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

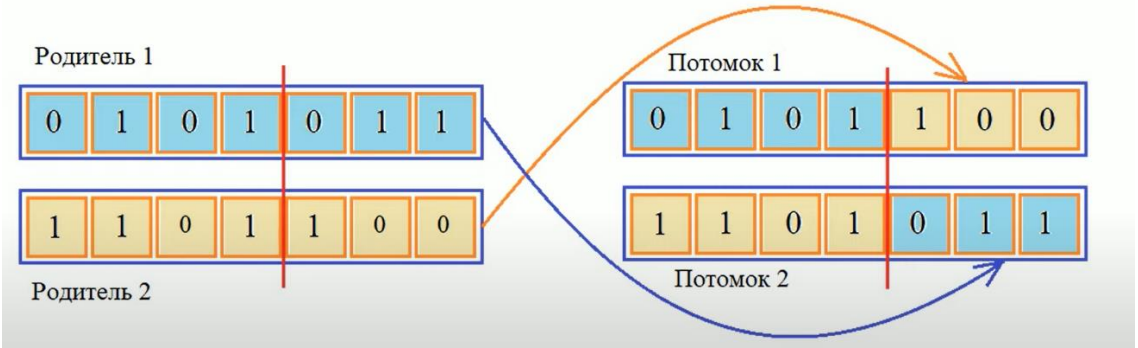
Генетические алгоритмы реализуют упрощенный вариант дарвиновской эволюции. Важным механизмом эволюции является скрещивание, или рекомбинация, – когда потомок приобретает комбинацию признаков своих родителей. Скрещивание помогает поддерживать разнообразие популяции и со временем закреплять лучшие признаки. 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.

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.

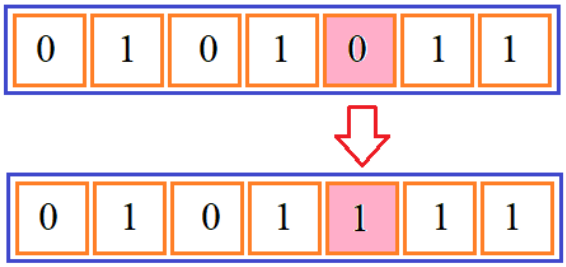
Компоненты генетических алгоритмов

1. 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: 011101010
2. 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.

3. 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.
4. 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.
5. 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.



6. 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.



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.

Преимущества генетических алгоритмов

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

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

Базовая структура генетического алгоритма

Разберем базовую структуру генетического алгоритма:

1. Создать начальную популяцию. 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.

2. Вычислить приспособленность каждого индивидуума в популяции. 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.
3. Отбор. Оператор отбора отвечает за отбор индивидуумов из текущей популяции таким образом, что предпочтение отдается лучшим.
4. Мутация. Оператор мутации вносит случайные изменения в один или несколько генов хромосомы вновь созданного индивидуума. Обычно вероятность мутации очень мала.
5. Скрещивание. 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.
6. Условия остановки выполнены? Может существовать несколько условий, при выполнении которых процесс останавливается:
    6.1.1. достигнуто максимальное количество поколений. Это условие заодно позволяет ограничить время работы алгоритма и потребление им ресурсов системы;
    6.1.2. 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.
    6.1.3. с момента начала прошло заранее определенное время;
    6.1.4. превышен некоторый лимит затрат, например процессорного времени или памяти;
    6.1.5. наилучшее решение заняло часть популяции, большую заранее заданного порога.

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.

Deap

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.

Документация: https://deap.readthedocs.io/en/master/

Исходный код: https://github.com/DEAP/deap

Модуль 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, 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

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 присвоить 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:

```
1 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,).

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:

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

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:

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

Результат работы этой строки:

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

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():

```
1 def sumOfTwo(a, b):
2 | return a + b
```

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

```
1 from deap import base
2 toolbox= base.Toolbox()
3 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:

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

Этот вызов эквивалентен такому:

```
1 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:

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

Опишем, что здесь было сделано.

1. 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.

2. Имя mate зарегистрировано как псевдоним существующей в модуле tools функции cxTwoPoint(). В результате создается оператор toolbox. mate, который выполняет двухточечное скрещивание.
3. 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. Функции отбора находятся в файле selection.py (https://github.com/DEAP/deap). Перечислим некоторые из них:

1. selRoulette() – отбор по правилу рулетки;
2. selStochasticUniversalSampling() – стохастическая универсальная выборка;
3. selTournament() – турнирный отбор.

Функции скрещивания находятся в файле crossover.py:

1. cxOnePoint() – одноточечное скрещивание;
2. cxUniform() – равномерное скрещивание;
3. cxOrdered() – упорядоченное скрещивание (OX1);
4. cxPartialyMatched() – скрещивание с частичным сопоставлением (partially matched crossover – PMX).

В файле mutation.py находятся две функции мутации:

1. mutFlipBit() – мутация инвертированием бита;
2. 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:

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

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

```
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:

```
1 def zeroOrOne():
2 | return random.randint(0, 1)
3 randomList = tools.initRepeat(list, zeroOrOne, 30)
```

Или можно воспользоваться модулем toolbox:

```
1 toolbox.register("zeroOrOne", random.randint, 0, 1)
2 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:

```
1 def someFitnessCalculationFunction(individual):
2 | return _some_calculation_of_the_fitness
3 toolbox.register("evaluate",someFitnessCalculationFunction)
```

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

Задача OneMax

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.

Реализация

Подготовка

1. Импортируем библиотеки и объявим несколько констант, содержащих значения параметров самой задачи и генетического алгоритма:

```
from deap import base
from deap import creator
from deap import tools

import random

import matplotlib.pyplot as plt
import seaborn as sns

# 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
```

2. 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.

3. 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 $a \leq N \leq b$. 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)
```

4. 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,))
```
5. 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)
```
6. 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:
   1. класс Individual в качестве типа контейнера, в который помещаются созданные объекты;
   2. оператор zeroOrOne в качестве функции генерации объектов;
   3. 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)
```

7. регистрируем оператор populationCreator, создающий список индивидуумов. В его определении также используется оператор initRepeat со следующими аргументами:
   1. класс list в качестве типа контейнера;
   2. 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)
```

8. 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
```

Значения приспособленности в DEAP представлены кортежами, поэтому если возвращается всего одно значение, то после него нужно поставить запятую.

9. 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)
```

10. Генетические операторы обычно создаются как псевдонимы существующих функций из модуля tools с конкретными значениями аргументов. В данном случае аргументы будут такими:
    1. турнирный отбор с размером турнира 3;
    2. одноточечное скрещивание;
    3. 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:

```
# турнирный отбор с размером турнира 3
toolbox.register("select", tools.selTournament, tournsize=3)

# одноточечное скрещивание;
toolbox.register("mate", tools.cxOnePoint)

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

Функция main()

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


    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))
```

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
```

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), поскольку 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:
```
best_index = fitnessValues.index(max(fitnessValues))
print("Лучший индивидуум = ", *population[best_index], "\n")
```
17. Визуализация.
```
    sns.set_style("whitegrid")
    plt.plot(maxFitnessValues, color='red')
    plt.plot(meanFitnessValues, color='green')
    plt.legend()
    plt.xlabel('Поколение')
    plt.ylabel('Макс/средняя приспособленность')
    plt.title('Зависимость максимальной и средней приспособленности от поколения')
    plt.show()


if __name__ == '__main__':
    main()
```

Выполнение программы

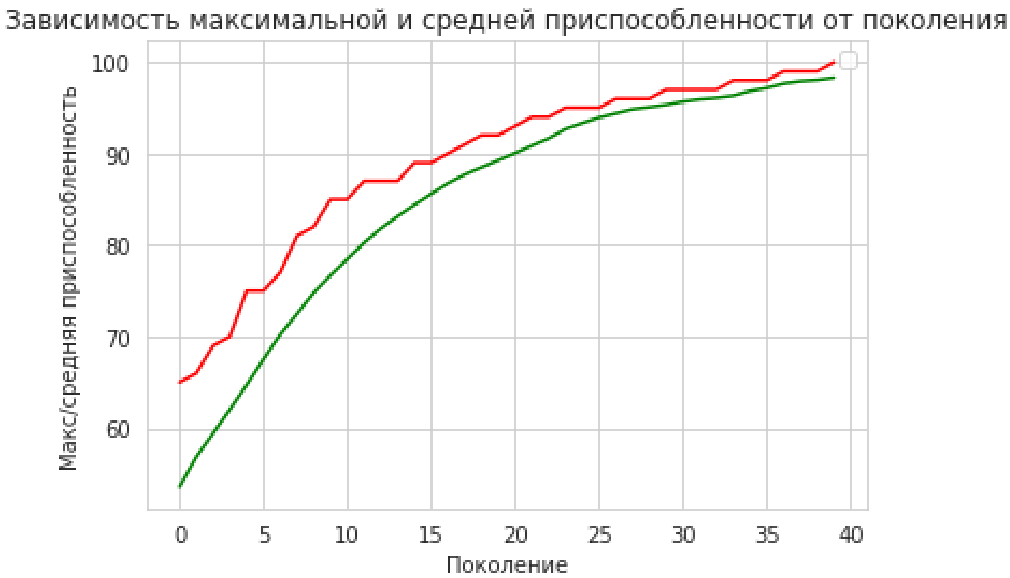
Результат программы:

```
- 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 1 0 1 1 1 1 1 0 1

- Поколение 2: Макс приспособ. = 66.0, Средняя приспособ. = 56.855
Лучший индивидуум =  1 1 1 0 0 1 0 1 1 1 0 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1 1 1 1 0 1 1 0 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1
```

..

```
- Поколение 39: Макс приспособ. = 99.0, Средняя приспособ. = 98.04
Лучший индивидуум =  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

- 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
```

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.



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

<u>Использование встроенных алгоритмов</u>

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.

Далее реализуем то же решение задачи OneMax, что и программа из предыдущего раздела, но с меньшим объемом кода. Отличается только метод main.

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")
```

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
```

..

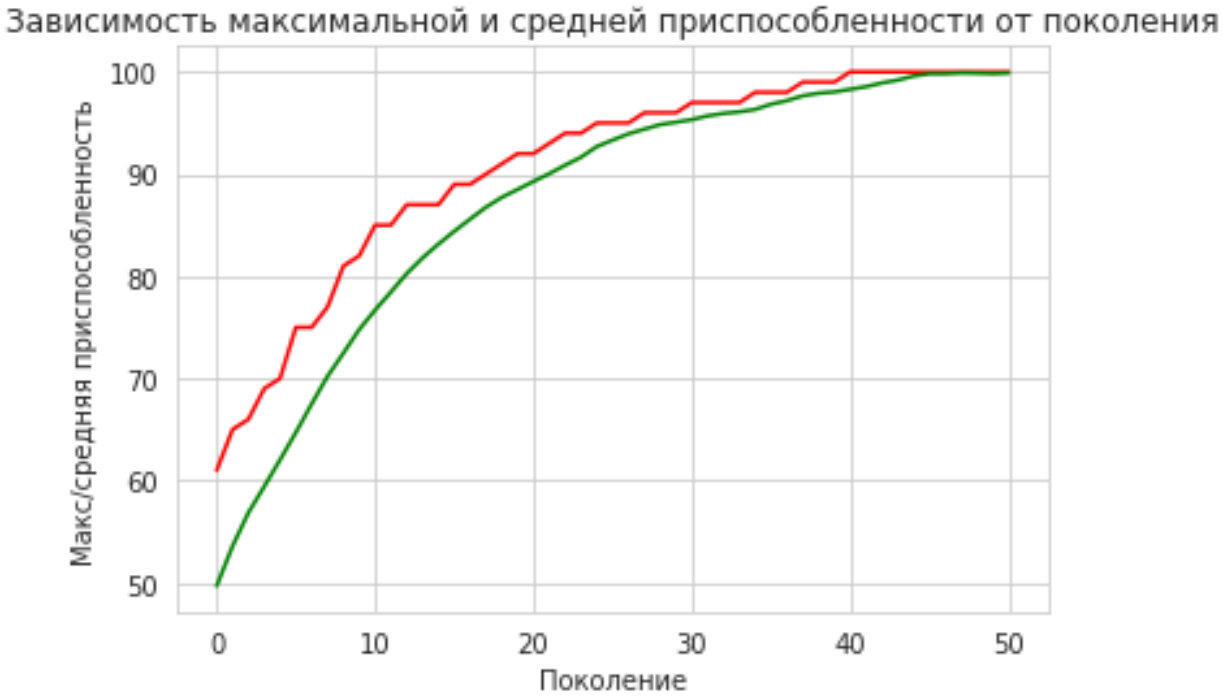
```
39      192     99      98.04
40      173     100     98.29

..

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:

1. в выводе имеется строка для поколения 0, раньше мы ее не включали;
2. 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.



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.

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 приведен вывод - 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, 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, 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, 0, 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, 0, 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
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
```

Лучшим является все тот же индивидуум, содержащий одни единицы.

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

Дополнение моделей машинного обучения методами выделения признаков

Выделение признаков – это процесс отбора самых важных и полезных признаков из общего их множества. Помимо повышения верности модели, успешное выделение признаков дает следующие преимущества:

1. уменьшается время обучения модели;
2. обученная модель проще, ее легче интерпретировать;
3. обученная модель лучше обобщается, т. е. лучше работает на новых данных, которые не предъявлялись в процессе обучения.

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 x0, …, x4 и случайного шума:

$$y(x0, x1, x2, x3, x4) = 10 \cdot \sin(\pi \cdot x0 \cdot x1) + 20(x2 – 0.5)2 + 10x3 + 5x4 + noise \cdot N(0,1).$$

Переменные x0, …, x4 – 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 x0, …, x4 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.

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.

Реализация

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)
```
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)
```
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
```

Исходный код:

```python
import numpy as np

from sklearn import model_selection
from sklearn import datasets

from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import seaborn as sns

class Friedman1Test:


    VALIDATION_SIZE = 0.20
    NOISE = 1.0

    def __init__(self, numFeatures, numSamples, randomSeed):


        self.numFeatures = numFeatures
        self.numSamples = numSamples
        self.randomSeed = randomSeed

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

        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)

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

    def __len__(self):

        return self.numFeatures


    def getMSE(self, zeroOneList):

        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)

        self.regressor.fit(currentX_train, self.y_train)

        prediction = self.regressor.predict(currentX_validation)

        return mean_squared_error(self.y_validation, prediction)



def main():

    test = Friedman1Test(numFeatures=15, numSamples=60, randomSeed=42)

    scores = []

    for n in range(1, len(test) + 1):
        nFirstFeatures = [1] * n + [0] * (len(test) - n)
        score = test.getMSE(nFirstFeatures)
        print("%d первых признаков: оценка = %f" % (n, score))
        scores.append(score)

    sns.set_style("whitegrid")
    plt.plot([i + 1 for i in range(len(test))], scores, color='red')
    plt.xticks(np.arange(1, len(test) + 1, 1.0))
    plt.xlabel('СКО для выделенных признаков')
    plt.ylabel('СКО')
    plt.title('n первых признаков')
    plt.show()


if __name__ == "__main__":
    main()
```

Решение с помощью генетического алгоритма

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. Разница в том, что функция приспособленности теперь возвращает СКО модели регрессии, вычисляемую в классе Friedman1Test.

1. Сначала создаем экземпляр класса Friedman1Test с нужными параметрами:
```
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),


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

Исходный код:

```python
from deap import base
from deap import creator
from deap import tools
import random
import numpy
import matplotlib.pyplot as plt
import seaborn as sns

NUM_OF_FEATURES = 15
NUM_OF_SAMPLES = 60

POPULATION_SIZE = 30
P_CROSSOVER = 0.9
P_MUTATION = 0.2
MAX_GENERATIONS = 30
HALL_OF_FAME_SIZE = 5

RANDOM_SEED = 42
random.seed(RANDOM_SEED)

friedman = Friedman1Test(NUM_OF_FEATURES, NUM_OF_SAMPLES, RANDOM_SEED)

toolbox = base.Toolbox()

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


creator.create("Individual", list, fitness=creator.FitnessMin)

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

def friedmanTestScore(individual):
    return friedman.getMSE(individual),

toolbox.register("evaluate", friedmanTestScore)

toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/len(friedman))

def main():

    population = toolbox.populationCreator(n=POPULATION_SIZE)

    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("min", numpy.min)
    stats.register("avg", numpy.mean)

    hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

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

    best = hof.items[0]
    print("--  Лучший индивидуум = ", best)
    print("-- Лучшая приспособленность = ", best.fitness.values[0])

    minFitnessValues, meanFitnessValues = logbook.select("min", "avg")

    sns.set_style("whitegrid")
    plt.plot(minFitnessValues, color='red')
    plt.plot(meanFitnessValues, color='green')
    plt.xlabel('Популяция')
    plt.ylabel('Минимальная / Средняя приспособленность')
    plt.title('Минимальная и средняя приспособленность по популяции')
    plt.show()


if __name__ == "__main__":
```

Вывод программы:

```
gen     nevals  min     avg
0       30      13.4946 30.2123
1       20      13.4946 22.9373
2       22      11.1869 19.0558
3       23      11.1869 17.0366
4       24      11.1869 14.0624
5       21      10.8123 13.2361
6       20      10.5795 12.3553
7       21      10.5795 12.8751
8       21      9.67682 12.0993
9       23      9.67682 11.2531
10      20      9.67682 10.8748
11      16      9.67682 10.854
12      22      7.45319 10.5713
13      22      7.45319 10.1786
14      22      7.45319 10.1182
15      22      7.26529 11.4639
16      23      7.26529 10.8954
17      23      6.70267 9.41857
18      24      6.70267 10.9443
19      23      6.70267 10.3826
20      24      6.70267 9.28936
21      23      6.70267 8.39784
22      25      6.70267 9.43918
23      24      6.70267 7.29398
24      19      6.70267 7.57122
25      19      6.70267 6.88776
26      24      6.70267 7.48048
27      24      6.70267 7.96912
28      19      6.70267 7.23222
29      22      6.70267 7.67753
30      24      6.70267 8.10665
-- Лучший индивидуум =  [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
-- Лучшая приспособленность =  6.702668910463276
```

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.

Выделение признаков для классификации набора данных Zoo

№	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.

Модель классификации, обучаемая на этом наборе, будет использовать признаки со 2 по 17 для предсказания признака 18.

Реализация

1. Метод класса __init__() загружает набор данных Zoo и пропускает первый признак, название животного:
```
self.data = read_csv('zoo.data', 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, в данном случае воспользовались 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,
                                   shuffle=True)
```

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)
```
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 97 %:
```
Выделены все признаки:  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] , точность =  0.9704761904761904
```
Далее мы попробуем улучшить верность классификатора, выделив только часть признаков.

Исходный код:

```python
import random
from pandas import read_csv
from sklearn import model_selection
from sklearn.tree import DecisionTreeClassifier
class Zoo:

    NUM_FOLDS = 5

    def __init__(self, randomSeed):

        self.randomSeed = randomSeed

        self.data = read_csv('zoo.data', header=None, usecols=range(1, 18))

        self.X = self.data.iloc[:, 0:16]
        self.y = self.data.iloc[:, 16]

        self.kfold = model_selection.KFold(n_splits=self.NUM_FOLDS,
                                           random_state=self.randomSeed,
                                           shuffle=True)

        self.classifier = DecisionTreeClassifier(random_state=self.randomSeed)

    def __len__(self):
        return self.X.shape[1]

    def getMeanAccuracy(self, zeroOneList):

        zeroIndices = [i for i, n in enumerate(zeroOneList) if n == 0]
        currentX = self.X.drop(self.X.columns[zeroIndices], axis=1)

        cv_results = model_selection.cross_val_score(self.classifier,
                                                     currentX, self.y,
                                                     cv=self.kfold,
                                                     scoring='accuracy')

        return cv_results.mean()

def main():
    zoo = Zoo(randomSeed=42)

    allOnes = [1] * len(zoo)
    print("-- Выделены все признаки: ", allOnes, ", верность = ",
          zoo.getMeanAccuracy(allOnes))

if __name__ == "__main__":
    main()
```

Решение с помощью генетического алгоритма

1. Сначала создаем экземпляр класса Zoo и передаем конструктору начальное значение генератора случайных чисел, чтобы обеспечить воспроизводимость результатов:
```
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. 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:
    1. мы запретили алгоритму не выбирать ни одного признака (создавать индивидуума, представленного только нулями), поскольку в таком случае классификатор возбудил бы исключение;
    2. 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,


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 = 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:

```
print("- Лучшие решения:")
for i in range(HALL_OF_FAME_SIZE):
    print(i, ": ", hof.items[i], ", fitness = ", hof.items[i].fitness.values[0],
          ", верность = ", zoo.getMeanAccuracy(hof.items[i]),
          ", признаков = ", sum(hof.items[i]))
```

Исходный код:

```python
from deap import base
from deap import creator
from deap import tools
import random
import numpy
import matplotlib.pyplot as plt
import seaborn as sns
POPULATION_SIZE = 50
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.3   # probability for mutating an individual
MAX_GENERATIONS = 50
HALL_OF_FAME_SIZE = 5
FEATURE_PENALTY_FACTOR = 0.001
# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)

# create the Zoo test class:
zoo = Zoo(RANDOM_SEED)

toolbox = base.Toolbox()

# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))

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

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)

# fitness calculation
def zooClassificationAccuracy(individual):
    numFeaturesUsed = sum(individual)
    if numFeaturesUsed == 0:
        return 0.0,
    else:
        accuracy = zoo.getMeanAccuracy(individual)
        return accuracy - FEATURE_PENALTY_FACTOR * numFeaturesUsed,  # return a tuple

toolbox.register("evaluate", zooClassificationAccuracy)
# genetic operators:mutFlipBit
toolbox.register("select", tools.selTournament, tournsize=2)

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

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

```python
def main():

    # create initial population (generation 0):
    population = toolbox.populationCreator(n=POPULATION_SIZE)
    # prepare the statistics object:
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("max", numpy.max)
    stats.register("avg", numpy.mean)
    # define the hall-of-fame object:
    hof = tools.HallOfFame(HALL_OF_FAME_SIZE)
    # perform the Genetic Algorithm flow with hof feature added:
    population, logbook = eaSimpleWithElitism(population, toolbox,
                                              cxpb=P_CROSSOVER, mutpb=P_MUTATION,
                                                    ngen=MAX_GENERATIONS,
                                              stats=stats,
                                              halloffame=hof,
                                              verbose=True)

    # print best solution found:
    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]))

    # extract statistics:
    maxFitnessValues, meanFitnessValues = logbook.select("max", "avg")
    # plot statistics:
    sns.set_style("whitegrid")
    plt.plot(maxFitnessValues, color='red')
    plt.plot(meanFitnessValues, color='green')
    plt.xlabel('Generation')
    plt.ylabel('Max / Average Fitness')
    plt.title('Max and Average fitness over Generations')
    plt.show()

if __name__ == "__main__":
    main()
```

```
- Лучшие решения:
0 :  [0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0] , приспособленность =  0.985 , верность =  0.99 , признаков =  5
1 :  [1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0] , приспособленность =  0.984 , верность =  0.99 , признаков =  6
2 :  [0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0] , приспособленность =  0.984 , верность =  0.99 , признаков =  6
3 :  [0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0] , приспособленность =  0.984 , верность =  0.99 , признаков =  6
4 :  [0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0] , приспособленность =  0.984 , верность =  0.99 , признаков =  6
```

Эти результаты показывают, что все пять лучших решений достигли верности 99 %, используя пять или шесть признаков из 16. Самым лучшим является решение с пятью признаками:

1. млекопитающее;
2. водное;
3. наличие позвоночника;
4. плавники;
5. количество ног;

Выделив именно эти 5 признаков из 16, мы смогли не только уменьшить размерность задачи, но и повысить верность модели с 97 % до 99 %.

Практическое задание

1. Загрузить датасет breast-cancer-wisconsin1.data;
2. Реализовать выделение признаков для классификации набора данных breast-cancer-wisconsin1.data по примеру приведенным выше с zoo.data;
3. Вывести лучшее решение и на сколько увеличилось значение верности;
4. Сформировать отчёт.