуум = xor(and_(not_(and_(in_1, in_2)), not_(and_(1, in_2))),
xor(or_(xor(in_1, in_0), and_(0, 0)), 1))
-- длина=19, высота=4
-- Лучшая приспособленность = 0.0
```

Это простая задача, поэтому приспособленность быстро достигла минимального значения 0, т. е. мы нашли решение, которое строит правильную таблицу истинности для контроля по четности. Однако получившееся выражение состоит из 19 элементов и четырех уровней иерархии – слишком сложно. Оно показано на рисунке ниже, созданном программой:



Решение задачи о контроле по четности,
найденное первой версией программы

Как было сказано выше, красная вершина графа соответствует корню дерева программы, т. е. первой операции XOR в выражении.

Такой относительно сложный граф получился, потому что мы не ставили целью использовать более простые выражения. При условии что ограничение на высоту дерева удовлетворяется, за сложность выражения не начислялось никаких штрафов. В следующем подразделе мы попытаемся исправить ситуацию, немного модифицировав программу в надежде получить желаемый результат – реализацию контроля по четности – ценой более простого выражения.

Упрощение решения

В приведенной выше реализации мы предприняли меры для ограничения размера деревьев, представляющих потенциальные решения. Но лучшее решение оказалось чрезмерно сложным. Один из способов понудить алгоритм порождать более простые решения – ввести небольшой штраф за сложность. Но штраф должен быть достаточно малым, чтобы алгоритм не отдавал предпочтения слишком простым выражениям, не решающим задачу. Скорее, он должен служить для разрешения конфликта между двумя хорошими решениями и выбирать из них более простое. Этот подход реализован в программе, находящейся по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter12/02-gp-even-parity-reduced.py.

Эта программа отличается от предыдущей всего в двух местах.

1. Прежде всего изменяется функция стоимости, которую алгоритм стремится минимизировать. К исходной ошибке добавляется небольшой штраф, зависящий от высоты дерева:

```
def getCost(individual):
    return parityError(individual) + individual.height / 100,
```

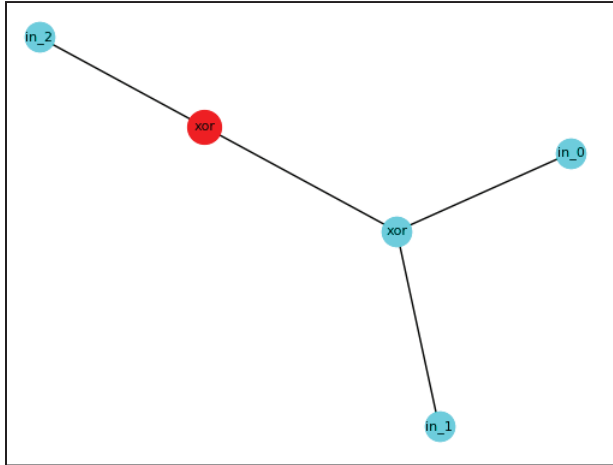
2. А второе изменение находится в конце программы – после печати лучшего найденного решения. Мы добавили печать самой полученной ошибки, без штрафа:

```
print("-- Лучшая ошибка четности = ", parityError(best))
```

Модифицированная версия дает такой результат:

```
gen nevals min  avg
0   60     2.03 3.9565
1   50     2.03 3.7885
...
5   47     0.04 3.45233
...
10  48     0.03 3.0145
...
15  49     0.02 2.57983
...
20  45     0.02 2.88533
-- Лучший индивидуум = xor(xor(in_0, in_1), in_2)
-- длина=5, высота=2
-- Лучшая приспособленность = 0.02
-- Лучшая ошибка четности = 0
```

Как видим, уже после пяти поколений алгоритм нашел решение, точно повторяющее искомую таблицу истинности, поскольку значение приспособленности для него почти не отличалось от нуля. Однако алгоритм продолжил работу, и высота дерева уменьшилась с четырех (штраф равен 0.04) до двух (штраф равен 0.02). Итоговое лучшее решение оказалось очень простым, состоящим всего из пяти элементов – три входа и два оператора XOR. На самом деле мы нашли лучшее известное решение, которое приводили раньше, – состоящее из двух вентилей XOR. Оно показано на рисунке, созданном программой.



Решение задачи о контроле по четности,
найденное модифицированной версией программы

В следующем разделе мы рассмотрим еще один бионический популяционный алгоритм. Однако его идея отличается от знакомых нам генетических алгоритмов с их операторами отбора, скрещивания и мутации. Вместо них для модификации популяции в каждом поколении применяются другие правила – добро пожаловать в мир роевых алгоритмов.

Оптимизация методом роя частиц

Идея **оптимизации методом роя частиц** (particle swarm optimization – **PSO**) подсмотрена у природных сообществ отдельных организмов – стай птиц, косяков рыб и т. п., обычно называемых роями. Организмы взаимодействуют в рое без какой-либо централизованной координации, но совместно достигают общей цели. Это наблюдаемое поведение положило начало вычислительному методу, который может решить задачу или выполнить оптимизацию, пользуясь группой потенциальных решений, представленных частицами, – по аналогии с организмами в рое. Частицы перемещаются в пространстве поиска, отыскивая лучшее решение, а их перемещение подчиняется простым правилам, формулируемым в терминах положения и скорости (векторной).

Алгоритм PSO итеративный, на каждой итерации вычисляется положение каждой частицы и обновляются ее лучшее на текущий момент положение, а также положение лучшей частицы во всей группе. Затем обновляются скорости всех частиц на основе следующей информации:

- текущая скорость и направление движения частицы – представляют инерцию;
- текущее лучшее положение (локально лучшее) – представляет когнитивную силу;

- положение лучшей частицы во всей группе (глобально лучшее) – представляет социальную силу.

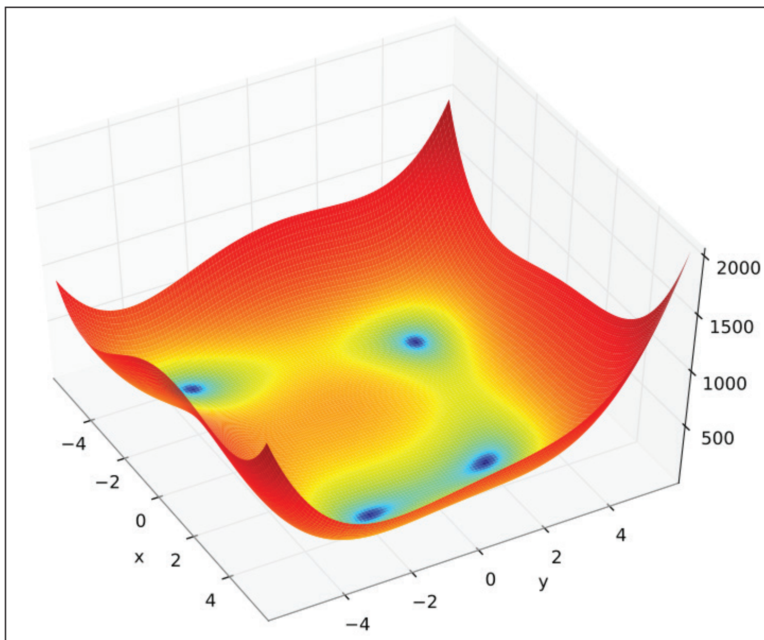
Затем производится обновление положения частиц, исходя из пересчитанной скорости.

Процесс продолжается до выполнения условия остановки, например достижения предельного числа итераций. Наилучшее положение группы на данный момент считается решением, найденным алгоритмом.

Этот простой, но эффективный процесс подробно иллюстрируется в следующем разделе, где мы напишем программу, которая будет оптимизировать функцию с помощью алгоритма PSO.

Пример применения PSO – оптимизация функции

Для демонстрации воспользуемся алгоритмом PSO, чтобы найти положение минимума (или минимумов) тестовой функции Химмельблау, которую мы уже оптимизировали в главе 6 с помощью генетического алгоритма. Эта функция изображена на рисунке ниже.



Функция Химмельблау

Источник: https://commons.wikimedia.org/wiki/File:Himmelblau_function.svg.
Автор: Morn the Gorn. Является общественным достоянием

Напомним, что математически функция описывается формулой

$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$.

У нее четыре глобальных минимума, равных 0. Они отмечены синим цветом и находятся в точках:

- $x = 3.0$, $y = 2.0$;
- $x = -2.805118$, $y = 3.131312$;
- $x = -3.779310$, $y = -3.283186$;
- $x = 3.584458$, $y = -1.848126$.

Мы будем искать только один из этих минимумов.

Реализация оптимизации методом роя частиц

Для нахождения минимума функции Химмельблау методом роя частиц мы написали на Python программу, которая находится в файле по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter12/03-pso-himmelblau.py.

Ниже описаны основные части программы.

1. Для начала зададим значения различных констант. Размерность задачи в нашем случае равна 2, а она, в свою очередь, определяет размерность положения и скорости частиц. Следующая константа – размер популяции, т. е. количество частиц в рое, и, наконец, количество поколений, или итераций алгоритма.

```
DIMENSIONS = 2
POPULATION_SIZE = 20
MAX_GENERATIONS = 500
```

2. Далее зададим несколько дополнительных констант, влияющих на порядок создания и обновления частиц. Их роль станет ясна, когда мы будем рассматривать код программы:

```
MIN_START_POSITION, MAX_START_POSITION = -5, 5
MIN_SPEED, MAX_SPEED = -3, 3
MAX_LOCAL_UPDATE_FACTOR = MAX_GLOBAL_UPDATE_FACTOR = 2.0
```

3. Поскольку мы стремимся найти минимум функции Химмельблау, определим единственную цель – минимизирующую стратегию приспособления:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

4. Теперь создадим класс `Particle`. Поскольку этот класс представляет положение в непрерывном пространстве, мы могли бы унаследовать его от обычного списка чисел с плавающей точкой. Однако здесь мы решили воспользоваться N-мерным массивом (`ndarray`) из библиотеки `numpy`, так как он особенно удобен для выполнения поэлементных алгебраических операций, в частности сложения и умножения, которые понадобятся при обновлении положения частицы. Помимо текущего положения, методу создания объекта класса `Particle` передается несколько дополнительных атрибутов:

- fitness – определенная выше минимизирующая стратегия приспособления;
- speed – здесь будут храниться компоненты текущего вектора скорости частицы. Начальное значение равно None, но позже скорость будет инициализирована с помощью другого массива ndarray;
- best – представляет лучшее найденное на данный момент положение этой частицы (локально лучшее).

В итоге определение метода создания объекта класса Particle выглядит следующим образом:

```
creator.create("Particle", np.ndarray,
fitness=creator.FitnessMin, speed=None, best=None)
```

5. Для конструирования одной частицы нам понадобится вспомогательная функция, которая создает частицу и инициализирует ее случайным образом. Мы воспользуемся функцией random.uniform() из библиотеки numpy, чтобы сгенерировать случайное положение и скорость частицы в заданных пределах:

```
def createParticle():
    particle = creator.Particle(np.random.uniform(
                                    MIN_START_POSITION,
                                    MAX_START_POSITION,
                                    DIMENSIONS))
    particle.speed = np.random.uniform(MIN_SPEED, MAX_SPEED,
                                    DIMENSIONS)
    return particle
```

6. Эта функция используется в определении оператора, который создает экземпляр частицы, а он, в свою очередь, применяется в операторе создания популяции:

```
toolbox.register("particleCreator", createParticle)
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.particleCreator)
```

7. Далее определим сердце алгоритма – метод updateParticle(), отвечающий за обновление положения и скорости каждой частицы в популяции. Ему передается одна частица и лучшее найденное на данный момент положение.
Первым делом метод создает два случайных множителя – для локального и глобального обновлений – в заранее заданном диапазоне. Затем он вычисляет два обновления скорости (локальное и глобальное) и прибавляет их к текущей скорости частицы.
Заметим, что все участвующие в вычислениях значения имеют тип массива ndarray, в нашем случае двумерного, а вычисления производятся поэлементно.
Обновленная скорость частицы – это комбинация ее исходной скорости (представляющей инерцию), ее лучшего известного положения

(когнитивная сила) и лучшего известного положения частицы во всей популяции (социальная сила):

```
def updateParticle(particle, best):

    localUpdateFactor = np.random.uniform(0,
                              MAX_LOCAL_UPDATE_FACTOR,
                              particle.size)
    globalUpdateFactor = np.random.uniform(0,
                              MAX_GLOBAL_UPDATE_FACTOR,
                              particle.size)

    localSpeedUpdate = localUpdateFactor * (particle.best - particle)
    globalSpeedUpdate = globalUpdateFactor * (best - particle)

    particle.speed = particle.speed + (localSpeedUpdate + lobalSpeedUpdate)
```

8. Метод `updateParticle()` далее проверяет, что новая скорость не выходит за установленные пределы, и обновляет положение частицы с учетом пересчитанной скорости. Выше мы отмечали, что положение и скорость имеют тип `ndarray`, содержащий по одному элементу в каждом измерении:

```
particle.speed = np.clip(particle.speed, MIN_SPEED, MAX_SPEED)
particle[:] = particle + particle.speed
```

9. Теперь зарегистрируем метод `updateParticle()` как оператор инструментария, который будет впоследствии использоваться в главном цикле:

```
toolbox.register("update", updateParticle)
```

10. Нам еще нужно определить подлежащую оптимизации функцию Химмельблау и зарегистрировать ее как оператор вычисления приспособленности:

```
def himmelblau(particle):
    x = particle[0]
    y = particle[1]
    f = (x ** 2 + y - 11) ** 2 + (x + y ** 2 - 7) ** 2
    return f, # return a tuple

toolbox.register("evaluate", himmelblau)
```

11. Вот мы, наконец, и добрались до метода `main()`, где можем начать с создания популяции частиц:

```
population = toolbox.populationCreator(n=POPULATION_SIZE)
```

12. Прежде чем входить в главный цикл, нужно создать объект `stats`, необходимый для вычисления статистики, и объект `logbook`, в котором будет сохраняться статистика на каждой итерации:

```
stats = tools.Statistics(lambda ind: ind.fitness.values)
```

```
stats.register("min", np.min)
stats.register("avg", np.mean)

logbook = tools.Logbook()
logbook.header = ["gen", "evals"] + stats.fields
```

13. Главный цикл программы содержит внешний цикл по поколениям, в котором производится обновление частиц. На каждой итерации имеется два вспомогательных цикла, в каждом из которых обходятся все частицы в популяции. В первом цикле, показанном ниже, к каждой частице применяется оптимизируемая функция и при необходимости обновляются локально и глобально лучшие частицы:

```
particle.fitness.values = toolbox.evaluate(particle)

# локально лучшая
if particle.best is None or particle.best.size == 0 or
particle.best.fitness < particle.fitness:
    particle.best = creator.Particle(particle)
    particle.best.fitness.values = particle.fitness.values

# глобально лучшая
if best is None or best.size == 0 or best.fitness < particle.fitness:
   best = creator.Particle(particle)
   best.fitness.values = particle.fitness.values
```

14. Во втором внутреннем цикле вызывается оператор `update`. Ранее мы видели, что этот оператор обновляет скорость и положение частицы на основе комбинации инерции, когнитивной силы и социальной силы:

```
toolbox.update(particle, best)
```

15. В конце внешнего цикла мы сохраняем и печатаем статистику для текущего поколения:

```
logbook.record(gen=generation, evals=len(population),
**stats.compile(population))
print(logbook.stream)
```

16. По завершении внешнего цикла печатается информация о лучшем положении, найденном за все время работы алгоритма. Оно и считается решением задачи:

```
# печать информации о лучшем найденном решении
print("-- Лучшая частица = ", best)
print("-- Лучшая приспособленность = ", best.fitness.values[0])
```

Программа печатает следующий результат:

```
gen evals min      avg
0   20    8.74399  167.468
```

```
1   20   19.0871   357.577
2   20   32.4961   219.132
...
497 20    7.2162   412.189
498 20    6.87945  273.712
499 20   16.1034   272.385

-- Лучшая частица = [-3.77695478 -3.28649153]
-- Лучшая приспособленность = 0.0010248367255068806
```

Как видим, алгоритм смог найти один минимум рядом с точкой $x = -3.77$, $y = -3.28$. Статистика показывает, что лучший результат был достигнут в поколении 480. Также видно, что частицы колеблются, то приближаясь к лучшему результату, то отдаляясь от него.

Для нахождения других минимумов можно перезапустить алгоритм с иным начальным значением генератора случайных чисел. Можно также штрафовать решения в областях поблизости от уже найденных минимумов, как мы делали при оптимизации функции Симионеску в главе 6. Еще один подход – использовать несколько роев для нахождения всех минимумов в одном прогоне; попробуйте реализовать эту идею самостоятельно (см. также раздел «Для дальнейшего чтения»).

В следующем разделе мы кратко рассмотрим еще несколько алгоритмов из семейства эволюционных вычислений.

Другие родственные методы

Помимо уже рассмотренных, существует немало других методов решения задач и оптимизации, черпающих вдохновение из дарвиновской теории эволюции, а также из других биологических систем и видов поведения. Ниже описаны некоторые из них.

Эволюционные стратегии

Эволюционные стратегии (ЭС) – разновидность генетического алгоритма, в которой упор делается на мутацию, а не на скрещивание как движущую силу эволюции. Мутация адаптивна, ее сила изменяется вместе со сменой поколений. Оператор отбора в эволюционной стратегии всегда основан на ранге, а не на фактическом значении приспособленности. Простой вариант этого метода называется (1 + 1). В нем всего два индивидуума – родитель и его мутировавший потомок. Лучший из них становится родителем следующего мутировавшего потомка. В более общем случае (1 + λ) имеется один родитель и λ мутировавших потомков, и лучший потомок становится родителем следующих λ потомков. Существуют также варианты алгоритма с несколькими родителями и оператором скрещивания.

Дифференциальная эволюция

Дифференциальная эволюция (ДЭ) – частный случай генетического алгоритма, предназначенный для оптимизации вещественных функций. ДЭ отличается от генетического алгоритма в следующих отношениях:

- популяция в ДЭ всегда представляется множеством вещественных векторов;
- вместо замены всего текущего поколения новым алгоритм ДЭ продолжает обходить популяцию, модифицируя по одному индивидууму за раз или оставляя исходного, если он лучше модифицированного;
- традиционные операторы скрещивания и мутации заменяются специализированными, которые модифицируют значение текущего индивидуума, исходя из значений трех других случайно выбранных индивидуумов.

Муравьиный алгоритм оптимизации

Идея **муравьиных алгоритмов оптимизации** (ant colony optimization – **ACO**) подсказана способом поиска пищи некоторыми видами насекомых. Муравьи начинают рыскать случайным образом, и когда один находит пищу, он возвращается в муравейник, помечая свой путь феромонами. Другие муравьи, нашедшие пищу в том же месте, усиливают след, оставляя собственные феромоны. Со временем феромоновые метки выветриваются, поэтому более короткие пути и пути, по которым прошло больше муравьев, имеют преимущество.

В муравьиных алгоритмах используются искусственные муравьи, которые обследуют пространство поиска, стремясь найти лучшие решения. Муравьи запоминают, где были и какие решения нашли по пути. Эта информация используется муравьями на следующих итерациях для поиска еще лучших решений. Часто такие алгоритмы сочетаются с методами локального поиска, которые подключаются, когда найдена перспективная область.

Искусственные иммунные системы

Идея искусственных иммунных систем (artificial immune systems – AIS) почерпнута из характеристик адаптивных иммунных систем млекопитающих. Они способны распознавать новые угрозы, обучаться противодействию, а также применять полученные знания, чтобы быстрее реагировать при следующем появлении похожей угрозы.

В последнее время AIS находят применение в различных задачах машинного обучения и оптимизации. Можно выделить три основных направления.

- **Клональная селекция.** Имитация процесса, с помощью которого иммунная система выбирает лучшую клетку для распознавания и уничтожения антигена, проникшего в тело клетки. Клетка выбирается из пула уже существующих клеток различной специфичности, а после выбора

клонируется для создания популяции клеток, уничтожающих проникший антиген. Эта идея обычно применяется в задачах оптимизации и распознавания образов.
- **Негативная селекция.** Это аналог процесса, с помощью которого организм определяет и уничтожает клетки, которые могут атаковать его собственные ткани. Такие алгоритмы часто применяются в задачах обнаружения аномалий, в которых нормальные паттерны используются для «негативного» обучения фильтров, способных обнаруживать аномальные паттерны.
- **Алгоритмы иммунных сетей.** В их основе лежит теория о том, что иммунная система регулируется с помощью антител специального типа, которые прикрепляются к другим антителам. В таких алгоритмах антитела представляют вершины графа сети, а процесс обучения заключается в создании или удалении ребер между вершинами, в результате чего эволюционирует структура графа. Обычно они применяются в задачах машинного обучения без учителя, а также в задачах управления и оптимизации.

Искусственная жизнь

Искусственная жизнь – не ответвление эволюционных вычислений, а более широкая дисциплина, занимающаяся изучением систем и процессов, которые так или иначе имитируют естественную жизнь. Примерами могут служить компьютерное моделирование и робототехнические системы.

Эволюционные вычисления можно рассматривать как применение искусственной жизни, когда популяция, стремящаяся оптимизировать некоторую функцию приспособленности, является метафорой организма в поисках пищи. Например, механизмы образования ниш и разделения ресурсов, описанные в главе 2, напрямую вытекают из метафоры пищи.

Ниже перечислены основные разделы теории искусственной жизни.

- **программная:** программное (цифровое) моделирование;
- **аппаратная:** аппаратные (физические) роботы;
- **биологическая:** биохимические манипуляции и синтетическая биология.

Искусственную жизнь можно рассматривать как восходящий аналог искусственного интеллекта, поскольку в ее основе лежат биологические окружающие среды, механизмы и структуры, а не высокоуровневые познавательные способности.

Резюме

В этой главе мы познакомились с расширенным семейством эволюционных вычислений и некоторыми общими характеристиками его представителей. Затем применили генетическое программирование – частный случай гене-

тических алгоритмов – к решению задачи контроля по четности. После этого мы написали программу, в которой метод роя частиц использовался для нахождения минимумов функции Химмельблау. И закончили главу кратким обзором еще нескольких родственных методов решения задач.

Книга подошла к концу, и я хотел бы поблагодарить всех, кто вместе со мной отправился в это путешествие по различным аспектам и применениям генетических алгоритмов и эволюционных вычислений. Надеюсь, что книга показалась вам интересной и дала пищу для размышлений. Как было показано, генетические алгоритмы и родственные им методы применимы к самым разным задачам практически в любых областях вычислений и техники, в т. ч., весьма возможно, и в тех, в которых подвизаетесь вы сами. Напомним, что для применения генетического алгоритма нужен лишь способ представить решение и получить его численную оценку – или хотя бы сравнить два решения. Мы живем в век искусственного интеллекта и облачных вычислений, а генетические алгоритмы хорошо приспособлены к тому и другому и могут стать мощным оружием в вашем арсенале, когда придется решать очередную задачу.

Для дальнейшего чтения

За дополнительными сведениями по вопросам, рассмотренным в этой главе, рекомендуем обратиться к следующим источникам:

- генетическое программирование: бионическое машинное обучение http://geneticprogramming.com/tutorial/;
- Jannes Klaas «Machine Learning for Finance»;
- Manish Kumar and Anand Deshpande «Artificial Intelligence for Big Data»;
- многомодальная оптимизация методом роя частиц: конкурс CEC 2015 по многонишевой оптимизации с одной целевой функцией: https://ieeexplore.ieee.org/document/7257009.

Предметный указатель

A

AdaboostClassifier, ссылка, 195

C

CartPole, окружающая среда
  входной слой, 231
  выходной слой, 232
  оценивание решения, 231
  представление задачи на Python, 232
  представление решения, 231
  решение с помощью генетического алгоритма, 233
  скрытый слой, 231
  ссылка, 230
  управление с помощью нейронной сети, 230
creator, модуль, 62

D

DEAP
  введение, 61
  выполнение программы, 75
  подготовка, 69
  решение задачи OneMax, 67
  эволюция решения, 72

E

Eggholder, функция
  оптимизация, 154
  оптимизация с помощью генетического алгоритма, 155
  повышение скорости сходимости, 158
env, интерфейс, 222
  ссылка, 223

F

Fitness, класс, 62

I

Iris, набор данных, 209

K

k-точечное скрещивание, 44

M

MountainCar, окружающая среда
  оценивание решения, 225
  представление задачи на Python, 226
  представление решения, 225
  решение с помощью генетического алгоритма, 226
MountainCar-v0, ссылки, 225

N

networkx(nx), библиотека, 268
NQueensProblem, класс, 128

O

OpenAI Gym, библиотека, 221
opencv-python, библиотека, 241

P

Pillow, библиотека, 241

S

scikit-image, библиотека, 241

T

Toolbox, класс, 64

W

Wine, набор данных, ссылка, 195

Z

Zoo, набор данных, классификация для отбора признаков, 186

А

Агент, 221
Алгоритм адаптивного усиления (AdaBoost), 195

Б

Базовая структура генетического алгоритма, 34
  вычисление приспособленности, 36
  применение операторов отбора, скрещивания и мутации, 36
  проверка условий остановки, 37
  создание начальной популяции, 36
Бионические вычисления, 260

В

Вознаграждение, 221
Встроенные генетические алгоритмы, 76
  выполнение программы, 78
  зал славы, 79
  объект logbook, 77
  объект Statistics, 77
  эксперименты с параметрами алгоритма, 81
Выделение признаков для задачи регрессии Фридмана-1, 181
  представление решения, 182
  решение с помощью генетического алгоритма, 184

Г

Генетические алгоритмы
  глобальная оптимизация, 28
  добавление функции обратного вызова, 249
  и обучение с подкреплением, 221
  искусство решения задач, 57
  непрерывное обучение, 31
  обзор, 20
  оптимизация архитектуры МСП, 212
  оптимизация объединенной конфигурации МСП, 217
  оптимизация функции Химмельблау, 161
  оптимизация функции Eggholder, 155
  преимущества, 28
  применение для реконструкции изображений, 244
  применимость к задачам, не имеющим математического представления, 30
  применимость к сложным задачам, 29
  распараллеливание, 30
  с вещественным кодированием, 49
  сценарии применения, 32
  теорема о схемах, 25
  теоретические основы, 24
  устойчивость к шуму, 30
Генетические алгоритмы, аналогия, 21
  генотип, 22
  мутация, 24
  отбор, 23
  популяция, 22
  скрещивание, 23
  функция приспособленности, 23
Генетические алгоритмы, ограничения, 31
  большой объем счетных операций, 32
  настройка гиперпараметров, 31
  отсутствие гарантированного решения, 32
  преждевременная сходимость, 32
  специальные определения, 31
Генетические операторы
  для задач с вещественными числами, 152
  создание, 65
Генетический поиск на сетке, 196
Генетическое программирование
  обзор, 260
  пример, 262
  упрощение решения, 269
Генотип, 22, 98
Гиперпараметры
  алгоритм адаптивного усиления, 195
  настройка, 194
  обзор, 193
Гипотеза структурных элементов, 24
Глубокое обучение
  и сверточные нейронные сети, 208
  обзор, 206

Глубокое обучение, архитектура классификатора
набор данных Iris, 209
оптимизация, 209
оценка верности классификатора, 211
представление конфигурации скрытого слоя, 210
Градиентный спуск, 27
Графический процессор (GPU), 209

Д
Дарвиновская эволюция, принципы, 21
Дифференциальная эволюция, 278
Древовидные структуры
примитивы, 261
терминальные символы, 261

Ж
Жесткие ограничения, 134

З
Задача коммивояжера
открытые методы класса, 105
постановка, 102
представление на Python, 104
представление решения, 104
решение с помощью генетического алгоритма, 106
улучшение результатов с помощью элитизма, 109
файлы эталонных данных TSPLIB, 103
Задача о восьми ферзях, 125
Задача о маршрутизации транспорта
постановка, 114
представление на Python, 116
представление решения, 115
решение с помощью генетического алгоритма, 118
Задача о рюкзаке, 96
представление на Python, 98
решение с помощью генетического алгоритма, 99
Rosetta Code, 97
Задача о составлении графика дежурств медсестер, 133
жесткие и мягкие ограничения, 134
представление на Python, 135
представление решения, 133
решение на основе генетического алгоритма, 137
Задача об *N* ферзях
представление на Python, 128
представление решения, 126
решение с помощью генетического алгоритма, 129
Задача OneMax, 67
выбор генетических операторов, 68
выбор хромосомы, 67
вычисление приспособленности, 68
задание условия остановки, 68
Зал славы, 79, 130, 185, 214

И
Имитации двоичного скрещивания, 51, 152
Индекс распределения, 52, 153
Искусственная жизнь, 279
Искусственные иммунные системы, 278
алгоритмы иммунных сетей, 279
клональная селекция, 278
негативная селекция, 279
Искусственные нейронные сети, 206
Использование, 110
Исследование, 110
Исследования–использования дилемма, 158

К
Классификация, 177
Кодирование хромосом, 57
Комбинаторная оптимизация, 96
Контроль разбухания кода, 267
Коэффициент распределения, 153
Коэффициент скученности, 153

М
Масштабирование приспособленности, 41
Метод роя частиц, 271
оптимизация функции, 272
Многослойный перцептрон (МСП), 207

оптимизация архитектуры, 212
оптимизация объединенной конфигурации, 217
Муравьиный алгоритм, 278
Мутация, 36
вещественная, 53
инвертирование бита, 48
обменом, 48
обращением, 48
перетасовкой, 49

Н

Наблюдение, 221
Настройка гиперпараметров
объединение с оптимизацией архитектуры, 215
с помощью генетического поиска на сетке, 196
Непрерывные функции и каркас DEAP, 153
Ниши, 54
параллельное образование, 172
сравнение параллельного и последовательного образования, 56

О

Обработка изображений на Python
библиотеки, 240
измерение степени различия двух изображений, 243
рисование с помощью многоугольников, 242
Обратное распространение, 208
Обучение с подкреплением, 220
Обучение с учителем
алгоритмы, 180
выделение признаков, 180
классификация, 177
обзор, 176
регрессия, 179
Оптимизации архитектуры
объединение с настройкой гиперпараметров, 215
Основная теорема генетических алгоритмов, 25
Отбора методы, 37, 57
правило рулетки, 38
ранжированный отбор, 40
стохастическая универсальная выборка, 39
турнирный отбор, 42
Отбор пропорционально приспособленности, 38

П

Поиск на сетке, 194
Поисковые задачи, 96
Попиксельная среднеквадратическая ошибка
определение, 243
применение, 251
Преждевременная сходимость, 29
Признаки, 177
Приспособленности функция, 23, 57

Р

Разделения механизм, 54
Размеченные данные, 220
Раскраска графа, 140
жесткие и мягкие ограничения, 143
представление задачи на Python, 143
решение с помощью генетического алгоритма, 145
Регрессия, 179
Регрессия с градиентным усилением (GBR), 182
Реконструкция изображений, 244
Репозиторий машинного обучения Калифорнийского университета в Ирвайне, ссылка, 195
Решающее дерево (классификатор), ссылка, 194

С

Сверточные нейронные сети, 208
Симионеску функция
определение, 168
оптимизация, 171
Скрещивание смешением (BLX), 50, 152
Скрещивания методы, 43
двухточечное скрещивание, 44

одноточечное скрещивание, 43
равномерное скрещивание, 45
скрещивание для упорядоченных списков, 45
с частичным сопоставлением, 65
k-точечное скрещивание, 44
Скрещивания операция, 36
Случайный поиск, 194
Соблюдение ограничений в поисковых задачах, 125
Среднеквадратическая ошибка (СКО), 183
Структурное сходство (SSIM), 244

Т

Теорема о схемах, 25

У

Условная оптимизация, 168
использование для нахождения нескольких решений, 172
с помощью генетического алгоритма, 170

Ф

Фенотип, 98

Х

Химмельблау функция
образование ниш и разделение для поиска нескольких решений, 165
оптимизация, 159
Хромосомы, 27
для задач с вещественными числами, 152

Ц

Целевая функция, 23

Э

Эволюционные вычисления, 260
Эволюционные стратегии, 277
Элитизм, 53, 112

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Эйял Вирсански

Генетические алгоритмы на Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70 × 100 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 23,24. Тираж 200 экз.

Отпечатано в ПАО «Т8 Издательские Технологии»
109316, Москва, Волгоградский проспект, д. 42, корпус 5

Веб-сайт издательства: **www.dmkpress.com**

Генетические алгоритмы — это семейство алгоритмов поиска, оптимизации и обучения, черпающее идеи из естественной эволюции. Благодаря имитации эволюционных процессов генетические алгоритмы способы преодолевать трудности, присущие традиционным алгоритмам поиска, и находить высококачественные решения в самых разных задачах. Эта книга поможет освоить мощный, но в то же время простой подход к применению генетических алгоритмов, написанных на языке Python, и познакомиться с последними достижениями в области искусственного интеллекта.

После обзора генетических алгоритмов и описания принципов автор рассказывает об их отличиях от традиционных алгоритмов и о типах задач, к которым они применимы, как то: планирование, составление расписаний, игры и анализ функций. Вы также узнаете о том, как генетические алгоритмы позволяют повысить качество моделей машинного и глубокого обучения, решать задачи обучения с подкреплением и выполнять реконструкцию изображений. Наконец, будет упомянуто о некоторых родственных технологиях, открывающих новые возможности для будущих приложений.

Краткое содержание книги:

- применение современных средств на языке Python к созданию генетических алгоритмов;
- использование генетических алгоритмов для оптимизации функций и решения задач планирования и составления расписаний;
- повышение качества моделей машинного обучения и оптимизация архитектуры сети глубокого обучения;
- применение генетических алгоритмов к задачам обучения с подкреплением с использованием библиотеки OpenAI Gym;
- реконструкция изображений с помощью набора полупрозрачных фигур;
- другие бионические методы, в т. ч. генетическое программирование и оптимизация методом роя частиц.

Книга адресована программистам, специалистам по обработке данных и энтузиастам ИИ, желающим применить генетические алгоритмы в решении практических задач. Требуются владение языком Python на рабочем уровне и базовые знания математики и информатики.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@alians-kniga.ru

Packt>

www.дмк.рф

ISBN 978-5-97060-857-9