Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра высшей математики

Н. В. Князюк, О. В. Рыкова

МЕТОДЫ МАШИННОГО ОБУЧЕНИЯ

*Рекомендовано УМО по образованию
в области информатики и радиоэлектроники
в качестве пособия для специальностей
6-05-0611-02 «Информационная безопасность»,
6-05-0611-04 «Электронная экономика»,
6-05-0611-06 «Системы и сети инфокоммуникаций»,
6-05-0612-03 «Системы управления информацией»*

Минск БГУИР 2026

УДК 004.85(076)
ББК 32.973.202я73
К54

Р е ц е н з е н т ы:

кафедра цифровых систем и технологий Института бизнеса
Белорусского государственного университета
(протокол № 8 от 11.03.2024);

доцент кафедры математических методов в экономике
учреждения образования «Белорусский государственный
экономический университет»
кандидат физико-математических наук, доцент Ю. Л. Ратушева

Князюк, Н. В.

К54 Методы машинного обучения : пособие / Н. В. Князюк, О. В. Рыкова. – Минск : БГУИР, 2026. – 99 с. : ил.

ISBN 978-985-543-805-3.

Изложены основы машинного обучения, приведены необходимые сведения для приобретения базовых навыков работы с языком программирования Python. Даны определения основным понятиям, описаны основные алгоритмы машинного обучения. Подробно разобраны задания с применением математического аппарата и приведены программы на языке Python. Рекомендуется для студентов всех форм обучения инженерно-технических специальностей вузов.

УДК 004.85(076)
ББК 32.973.202я73

ISBN 978-985-543-805-3

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В науке и реальной жизни важной задачей является прогнозирование поведения сложных систем на основании их прошлого поведения. Многие задачи, возникающие в практических приложениях, не могут быть решены заранее известными методами или алгоритмами. Это происходит по той причине, что нам заранее неизвестны механизмы порождения исходных данных или же известная информация недостаточна для построения модели источника, генерирующего поступающие данные. Как говорят, мы получаем данные из «черного ящика». В этих условиях ничего не остается, кроме изучения доступной последовательности исходных данных и попыток прогнозирования при помощи совершенствования схемы в ходе процесса прогнозирования. Подход, при котором прошлые данные или примеры используются для первоначального формирования и совершенствования схемы предсказания, называется методом машинного обучения (Machine Learning – ML). Машинное обучение – чрезвычайно широкая и динамически развивающаяся область исследований, использующая огромное число теоретических и практических методов. Методы машинного обучения – это алгоритмы и модели, которые используются для обработки и анализа больших объемов данных. Они позволяют компьютерным системам получать знания из данных и применять их для принятия решений и прогнозирования.

В настоящее время существует несколько программных систем и библиотек программ, реализующих алгоритмы машинного обучения. Наиболее популярным средством является язык программирования Python и ряд библиотек (NumPy, pandas, scikit-learn, Matplotlib и др.), использующих его для реализации алгоритмов машинного обучения.

В первой части пособия – теоретической – рассматриваются модели машинного обучения, основные метрики оценки качествы работы алгоритмов, методы подготовки данных, приводятся примеры и необходимые пояснения. Также рассматриваются основные понятия языка Python, необходимые для работы. Во второй части – практической – приводятся методические рекомендации по решению базовых заданий, а также задания для самоподготовки.

Для успешного освоения данного курса нужно знать теорию вероятностей и математическую статистику, математический анализ, линейную алгебру.

ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

1. Основы языка программирования Python

Python – высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Язык является полностью объектно-ориентированным. Синтаксис ядра языка Python минималистичен. Стандартная библиотека включает большой набор полезных переносимых функций, включая как возможности работы с текстом, так и средства написания сетевых приложений.

Python служит средством для реализации машинного обучения. В Python существует множество библиотек и фреймворков, которые позволяют создавать и обучать модели машинного обучения. Перечислим некоторые из наиболее популярных библиотек Python для машинного обучения.

1. TensorFlow – это библиотека от Google, которая позволяет создавать и обучать модели машинного обучения. TensorFlow предоставляет широкий набор инструментов для работы с данными, создания моделей и их оптимизации. Она также поддерживает распределенное обучение (обучение на нескольких устройствах).

2. Keras – это высокоуровневый фреймворк для создания нейронных сетей. Он позволяет создавать модели машинного обучения с помощью простых и понятных API. Keras также поддерживает интеграцию с TensorFlow и другими библиотеками машинного обучения.

3. PyTorch – это библиотека от Facebook, которая предоставляет удобный интерфейс для создания и обучения нейронных сетей. PyTorch также поддерживает автоматическое дифференцирование, что позволяет легко оптимизировать модели.

4. Scikit-learn – это библиотека для машинного обучения, которая предоставляет широкий набор алгоритмов для классификации, регрессии, кластеризации и других задач. Scikit-learn также предоставляет инструменты для работы с данными, включая предобработку и визуализацию.

5. Pandas – это библиотека для работы с данными, которая предоставляет инструменты для чтения, записи и манипулирования табличными данными. Pandas также поддерживает работу с временными рядами и многомерными данными.

6. NumPy – это библиотека для работы с массивами данных, которая предоставляет инструменты для математических операций, линейной алгебры и статистики. NumPy также поддерживает работу с многомерными массивами данных.

Заметим, что это не полный список библиотек Python для машинного обучения, и выбор конкретной библиотеки зависит от требований к модели и уровня опыта пользователя.

1.1. Переменные в Python

Переменные в Python предназначены для хранения данных в памяти компьютера и для проведения над ними различных операций.

Переменная – это область памяти компьютера, у которой есть имя. Имя переменной состоит из трех частей:

- имя (или идентификатор) – это название, придуманное программистом, чтобы обращаться к переменной;
- значение – это информация, хранящаяся в памяти компьютера, с которой работает программа;
- адрес – это номер ячейки памяти, в которой хранится значение переменной.

Например, переменная **name** хранит строку **"Tom"**.

```
name = "Tom"
```

Имя переменной должно начинаться с буквы или знака подчеркивания, но не с цифры. Оно может содержать строчные, заглавные буквы, цифры и знаки подчеркивания.

Можно менять значение переменной в течение работы программы.

```
name = "Tom"
print(name)    # Вывод: Tom
name = "Bob"
print(name)    # Вывод: Bob
```

Переменные выполняют две важные функции: делают код понятнее и дают возможность многократно использовать введенные данные.

1.2. Типы данных в Python

В Python существуют разные типы данных, которые можно сгруппировать следующим образом:

- числовые данные: **int**, **float**, **complex** (целые числа, числа с плавающей точкой, комплексные числа);
- строковые типы: **str** (строки);
- последовательности: **list**, **tuple**, **range** (список, кортеж, диапазон);
- бинарные типы: **bytes**, **bytearray**, **memoryview** (байты, массивы байтов, представление памяти);
- ассоциативные данные: **dict** (словари);
- логический тип: **bool** (булевый тип);
- множественные типы: **set**, **frozenset** (множество, замороженное множество).

Чтобы узнать тип данных, нужно воспользоваться встроенной функцией **type()**.

```
>>> a = 3.5
>>> type(a)
<class 'float'>
```

Python – язык с динамической типизацией, т. е. значения присваиваются переменным не при компиляции, а при выполнении программы. Поэтому объявлять переменную заранее не нужно. Можно переменной присвоить значение другого типа по мере необходимости. Рассмотрим пример.

```
x = 10
print(type(x)) # <class ' int'>

x = "Hello"
print(type(x)) # <class ' str'>

x = [1,2,3]
print(type(x)) # <class ' list'>
```

Здесь переменная **x** сначала содержит целое число и имеет тип **int**. Затем мы присваиваем ей строку, и ее тип изменяется на **str**. Наконец, мы присваиваем ей список, и ее тип становится **list**. Все это происходит динамически во время выполнения программы.

1.3. Строки. Функции и методы строк

Строковый тип данных – один из основных типов данных в Python. Он используется для хранения символьной информации: букв, чисел, знаков препинания и других символов. На практике применяется, например, для записи Ф. И. О. или адресов клиентов в базах данных. Для создания строк мы используем парные кавычки ' ' или " ".

```
s1 = 'Python'
s2 = "Pascal"
```

В Python существует множество методов для работы со строками. Рассмотрим базовые операции над строками.

Конкатенация (сложение) объединяет строки.

```
s1 = 'spam'
s2 = 'eggs'
print(s1 + s2) # Вывод: 'spameggs'
```

При использовании функции **print()** по умолчанию каждый новый элемент выводится с новой строки или с пробелом.

Дублирование строки создает несколько копий строки.

```
print('spam' * 3) # Вывод: 'spamspamspam'
```

Длина строки (функция **len ()**) возвращает количество символов в строке.

```
print(len('spam') # Вывод: 4
```

Индексация строк. Когда необходимо обратиться к конкретному символу в строке, используют квадратные скобки **[]**, в которых указывается индекс (номер) нужного символа в строке.

Пусть **s = 'Python'**. Табл. 1 показывает, как работает индексация.

Табл. 1. Индексация строк

Выражение	Результат	Пояснение
s[0]	P	первый символ строки
s[1]	y	второй символ строки
s[2]	t	третий символ строки
s[3]	h	четвертый символ строки
s[4]	o	пятый символ строки
s[5]	n	шестой символ строки

Примечание. Первый символ строки равен **s[0]**, а не **s[1]**. В Python индексация начинается с 0.

В отличие от многих языков программирования в Python есть возможность работы с отрицательными индексами. Если первый символ строки имеет индекс 0, то последнему элементу присваивается индекс –1 (табл. 2).

Табл. 2. Использование отрицательных индексов

Выражение	Результат	Пояснение
s[-6]	P	первый символ строки
s[-5]	y	второй символ строки
s[-4]	t	третий символ строки
s[-3]	h	четвертый символ строки
s[-2]	o	пятый символ строки
s[-1]	n	шестой символ строки

Частая ошибка – обращение по несуществующему индексу в строке. Например, если **s = 'Python'** и мы попытаемся обратиться к **s[17]**, то получим ошибку.

```
IndexError: string index out of range
```

Ошибка возникает, поскольку строка содержит всего шесть символов.

Рассмотрим функции строк.

1. Следующие функции осуществляют поиск подстроки в строке:

- **S.find(str, [start], [end])** возвращает номер первого вхождения или –1;
- **S.rfind(str, [start], [end])** возвращает номер последнего вхождения или –1;
- **S.index(str, [start], [end])** возвращает номер первого вхождения или вызывает ValueError;
- **S.rindex(str, [start], [end])** возвращает номер последнего вхождения или вызывает ValueError.

2. **S.replace(шаблон, замена [, maxcount])** заменяет шаблон на замену, а **, maxcount** ограничивает количество замен.

3. **S.split(символ)** осуществляет разбиение строки по разделителю.

4. **S.isdigit()** выясняет, состоит ли строка из цифр.

5. **S.isalpha()** выясняет, состоит ли строка из букв.

6. **S.isalnum()** выясняет, состоит ли строка из цифр или букв.

7. **S.islower()** выясняет, состоит ли строка из символов в нижнем регистре.

8. **S.isupper()** выясняет, состоит ли строка из символов в верхнем регистре.

9. **S.isspace()** выясняет, состоит ли строка только из пробельных символов.

F-строки в Python, или «форматированные строковые литералы», представляют собой новый способ форматирования строк. Они обозначаются литерой **f** перед кавычками. F-строки позволяют вставлять выражения внутрь строковых литералов, используя фигурные скобки **{}**. Эти выражения оцениваются во время выполнения и заменяются на их значения. Рассмотрим пример.

```
name = "Дмитрий"
age = 25
print (f"Меня зовут {name}. Мне {age} лет.")
```

В примере **{age}** и **{name}** внутри F-строки заменяются на значения переменных **age** и **name** соответственно. F-строки делают текст более читаемым и работают быстрее, чем другие способы форматирования. Также они поддерживают расширенное форматирование чисел.

1.4. Операторы в Python

Оператор в Python – это специальная конструкция для операций над данными и управления логикой программы. Рассмотрим основные виды операторов и их функциональность.

1. Арифметические операторы (служат для выполнения математических операций):

- + (сложение);
- - (вычитание);
- * (умножение);
- / (деление);
- // (целочисленное деление);
- % (остаток от деления);
- ** (возведение в степень).

2. Операторы сравнения или реляционные операторы (используются для сравнения значений):

- == (равно);
- != (не равно);
- < (меньше);
- > (больше);
- <= (меньше или равно);
- >= (больше или равно).

3. Логические операторы (позволяют комбинировать условия):

- and (логическое И);
- or (логическое ИЛИ);
- not (логическое НЕ).

4. Операторы присваивания (используются для присвоения значений переменным):

- = (присваивание);
- +=, -= и др. (составные операторы присваивания).

5. Побитовые операторы (работают с битами чисел):

- & (побитовое И);
- | (побитовое ИЛИ);
- ^ (побитовое исключающее ИЛИ);
- ~ (побитовое НЕ);
- << (побитовый сдвиг влево);
- >> (побитовый сдвиг вправо).

6. Операторы членства (проверяют, принадлежит ли элемент к последовательности):

- in (принадлежит);
- not in (не принадлежит).

7. Операторы тождественности (сравнивают объекты на идентичность):

- is (идентичен);
- is not (не идентичен).

Операторы помогают нам создавать сложные выражения и управлять потоком выполнения программы.

Рассмотрим примеры.

1. Арифметические операторы:

```
x = 10
y = 3

print(x + y) # сложение: 13
print(x - y) # вычитание: 7
print(x * y) # умножение: 30
print(x / y) # деление: 3.33333333333335
print(x % y) # остаток от деления: 1
print(x // y) # деление нацело: 3
print(x ** y) # возведение в степень: 1000
```

2. Операторы сравнения:

```
x = 10
y = 3

print(x == y) # равно: False
print(x != y) # не равно: True
print(x > y) # больше: True
print(x < y) # меньше: False
print(x >= y) # больше или равно: True
print(x <= y) # меньше или равно: False
```

3. Логические операторы:

```
x = 5
print(x > 3 and x < 10) # Вернет True, потому что 5 больше 3 и 5 меньше 10

x = 5
print(x > 3 and x < 10) # Вернет True, потому что 5 больше 3 и 5 меньше 10

x = 5
print(not(x > 3 and x < 10)) # Вернет False, потому что not используется для инвертирования истинного условия
```

4. Операторы присваивания:

```
x = 10

print(x) # 10
x += 5

print(x) # 15
x -= 2
print(x) # 13
```

5. Побитовые операторы:

```
x = 10   # 1010 в двоичной системе
y = 4   # 0100 в двоичной системе

print(x & y) # битовый оператор И: 0
print(x | y) # битовый оператор OR: 14
print(~x) # битовый оператор НЕТ: -11
print(x ^ y) # битовый оператор XOR: 14
print(x >> y) # правый сдвиг: 2
print(x << y) # левый сдвиг: 40
```

6. Операторы членства:

```
x = [1,2,3,4,5]
print(3 in x) # Вернет True, потому что 3 присутствует в списке x

x = [1,2,3,4,5]
print(6 in x) # Вернет True, потому что 6 отсутствует в списке x
```

7. Операторы тождественности:

```
x = [1,2,3]
y = x
print(y in x) # Вернет True, потому что y и x указывают на один и тот же объект

x = [1,2,3]
y = [1,2,3]
print(y in not x) # Вернет True, потому что y и x указывают на разные объекты
```

1.5. Условный оператор if в Python

Оператор **if** в Python – это конструкция, которая позволяет выполнить один из двух блоков кода в зависимости от того, выполняется условие (выражение после **if**) как истинное или ложное. Если условие истинно (*True*), то выполняется блок кода под **if**, если же условие ложно (*False*), то блок кода под **else**.

```
1  if условие:
2      # Блок кода, который выполняется, если условие истинно
3  else:
4      # Блок кода, который выполняется, если условие ложно
```

Примечания: 1. После условия необходимо поставить знак двоеточие **:**.

2. Работая с условной конструкцией, важно знать, что Python интерпретирует ненулевые значения как *True*. **None** и **0** интерпретируются как *False*.

Чтобы оставить блок пустым, нужно воспользоваться ключевым словом **pass**.

```
if 7>3:
    pass
print('Ошибки нет, внутри if пустой блок команд')
```

Программа сможет обрабатывать более двух событий без использования вложенного оператора **if**. Для этого нам понадобится инструкция **elif**. Само слово **elif** образовано от **else if**, что переводится как «иначе если».

Примечание. После **else** условие никогда не ставится. После **elif** обязательно нужно поставить логическое выражение.

```
if 5 > 1:
    print(1)
    print(2)
elif 3 > 2:
    print(3)
    print(4)
elif 6 > 4:
    print(5)
    print(6)
```

1.6. Циклы while и for в Python

Цикл **while** предназначен для неоднократного исполнения определенной последовательности действий до тех пор, пока заданное условие остается истинным. Поэтому этот цикл называют условным. После ключевого слова **while** пишется условие (логическое выражение, которое принимает значение True или False). После условия обязательно ставится знак двоеточия **:** и затем с новой строки одна под другой на одном уровне отступа перечисляются инструкции, которые будут выполняться в цикле.

Рассмотрим пример цикла **while**:

```
n = int(input())
while n ≠ 0:
        print(n + 5)
        n = int(input())
```

Цикл выполняется, пока пользователь не введет **0**.

В Python существуют два ключевых слова, с помощью которых можно остановить итерацию цикла преждевременно:

1. Ключевое слово **break** прерывает цикл и передает управление в конец цикла.

```
a = 1
while a < 5
        a += 1
        if a == 3:
                break
print(a) # 2
```

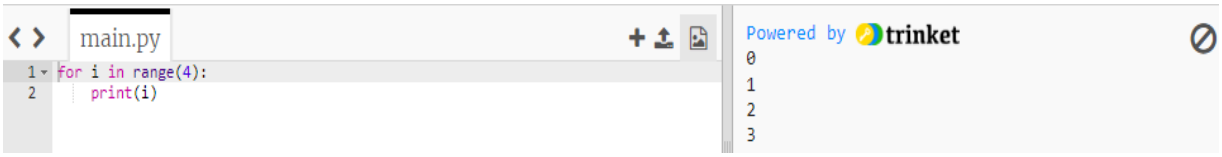
2. Ключевое слово **continue** прерывает текущую итерацию и передает управление в начало цикла, после чего условие снова проверяется. Если оно истинно, то исполняется следующая итерация.

```
a = 1

while a < 5:
        a += 1
        if a == a
                continue
        print(a) #2,4,5
```

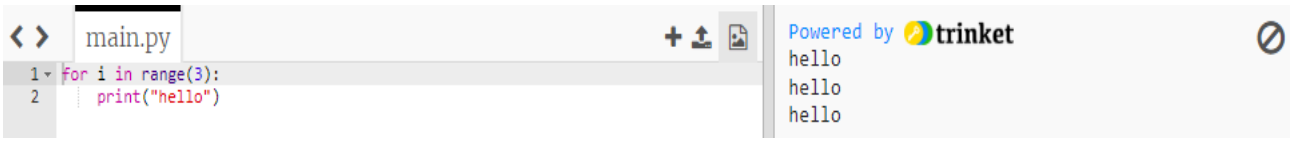
В отличие от цикла **while** цикл **for** используется в тех случаях, когда заранее известно количество итераций, совершаемых в цикле. Число исполнений цикла **for** определяется функцией **range()**.

Рассмотрим использование цикла **for** с функцией **range()**.

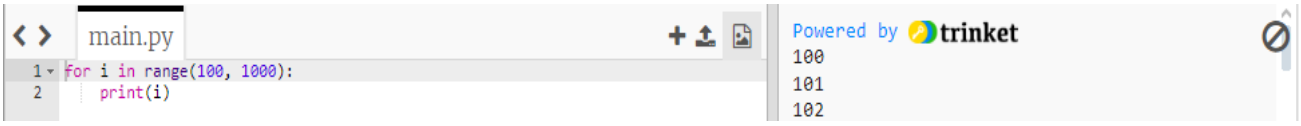


Существует два варианта применения функции **range()** в цикле **for**:

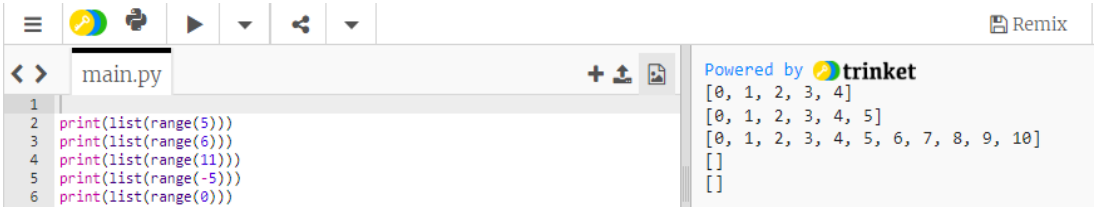
1. Повторение какого-то действия определенное количество раз.



2. Обход какой-то заданной последовательности. Мы это реализовывали, когда в цикле выводили ***i***, т. е. проходили последовательность чисел от 0 до 3. Если нужно пройти другую последовательность, например все трехзначные числа, то необходимо написать следующий код:

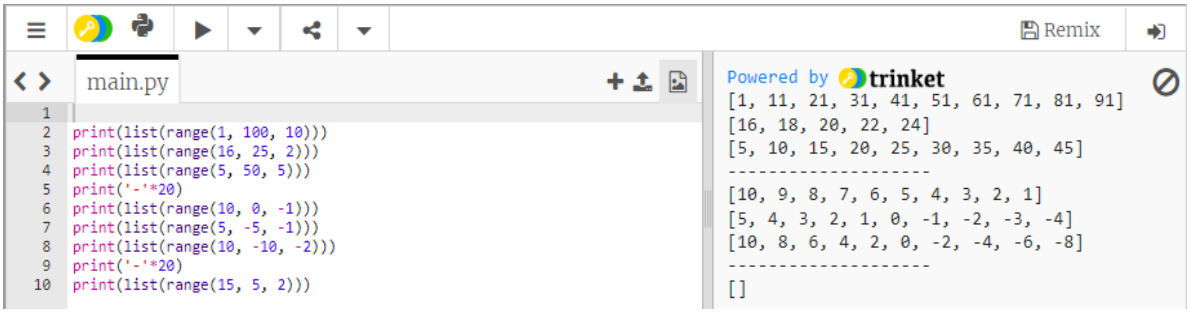


С помощью функции **range()** можно сформировать конечную арифметическую прогрессию. Заметим, что все последовательности начинаются с нуля и не включают переданное число. Поэтому если нужно, чтобы последовательность заканчивалась числом **5**, то необходимо вызвать **range(6)**. Если указать нуль или отрицательное значение, то получим пустую последовательность. Это связано с тем, что по умолчанию функция **range()** формирует возрастающую арифметическую прогрессию, начинающуюся с нуля, с шагом один. А от нуля до отрицательного числа, прибавляя единицу, пройти не получится.



В функцию **range()** можно также передавать второй параметр. В таком случае первое число говорит о том, откуда будет начинаться отсчет, а второе – где он будет заканчиваться (не включительно). Рассмотрим пример.

Повлиять на возрастание или убывание последовательности можно при помощи третьего параметра. Он влияет на шаг арифметической последовательности, т. е. на разницу между элементами.



С помощью функции **range()** можно:

- найти сумму арифметической прогрессии. Например, посчитать сумму чисел от 1 до 100. Для этого можно воспользоваться функцией **sum()** и передать ей на вход последовательность чисел от **1** до **100**:

sum(range(1, 101)) # получим 5050

- посчитать количество чисел в последовательности при помощи функции **len()**. Например, нужно найти количество чисел от 5 до 15 (не включительно) при шаге в 5:

len(range(5, 15, 5)) # получим 2

- использовать в конструкции множественного присвоения.

a, b, c = range(5,16, 5) # получим, что a = 5, b = 10, c = 15

- результат функции сохранить в переменную:

r = range(1, 7)

Для этой переменной можно узнать количество элементов:

len(r) # получим 6

или обратиться к ее элементу по индексу:

r[1] # получим 2

Примечание. Индексация начинается с **0**, поэтому, чтобы получить **1**, необходимо указать **r[0]**. Если использовать **r[1]**, то на выходе получится **2**.

1.7. Структуры данных

Структуры данных необходимы для организации и хранения данных. Основные часто используемые структуры данных в Python: список, кортеж, словарь.

Список

Список – это структура данных, которая хранит последовательность, т. е. упорядоченный набор элементов.

Правила объявления списка:

- список создается с помощью квадратных скобок **[]**;
- элементы списка нужно разделять запятыми;
- правила синтаксиса, характерные для определенных типов данных, нужно соблюдать внутри списка. Так, если у строки должны быть кавычки, то их нужно использовать и внутри списка, а для чисел и значений булева типа их не нужно использовать.

Пример объявления и вывода:

```
1  list = [3,5,67,8,1]
2  print(list, end='\n\n')
3  print(type(list))
```

Индексация происходит аналогично строкам

```
1  list = [3,5,67,8,1]
2  print(list[0], list[-1])
```

Основные операции:

```
list = [3,5,67,8,1]
list.append (22) # добавить в конец 22
list.insert(0,23) # добавить 23 перед индексом 0
list.remove (22) # удалить 22
list.pop(0) # удалить и вернуть элемент
list.reverse() # развернуть список
list.count(22) # количество вхождений элемента 22
len(list) # количество всех значений
max(list) # получить максимальное значение
min(list) # получить минимальное значение
sum(list) # получить сумму всех значений
```

Кортеж

Кортеж похож на список, но имеет одно важное отличие – он неизменяем. Кортеж целесообразно использовать, когда наверняка известно, что набор данных не изменится в процессе работы программы. Кортеж может состоять из данных разных типов и использовать индексы, которые определяют конкретный порядок элементов.

В отличие от списков кортежи создаются с помощью круглых скобок.

```
1 p_tup = ("Лондон", "Пекин", 44, True)
2 print(p_tup, p_tup[-1])
```

Вывод:

('Лондон', 'Пекин', 44, True) True

Словарь

Словарь состоит из пар «ключ – значение», которые разделяются запятыми. По аналогии со списком его можно называть последовательностью данных.

В отличие от списков и кортежей у словарей нет определенного порядка. Такая структура нацелена на увеличение производительности и предполагает доступ к значению по ключу. В качестве ключей используются только неизменяемые объекты, а в качестве значений можно использовать как неизменяемые объекты, так и изменяемые.

Объявляются словари при помощи фигурных скобок **{}**. Ключ и значение разделяются двоеточием, а пары «ключ – значение» отделяются запятыми друг от друга.

```
1    p_ages = ("Андрей": 32, "Виктор": 29, "Максим": 22)
2    print(p_ages["Максим"] # вывод значения по ключу
3    print(list(p_ages.keys()) [list(p_ages.values()).index(29)]) # вывод ключа по значению
```

Вывод:

22
Виктор

Приведем некоторые из методов и функций словарей:

- **.keys()** используется для получения кортежей с ключами словаря;
- **.values()** используется для получения кортежей со значениями каждого ключа словаря;
- **.items()** используется для создания кортежей с ключами и значениями;
- **.get()** используется для получения значения по ключу.

Примечание. Ключи в словаре могут быть только строками, целыми числами или числами с плавающей точкой. Значения могут быть любого типа.

2. Основные понятия анализа данных и машинного обучения

Машинное обучение (МО) представляет собой методы, позволяющие настраивать алгоритмы для решения конкретных задач путем обучения на основе данных, при помощи которых необходимо вывести новые знания, а также получить новые взаимосвязи в них (часто неявные) или доказать, что их нет. Какие цели можно достичь?

1. На основе полученных зависимостей можно построить прогноз для заданной величины. Например, по данным об урожае картофеля за 10 лет в данном регионе прогнозировать урожайность в следующем году.

2. Классифицировать объекты на основе данных о них. Например, имея данные о применении различных методик обучения, классифицировать методику обучения как эффективную или неэффективную.

3. Визуализация данных помогает выбрать стратегию анализа данных. Иногда визуализация данных сама является целью исследования.

4. Поиск новых зависимостей в данных помогает прийти к новым знаниям о предмете исследования.

Решение задачи машинного обучения состоит из нескольких этапов.

1. **Получение данных.** Нужно собрать данные, на которых впоследствии модель будет обучаться.

2. **Исследовательский анализ данных.** Данные могут содержать неточности и шумы, поэтому, прежде чем переходить к созданию модели машинного обучения, нужно специальным образом подготовить эти данные. На этом этапе данные исследуются на предмет скрытых закономерностей.

3. **Подготовка факторов.** В машинном обучении характеристики объекта, которые используются для его описания, принято называть факторами. Не все факторы могут быть полезны при построении модели машинного обучения, на этом этапе отбирают нужные факторы.

4. **Создание модели МО.** Нужно выбрать модель, которая лучше всего подходит для решения поставленной задачи, а затем обучить ее на какой-то части обучающей выборки и проверить, насколько хорошо модель ведет себя на другой части обучающей выборки. Если модель ведет себя неоптимально, то можно поменять какие-то ее параметры и повторить процесс обучения.

5. **Оценка качества.** Нужно оценить итоговое качество работы модели на тестовой выборке – на данных, которые модель еще не получала.

В машинном обучении выделяют два основных подхода: обучение с учителем (Supervised Learning) и обучение без учителя (Unsupervised Learning). Обучение с учителем – это один из видов машинного обучения, при котором модель обучается на основе предоставленных ей данных, содержащих правильные ответы (метки). Таким образом, модель должна научиться предсказывать правильный ответ для новых данных, которых она не получала ранее. Обучение без учителя – это подход, при котором система обучается на основе неразмеченных данных, т. е. без предоставления правильных ответов (меток). Задача

модели заключается в том, чтобы самостоятельно выявить закономерности и структуру в данных.

Основными являются следующие виды задач машинного обучения:

- задачи классификации;
- задачи регрессии;
- задачи ранжирования.

Задача классификации: входные данные требуется отнести к определенному классу (например, изображения разделить на кошек и собак, распознать произнесенное слово, по медицинским данным поставить диагноз и т. д.). Классификация может быть бинарной (разделение данных на два класса) и многоклассовой (разделение данных на несколько классов).

Задача регрессии: нужно сделать прогноз в виде чисел на основе входных данных (например, оценка курса валют по предыдущим показаниям, прогноз объема продаж товара определенного вида, построение прогноза некоторой функции по измеренным эмпирическим данным и т. д.). В отличие от классификации, где выходные данные дискретны, регрессия выдает непрерывные числовые значения.

Задачи ранжирования относятся к задачам, где необходимо упорядочить элементы набора данных по их значимости или релевантности. Например, в поисковой системе задача ранжирования заключается в определении порядка отображения результатов поиска, чтобы пользователи видели наиболее релевантные результаты в начале списка. Эта задача также актуальна в рекомендательных системах, где нужно упорядочить предложенные элементы для пользователя по степени их вероятной полезности или интересности. Результатом выполнения задачи ранжирования является ранжированный список элементов в соответствии с их значением или релевантностью для конкретной задачи.

Обычно для решения всех этих задач применяют алгоритмы машинного обучения. Это связано с тем, что входные данные обычно имеют сложную структуру, требующую разработки нетривиальных критериев для их обработки. Человеку зачастую сложно самостоятельно создать эти критерии, поэтому необходимо использовать метод обучения на основе обучающей выборки.

Обучающая выборка представляет собой исходные данные, полученные из результатов измерений. Это могут быть изображения, звуковые сигналы, данные о росте, весе, возрасте человека и др. Все эти данные представлены в виде числовых значений:

$$x_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \dots \\ x_{in} \end{bmatrix} = \left[x_{i1}, x_{i2}, \dots, x_{in}\right]^T,$$

где i – номер объекта; вектор $\left[x_{i1}, x_{i2}, \dots, x_{in}\right]^T$ представляет измерение i-го объекта.

Далее можно объединить все эти наборы в матрицу:

$$X = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \dots \\ x_l \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots \\ x_{l1} & x_{l2} & \dots & x_{\ln} \end{bmatrix}^T .$$

Матрица X состоит из l векторов, каждый из которых содержит n координат. Добавим выходные данные – целевые значения (target):

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_{\mathrm{n}} \end{bmatrix} = \left[y_1, y_2, ..., y_n \right]^T .$$

В задачах классификации выходные данные могут принимать одно из следующих значений:

- $y \in \{-1; +1\}$ – при бинарной классификации;
- $y \in \{1; 2; ...; M\}$ – при многоклассовой классификации (классы не пересекаются);
- $y \in \{0; 1\}^M$ – при многоклассовой классификации с пересекающимися классами (в данном случае M – это размерность пространства).

В задачах регрессии каждый вход связан с одним или несколькими вещественными числами:

- $y \in R$ – одномерная задача регрессии;
- $y \in R^m$ – m-мерная задача регрессии.

В задачах ранжирования Y – некоторое упорядоченное множество значений.

Размеченные данные представляют собой совокупность данных $X^l = \left\{ (x_i; y_i)_{i=1}^l \right\}$, которая может использоваться в качестве обучающей выборки, поскольку каждому входному вектору (наблюдению) x_i соответствует значение или вектор y_i.

Алгоритм обучения $a(X^l)$ – функция перехода из пространства X^l в пространство Y. Алгоритм может ошибиться и дать неверный ответ. Поэтому необходимо иметь инструмент оценки качества алгоритма. Таким инструментом является функционал ошибки. Функционал ошибки $Q(a, X^l)$ – ошибка алгоритма a на выборке X^l.

Рассмотрим функционал ошибки

$$Q(a, X^l) = \frac{1}{l}\sum_{i=1}^{l} L(a, x_i),$$

зависящий от модели $a(x)$ и вида функции потерь $L(a, x)$. Задача состоит в том, чтобы выбрать наилучшую модель, т. е. чтобы функционал принимал наименьшее значение на обучающей выборке:

$$\mu(X^l) = \arg\min_{a \in A} Q(a, X^l),$$

где $\mu(X^l)$ – найденная модель на этапе обучения по всем возможным моделям $a \in A$; arg min – определяет значение аргумента, при котором функция принимает наименьшее значение.

В случае параметрических моделей $a(x) = g(x, \theta)$ это выражение может быть записано относительно вектора параметров θ:

$$\mu(X^l) = \arg\min_{\theta} Q(a, X^l).$$

После обучения принимаем $a = \mu(X^l)$ и используем найденную зависимость при практической реализации.

Рассмотрим **метод наименьших квадратов** (в качестве функции потерь рассматривается квадратичная функция). Здесь данные формируются по закону

$$y_i = kx_i + b + \varepsilon_i, \ i = 1, 2, ...,$$

где ε_i – гауссовский шум с нулевым средним.

Для такой задачи оптимальная модель имеет вид линейной функции:

$$a(x) = \bar{k}x + \bar{b}.$$

Далее определим неизвестные параметры $\bar{k}$, $\bar{b}$. Для этого составим функцию двух переменных и найдем, при каких значениях $\bar{k}$, $\bar{b}$ эта функция принимает минимальное значение:

$$Q(a, X^l) = \frac{1}{l}\sum_{i=1}^{l} [a_i(x_i) - y(x_i)]^2 \to \min.$$

Получим следующую систему уравнений:

$$\begin{cases} \dfrac{\partial Q}{\partial \bar{k}} = \dfrac{2}{l}\sum\limits_{i=1}^{l}\left(\bar{k}x_i + \bar{b} - y_i\right)\cdot x_i = 0; \\ \dfrac{\partial Q}{\partial \bar{b}} = \dfrac{2}{l}\sum\limits_{i=1}^{l}\left(\bar{k}x_i + \bar{b} - y_i\right) = 0; \end{cases}$$

$$\begin{cases} \bar{k}\sum\limits_{i=1}^{l} x_i^2 + \bar{b}\sum\limits_{i=1}^{l} x_i = \sum\limits_{i=1}^{l} y_i \cdot x_i; \\ \bar{k}\sum\limits_{i=1}^{l} x_i + \bar{b}\,n = \sum\limits_{i=1}^{l} y_i. \end{cases}$$

Решая ее, делаем вывод: функция $a(x) = \bar{k}x + \bar{b}$ наилучшим образом приближает экспериментальные точки.

3. Переобучение модели

Модель обучается на данных, где она изучает зависимости между входными признаками и выходными значениями. В результате обучения может получиться модель, которая очень хорошо описывает именно ту выборку, на которой она обучалась. Она может начать запоминать шумы и специфические особенности этого набора данных вместо того, чтобы обобщать общие закономерности. Стоит дать модели новый объект, не входящий в обучающую выборку, она даст ответ с большой ошибкой. Этот факт свидетельствует о *переобучении* модели (рис. 1).

Например, если задача состоит в предсказании роста человека по его возрасту, переобученная модель может точно предсказывать рост для людей из обучающего набора, но давать неточные прогнозы для людей с разным ростом. Хорошо обученная модель делает менее точные, но более обобщенные прогнозы, которые ближе к реальности для всех людей, независимо от их характеристик.

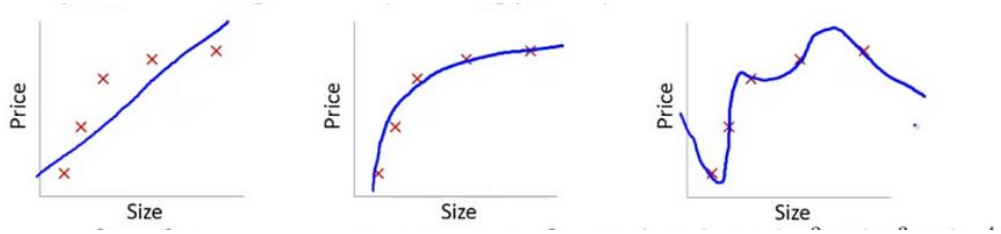


Рис. 1. Недообученная (слева), обученная (в центре), переобученная (справа) модели

Существуют признаки переобученности модели:

- малая ошибка на обучающей выборке и большая на реальном объекте;
- аномально большие веса модели.

Рассмотрим методы борьбы с переобучением.

3.1. Отложенная выборка

Имеющуюся выборку разбивают на две неравные части – обучающую выборку (около 70 % объектов) и тестовую (около 30 % объектов). Обучение модели проводится на обучающей выборке, а проверка ее качества – на тестовой. Возникает проблема разбиения выборки: если к обучающей части отнести первые 70 % выборки, то может получиться так, что некоторые уникальные объекты не попадут в обучающую выборку и модель не обучится на них. Следовательно, когда модель их встретит в тестовой части, она даст большую ошибку. Чтобы этого избежать, можно выбирать в тестовую выборку случайные объекты из выборки, а оставшиеся считать обучающей выборкой.

3.2. Кросс-валидация

Перекрестная проверка (или кросс-валидация) – это такой подход в разделении выборки, при котором все объекты попадут в обучающую часть выборки. Выборку делят на k равных частей – фолдов. Поэтому этот метод часто называют k-fold-перекрестной проверкой. Далее модель обучают k раз, причем каждый раз выделяется одна часть как тестовая, а остальные части используются для обучения модели. Таким образом, каждая часть данных используется как часть тестового набора данных и как часть обучающего набора данных.

В результате модель обучается k раз, и, следовательно, вычисляется k раз ошибка модели. Итоговой ошибкой модели становится средняя ошибка по всем оценкам. Достоинство этого метода – невозможность пропустить объект мимо обучающей выборки, недостаток – нужно много раз обучать модель.

Заметим, что проводить любую предобработку данных, включая настройку гиперпараметров модели, нужно на обучающем наборе данных внутри цикла кросс-валидации, чтобы избежать утечки информации и оптимистических оценок качества модели.

Выбор оптимального значения параметра k в кросс-валидации имеет решающее значение для точности оценки качества модели. Неправильно выбранное значение k может привести к оценкам модели с высокой дисперсией или предвзятостью.

Рассмотрим три основных подхода к выбору значения k.

1. Представительный подход. Значение k выбирается так, чтобы каждая группа (обучающая/тестовая часть) данных была статистически репрезентативной для более широкого набора данных. Это обеспечивает адекватность оценки качества модели.

2. Фиксированное значение k. Значение выбирается на основе экспериментов и обычно обеспечивает низкую предвзятость и небольшую дисперсию оценки качества модели.

3. Фиксированное значение $k = n$, где n – размер всего набора данных. Это дает возможность каждой тестовой части быть использованной в наборе дан-

ных. Такой подход обычно используется в случае, когда данных очень много и вычислительные ресурсы позволяют осуществить его.

Рассмотрим пример. Имеется набор данных, состоящий из шести наблюдений: [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]. Используем метод кросс-валидации с параметром $k = 3$. Это предполагает разделение данных на три группы, содержащие по два наблюдения в каждой.

Процесс кросс-валидации выглядит следующим образом:

1. Производится перемешивание данных и их последующее разделение на три группы:
   - Группа 1: [0.5, 0.2];
   - Группа 2: [0.1, 0.3];
   - Группа 3: [0.4, 0.6].
2. Проводится обучение и тестирование трех моделей по следующей схеме:
   - Модель 1: обучается на данных из Групп 1 и 2, после чего производится ее оценка на данных из Группы 3;
   - Модель 2: обучается на данных из Групп 2 и 3, а затем оценивается на данных из Группы 1;
   - Модель 3: обучается на данных из Групп 1 и 3 с последующей оценкой на данных из Группы 2.
3. После проведения оценки каждой модели собираются и анализируются их оценки качества с целью принятия решения о качестве алгоритма машинного обучения.

Автоматизировать этот процесс можно с помощью метода **KFold()** (библиотека *scikit-learn*). Он принимает на вход количество групп, на которые нужно разделить датасет, флаг, указывающий, нужно ли перетасовать данные перед разделением, и числовую затравку для псевдослучайного генератора чисел, используемого до перетасовки датасета.

Например, создадим экземпляр, который разделит набор данных на три группы, предварительно перемешав их, и использует значение **1** в качестве затравки для генератора псевдослучайных чисел:

```
kfold = KFold(3, shuffle=True, random_state=1)
```

Затем можно вызвать функцию **split()** на объекте **kfold**, передавая ей на вход выборку данных. При каждом вызове **split()** будут возвращаться индексы обучающей и тестовой выборок. Возвращаемые массивы содержат индексы исходного датасета, указывая на соответствующие образцы в каждой группе на каждой итерации.

Например, мы можем получить разделение индексов для выборки данных, используя созданный экземпляр **kfold**, следующим образом:

```
# перечисление выборок датасета
for train, test in kfold.split(data):
        print('train: %s, test: %s' % (train, test))
```

```
#scikit-learn k-fold кросс-валидация
from numpy import arrayform
sklearn.model_selection import KFold

# датасет
data = array ([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])

# подготовьте кросс-валидацию
kfold = KFold(3, True, 1)

# перечисление выборок датасета
for train, test in kfold.split(data):
        print('train: %s, test: %s' % (data[train], data [test]))
```

В приведенном примере выводятся конкретные наблюдения, которые были выбраны для каждого обучающего и тестового наборов данных. Индексы используются непосредственно в исходном массиве данных для доступа к значениям наблюдений:

train: [0.1, 0.4, 0.5, 0.6], test: [0.2, 0.3]
train: [0.2, 0.3, 0.4, 0.6], test: [0.1, 0.5]
train: [0.1, 0.2, 0.3, 0.5], test: [0.4, 0.6]

Перекрестная проверка может быть использована непосредственно для разделения набора данных перед моделированием. Это полезно в случае крупных наборов данных. Также реализация перекрестной проверки в библиотеке *scikit-learn* предоставляется как компонентная операция в рамках более общих методов, таких как поиск по сетке гиперпараметров и оценка моделей на наборе данных.

3.3. Регуляризация

Еще один способ борьбы с переобучением модели – регуляризация. При переобучении ошибка на обучающей выборке слишком маленькая. Чтобы избежать данной ситуации, можно добавить к функционалу ошибки некоторое слагаемое и затем найти минимум уже общей суммы:

$$Q(a, X) + \lambda \|\omega\| \to \min,$$

где λ – коэффициент регуляризатора; ω – веса модели.

В этом и заключается регуляризация. В алгоритмах машинного обучения используются два типа регуляризатора:

1. L1-регуляризатор – Lasso (Least Absolute Shrinkage and Selection Operator):

$$\|\omega\|_l = \sum_{i=1}^{d} |\omega_i|.$$

L1-регуляризатор добавляет в функцию потерь штраф, равный сумме абсолютных значений коэффициентов модели, и штрафует за сложность модели, т. е. за высокие значения весов. L1-регуляризатор можно применять для отбора признаков, поскольку он может обнулить веса малозначимых параметров модели. Это позволит исключить неинформативные признаки, что в свою очередь снизит сложность модели и улучшит ее обобщающую способность.

Заметим, что при использовании L1-регуляризатора могут возникать некоторые проблемы. Основная проблема заключается в том, что L1-регуляризация создает острые углы или разрывы около нуля, где производная функции потерь не определена. Это затрудняет вычисление градиента функции потерь, что может привести к неэффективной работе метода градиентного спуска.

Также важен выбор оптимального значения коэффициента регуляризации λ. Если значение λ слишком мало, то веса могут оставаться большими, и это приведет к переобучению модели. Если же значение λ слишком велико, то веса станут слишком маленькими, что приведет к недообучению модели.

2. L2-регуляризатор – Ridge:

$$\|\omega\|^2 = \sum_{i=1}^{d} \omega_i{}^2.$$

Этот регуляризатор не обнуляет веса, а только снижает их значения. Можно сказать, что L2-регуляризатор не штрафует, а скорее сглаживает пики, добавляя сумму квадратов весов для уменьшения их влияния. Он «выпрямляет» функцию активации, приближая ее к линейной форме, тем самым предотвращает переобучение.

Рассмотрим выражение

$$Q(\omega, X) + \lambda \cdot \|\omega\|^2 \to \min,$$

где λ – коэффициент регуляризации, позволяющий менять квадраты нормы весов.

Если значение λ велико, то второе слагаемое в функции потерь становится большим, что приводит к увеличению общей ошибки модели. Это происходит из-за того, что большое значение λ приводит к большому количеству штрафов, что делает модель недообученной. С другой стороны, если значение λ слишком маленькое, второе слагаемое также становится маленьким, а это может привести к тому, что модель будет делать очень большие переобученные веса. Поэтому для наилучшей регуляризации необходимо выбирать коэффициент исходя из метрик качества модели.

Пример кода L1-регуляризации

Приведем код, демонстрирующий использование L1-регуляризации. Сгенерируем данные, где входные переменные будут представлены широкой матрицей, зависимая переменная будет зависеть только от нескольких факторов, остальные факторы будут представлять собой шум. Затем применим L1-регуляризацию, чтобы найти разреженные весовые коэффициенты, которые определяют полезные измерения **X**.

Импортируем библиотеки NumPy и Matplotlib:

```
import numpy as np
import matplotlib.pyplot as plt
```

Определим сигмоидную функцию:

```
def sigmoid(z):
return 1 / (1 + np.exp(-z))
```

Установим **N = 50** и **D = 50**, чтобы это была широкая матрица. Значения **X** установим равномерно распределенными в диапазоне от –5 до +5:

```
N = 50
D = 50
X = (np.random.random((N,D)) – 0.5)*10
```

Истинные значения весовых коэффициентов, определяемые переменной **true_w**, установим равными **1**; **0,5** и **–0,5**, так что только первые три размерности имеют значение, остальные же 47 размерностей установим равными нулю, никак не влияющими на результат:

```
true_w = np.array([1, 0.5, -0.5] + [0]*(D-3))
```

Теперь определим **Y**. Это будет сигмоида от **X** плюс некоторый случайный шум:

```
Y = np.round(sigmoid(X.dot(true_w) + np.random.randn(N)*0.5))
```

Коэффициент обучения установим равным **0,001**, штраф при L1-регуляризации установим равным **2**, количество итераций – **5000**. Кроме того, рассчитаем значение функции затрат:

```
costs[]
w = np.random.randn(D) / np.sqrt(D)
learning_rate = 0.001
l1 = 2.0
for t in xrange(5000):
Yhat = sigmoid(X.dot(w))
delta = Yhat – Y
w = w – learning_rate*(X.T.dot(delta) + l1*np.sign(w))
cost = -(Y*np.log(Yhat) + (1-Y)*np.log(1 – Yhat)). mean() + l1*np.abs(w).mean()
costs.append(cost)
plt.plot(costs)
plt.show()
```

Выведем графики истинных весовых коэффициентов и вычисленных весовых коэффициентов:

```
plt.plot(true_w, label='true w')
plt.plot(w, label='w map')
plt.legend()
plt.show()
```

Результат работы программы представлен на рис. 2.

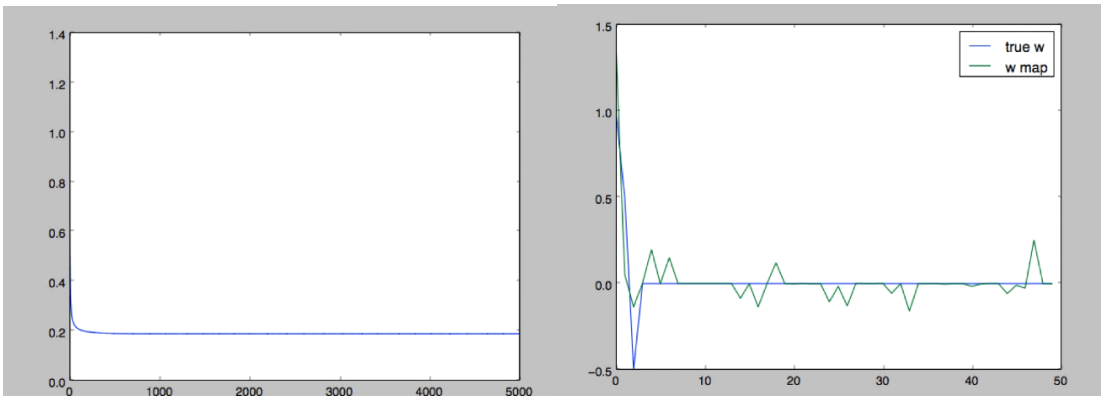


Рис. 2. Результат L1-регуляризации (**L = 2**)

Как видно из рис. 2, функция стоимости быстро сходится. Полученные значения весовых коэффициентов довольно хороши, но они не совпадают полностью с истинными значениями.

Запустим программу еще раз, установив штраф L1-регуляризатора равным 10:

l1 = 10.0

Получим новую функцию затрат и весовые коэффициенты (рис. 3).

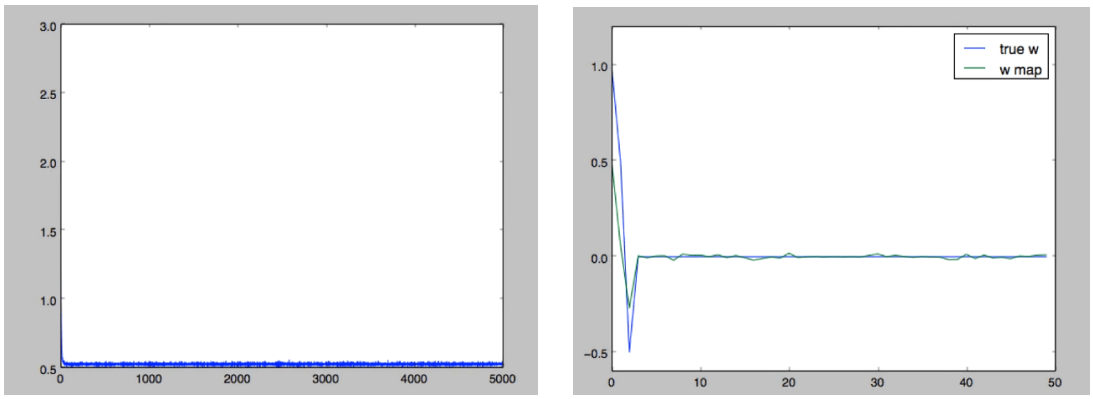


Рис. 3. Результат L1-регуляризации (**L = 10**)

Коэффициенты во втором случае намного ближе к нулю, поскольку сдвинуты в эту область регуляризацией. Следовательно, нужно выставлять меньший штраф регуляризации, чтобы достичь положительного результата.

Пример кода L2-регуляризации

Подключим библиотеки NumPy и Matplotlib.pyplot:

```
import numpy as np
import matplotlib.pyplot as plt
```

Запишем функцию в виде полинома:

```
x=np.arange(0,10.1,0.1)
y=np.array([-a**4+100*a**2+a for a in x])
```

Укажем размер признакого пространства (степень полинома **N – 1**).

```
x_train, y_train = x[::2], y[::2]
N=13
```

Первоначально выставим коэффициент λ равным **0**. Далее мы его будем менять до того значения, которое полностью будет соответствовать графику:

```
L=0
```

Формируем матрицу входных векторов:

```
X = np.array([[a**n for n in range(N)] for a in x])
```

Запишем матрицу λI из выражения

$$\omega_* = \left(X^{\mathrm{T}} \cdot X \middle| \lambda \cdot I \right)^{-1} \cdot X^{\mathrm{T}} \cdot Y.$$

```
IL = np.array([[L if i==j else 0 for j in range(N)] for i in range(N)])
#print(IL)
```

Первый коэффициент не регуляризуем:

```
IL[0][0] = 0
```

В матрице **X** оставляем только четные элементы:

```
X_train = X[::2]
```

Определяем вектор **Y**:

```
Y = y_train
```

Вычисляем вектор коэффициентов исходя из формулы (ω_*):

```
A= np.linalg.inv(X_train.T @ X_train+IL)
w= Y @ X_train @ A
print(w)
```

Отображаем график:

```
yy = [np.dot(w,x) for x in X]
```

Прогноз модели:

```
plt.plot(x,yy)
```

Истинное поведение модели:

```
plt.plot(x,y)
plt.grid(True)
plt.show()
```

Запускаем программу с **L = 0** (рис. 4).

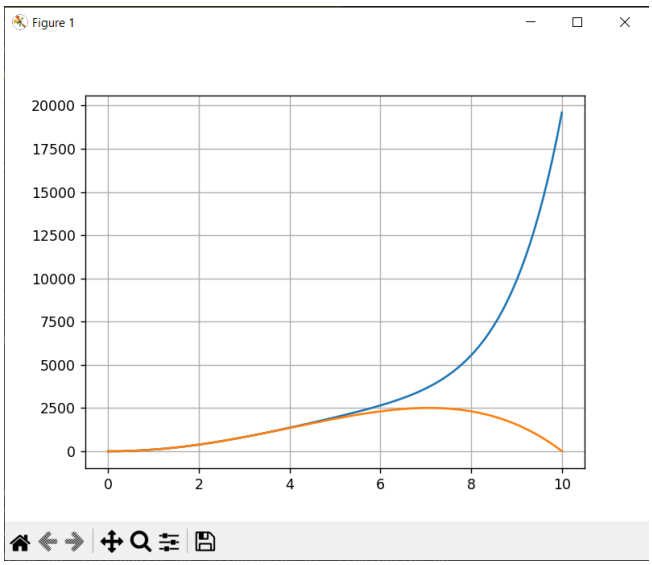


Рис. 4. Результат L2-регуляризации (**L = 0**)

Теперь поставим **L = 10** (рис. 5).

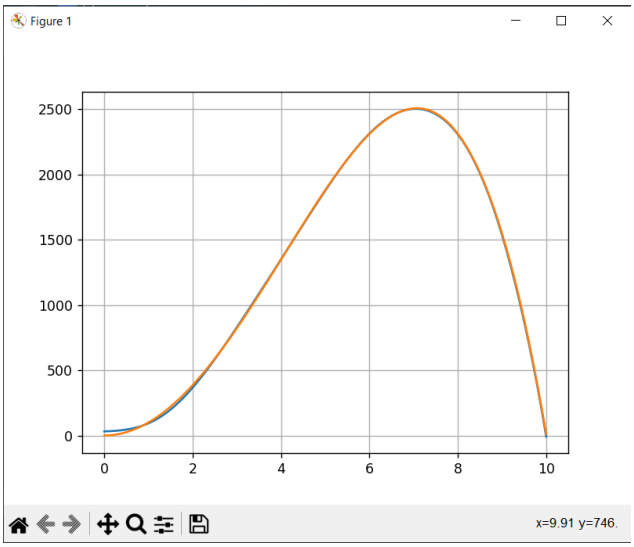


Рис. 5. Результат L2-регуляризации (**L = 10**)

Из рис. 5 видно, что прогноз достаточно похож на поведение реальной модели.

4. Метрики для оценки качества модели

Рассмотрим основные метрики для различных задач машинного обучения.

В **задачах классификации** модель должна предсказать категориальную метку для данного набора признаков. Приведем основные метрики, используемые для оценки качества модели классификации.

Accuracy (доля правильных ответов)

Доля правильных ответов *accuracy*(*a*, *X*) – наиболее популярная метрика качества классификационных моделей. Она указывает долю правильно классифицированных объектов от общего числа объектов, т. е. тех ответов алгоритма, которые совпали с истинными значениями в обучающей или тестовой части выборки:

$$accuracy(a, X) = \frac{1}{n}\sum_{i=1}^{n}[a(x_i) = y_i].$$

Например, если есть 100 объектов и модель правильно классифицировала 95 из них, то точность составит 0,95, или 95 %.

Заметим, что если в выборке много объектов класса 0 и мало объектов класса 1, то модель будет хорошо определять объекты класса 0, а с объектами класса 1 будет ошибаться. И если поменять разметку классов на обратную, то доля правильных ответов резко снизится.

Матрица ошибок

Ошибки классификации бывают двух типов: False Negative (FN) и False Positive (FP) (табл. 3).

Табл. 3. Ошибки классификации

Предсказание	Факт	
	$y = 1$	$y = -1$
$a(x) = 1$	TP	FP
$a(x) = -1$	FN	TN

FN (False Negative) – количество объектов, которые на самом деле относятся к классу –1, но алгоритм отнес их к классу 1.

FP (False Positive) – количество объектов, которые на самом деле являются классом 1, но алгоритм их отнес к классу –1.

Матрица ошибок содержит и правильные ответы, которые также делятся на две группы: TP (True Positive) и TN (True Negative).

TP – количество объектов класса 1, которые нашла модель.

TN – количество объектов класса –1 или 0, которые нашла модель.

Матрица ошибок дает информацию не только о том, сколько ошибок делает алгоритм, но и позволяет понять, насколько он точен и как полно находит заданный класс. С этой целью вводятся понятия «точность» (precision) и «полнота» (recall). Точность показывает, насколько можно доверять классификатору. Классификатор с высокой точностью поместит в класс 1 только объекты этого класса.

Precision (точность) – это доля истинно положительных примеров среди всех примеров, классифицированных как положительные. То есть если модель предсказала, что 10 примеров положительны и из них только 9 действительно положительны, то точность составит 0,9, или 90 %.

Вычисляется точность по формуле

$$precision(a, X) = \frac{TP}{TP + FP}.$$

Точность – это отношение количества правильно отнесенных к определенному классу объектов к количеству всех объектов этого класса в выборке.

Recall (полнота) – это доля истинно положительных примеров среди всех реальных положительных примеров. То есть если в реальности у нас 20 положительных примеров и модель правильно определила только 15 из них, то полнота составит 0,75, или 75 %.

Полнота показывает, как хорошо классификатор находит объекты заданного класса. Классификатор с высокой полнотой найдет все объекты класса 1 в выборке, но при этом прихватит в класс 1 и много объектов из класса –1.

Вычисляется полнота по формуле

$$recall(a, X) = \frac{TP}{TP + FN}.$$

Полнота – это отношение количества правильно отнесенных к определенному классу объектов к количеству всех объектов, отнесенных классификатором к этому классу.

Настроить алгоритм на высокие и точность, и полноту, как правило, невозможно. Необходимо что-то выбирать. Выбор определяется конкретной задачей, в которой нужно понять, что важнее – найти все объекты класса или найти объекты только одного класса. Компромиссом между этими мерами является метрика F1-score.

F1-score (F-мера)

F1-score вычисляется по формуле

$$F = \frac{2 \cdot precision \cdot recall}{precision + recall}.$$

F1-score – это среднее гармоническое между точностью и полнотой. Эта метрика полезна в случаях несбалансированных классов, когда один класс представлен значительно большим количеством элементов, чем другой.

AUC-ROC-кривая

Еще одной популярной метрикой качества классификации является ***AUC-ROC-кривая*** *(Area Under the Receiver Operating Characteristic Curve).* AUC-ROC-кривая строится в координатах TPR и FPR соответственно:

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}, \quad \mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}.$$

Вид AUC-ROC-кривой представлен на рис. 6.

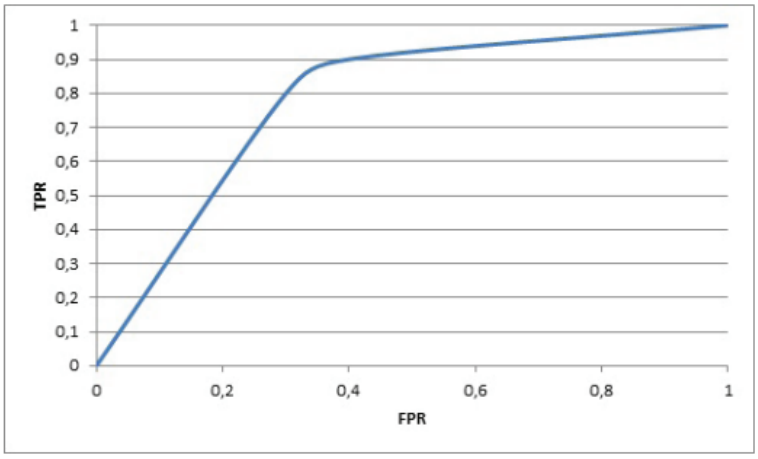


Рис. 6. AUC-ROC-кривая

Качество классификатора оценивается площадью под AUC-ROC-кривой: чем она ближе к единице, тем лучше классификатор. График ROC-AUC-кривой для идеального классификатора представлен на рис. 7.

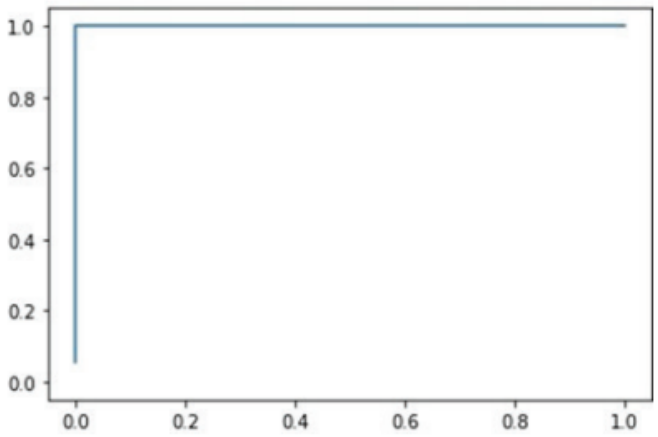


Рис. 7. AUC-ROC-кривая идеального классификатора

Метрика AUC-ROC измеряет качество ранжирования предсказаний модели, учитывая различные пороговые значения классификации. Значение AUC-ROC 1.0 соответствует идеальной модели классификации, в то время как значение 0.5 соответствует случайной модели.

В **задачах регрессии** модель должна предсказать непрерывное значение на основе данного набора признаков.

Ниже приведены основные метрики, используемые для оценки качества модели регрессии.

Mean Squared Error

Mean Squared Error (MSE, среднеквадратичная ошибка) – это среднее значение квадратов разности между истинными и предсказанными значениями:

$$\mathrm{MSE}(a, X) = \frac{1}{n}\sum_{i=1}^{n}\left[a(x_i) - y_i\right]^2 ,$$

где n – количество наблюдений, по которым строится модель и количество прогнозов.

MSE применяется в тех случаях, когда требуется подчеркнуть большие ошибки и выбрать модель, которая дает меньше именно больших ошибок. Чем меньше MSE, тем лучше модель. График среднеквадратичной ошибки принимает вид параболы (рис. 8). Среднеквадратичная ошибка – гладкая непрерывная функция, поэтому она может быть использована для дифференцирования.

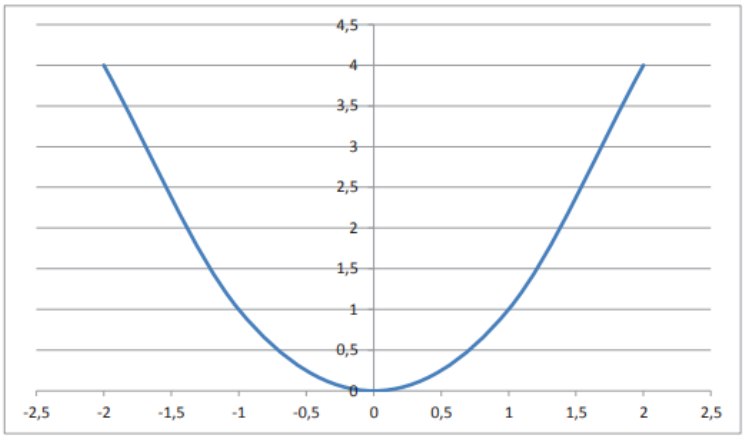


Рис. 8. График среднеквадратичной ошибки

Недостатком использования MSE является то, что если на одном или нескольких неудачных примерах, возможно, содержащих аномальные значения, будет допущена значительная ошибка, то возведение в квадрат приведет к ложному выводу, что вся модель работает плохо.

Root Mean Squared Error

Root Mean Squared Error (RMSE, корень из среднеквадратичной ошибки) – это квадратный корень из MSE:

$$\mathrm{RMSE}(a, X) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}[a(x_i) - y_1]^2}.$$

RMSE более интерпретируем, поскольку ошибка измеряется в тех же единицах, что и исходный целевой признак.

Mean Absolute Error

Mean Absolute Error (MAE, средняя абсолютная ошибка) – это среднее значение абсолютных разностей между истинными и предсказанными значениями:

$$\mathrm{MAE}(a, X) = \frac{1}{d}\sum_{i=1}^{d}|a(x_i) - y_1|.$$

Чем меньше MAE, тем лучше модель. Как видно из рис. 9, абсолютную ошибку нельзя дифференцировать.

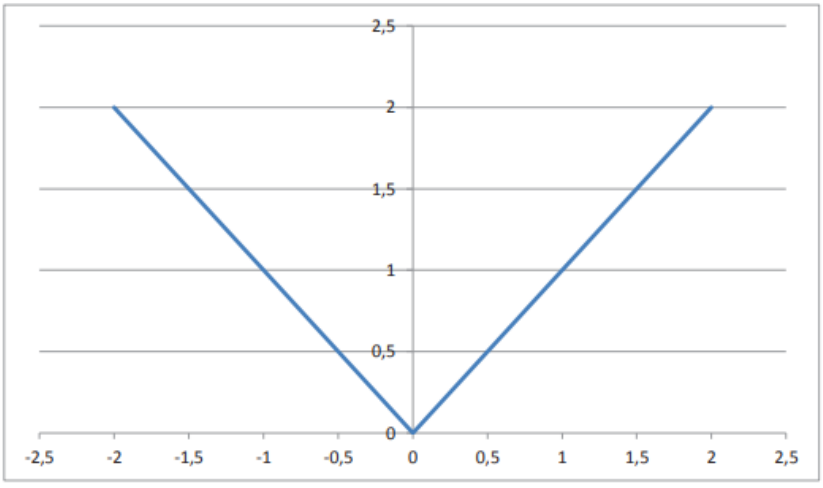


Рис. 9. График абсолютной ошибки

R-squared (R-квадрат)

R-squared (коэффициент детерминации) – это доля дисперсии целевого признака, объясненная моделью

$$R^2(a, X) = 1 - \frac{\sum_{i=1}^{d} [a(x_i) - y_i]^2}{\sum_{i=1}^{d} (y_i - \bar{y})},$$

где $\bar{y}$ – среднее значение ответов на выборке.

R-squared всегда находится между 0 и 1. Значение 0 указывает, что модель не объясняет вариацию целевого признака, с другой стороны, значение 1 указывает на то, что модель идеально объясняет данные.

В **задачах ранжирования** модель должна предсказать порядок элементов, а не конкретные классы или значения. Например, рекомендация фильмов пользователю на основе его предпочтений является задачей ранжирования. В задаче ранжирования каждый объект описывается вектором значений характеристических признаков. Задача состоит в том, чтобы отсортировать объекты по некоторому критерию.

Ниже приведены основные метрики, используемые для оценки качества модели ранжирования:

- *Precision at K (точность на K)* – это доля релевантных элементов среди первых *K* позиций ранжирования;
- *Normalized Discounted Cumulative Gain (NDCG, нормализованный дисконтированный кумулятивный выигрыш)*. Эта метрика учитывает порядок релевантных элементов в ранжировании, придавая больший вес более релевантным элементам;
- *Mean Average Precision (MAP, средняя средняя точность)* – это среднее значение средней точности по всем запросам. Средняя точность для каждого запроса – это среднее значение точности на *K* после каждого релевантного элемента в ранжировании.

5. Понижение размерности

Часто наборы данных характеризуются большим количеством признаков, насчитывающих сотни параметров. Понижение размерности сводится к уменьшению числа параметров модели. При этом значимые признаки, т. е. признаки, которые имеют влияние на функционал ошибки модели (а именно уменьшают его), должны быть оставлены. Удаление избыточных признаков способствует лучшему пониманию данных, ускоряет процесс настройки модели, повышает ее точность и улучшает интерпретацию.

Рассмотрим один из подходов к решению задачи понижения размерности – отбор признаков. Существует три основных метода отбора признаков:

- методы фильтров (filter methods);
- методы вложения (embedded methods);
- методы обертывания (wrapper methods).

Выбор конкретного метода зависит от задачи и имеющихся данных.

Методы фильтров (filter methods)

Методы фильтров заключаются в отборе параметров по заданному условию. Например, удаление из модели сильно коррелирующих между собой признаков. Эти методы применяются до обучения модели. К ним относятся:

- визуальный анализ (например, удаление признака, у которого единственное значение или пропущено большинство значений);
- оценка признаков с помощью статистического критерия (дисперсии, корреляции, X^2 и др.);
- экспертная оценка (например, удаление признаков с некорректными значениями или признаков, которые не подходят по смыслу).

Используя библиотеку *pandas-profiling*, можно оценить пригодность признаков, т. е. провести разведочный анализ данных. Также можно применить библиотеку *feature-selector*, которая отбирает признаки по следующим параметрам: коэффициент корреляции (если у признаков коэффициент корреляции больше порогового, то такие признаки удаляются), вариативность (признаки, состоящие из одного значения, удаляются), количество пропущенных значений (удаляются те признаки, у которых процент пропущенных значений больше порогового).

В библиотеке *scikit-learn* реализованы более сложные методы отбора признаков. Например, *VarianceThreshold* отбирает признаки, у которых дисперсия меньше заданного значения. *SelectKBest* и *SelectPercentile* оценивают взаимосвязь признаков с целевой переменной, используя статистические тесты. Они позволяют отобрать заданное количество признаков (*SelectKBest*) и долю наилучших по заданному критерию признаков (*SelectPercentile)*. В качестве статистических тестов используются *F-тест*, X^2 и *взаимная информация*.

F-тест оценивает степень линейной зависимости между признаками и целевой переменной, следовательно, он лучше всего подходит для линейных моделей. В *scikit-learn* он реализован как **f_regression** и **f_classif** для задач регрессии и классификации соответственно.

Статистический тест X^2 применяется в задачах классификации и оценивает зависимость между признаками и классами целевой пременной. Этот тип тестов требует неотрицательных и правильно масштабированных признаков.

Взаимная информация показывает, насколько четко определена целевая переменная, если известны значения признака. Этот тип тестов позволяет находить нелинейные зависимости. Реализован в *scikit-learn* для регрессии и классификации как **mutual_info_regression** и **mutual_info_classif** соответственно.

Методы вложения (embedded methods)

Методы вложения производят отбор признаков во время построения модели, применяя, например, регуляризаторы. Регуляризация Lasso (L1-регулязатор) – один из методов вложения.

L1-регулязатор добавляет штраф (penalty) к различным параметрам модели во избежание чрезмерной подгонки. При регуляризации линейной модели штраф применяется к весовым коэффициентам. L1-регулязатор может обнулить веса малозначимых параметров модели.

Методы обертывания (wrapper methods)

Особенностью методов обертывания является поиск всех возможных подмножеств признаков, оценка их качества путем «прогонки» через модель. Эти методы представлены тремя алгоритмами: полный перебор, «жадный» алгоритм и Add-Del.

В случае полного перебора ошибка считается на всех возможных комбинациях параметров. Выбирается та комбинация, которая дает минимальную ошибку. Этот метод долгий по времени и затратный по ресурсам.

В случае «жадного» алгоритма берется признак, который дает минимальную ошибку модели, после к нему добавляют по одному признаку (рис. 10). Если функционал ошибки уменьшается, то признаки продолжают добавлять, иначе добавление признаков прекращают.

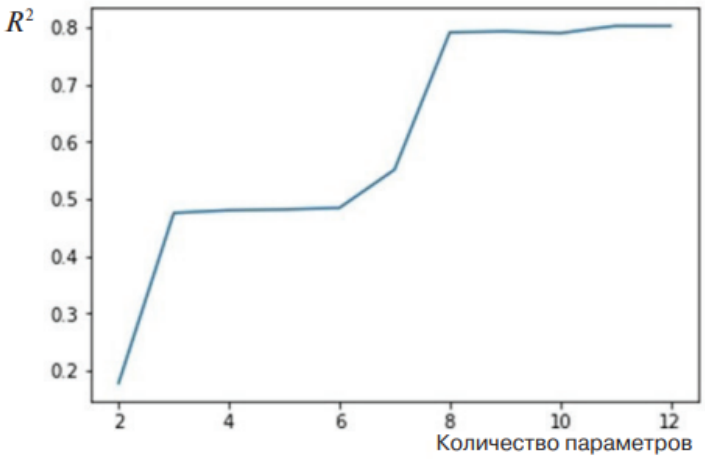


Рис. 10. Пример работы «жадного» алгоритма

С помощью коэффициента детерминации R^2 оценивается точность модели, и, как видно из рис. 10, после 10 параметров точность модели практически не меняется.

Алгоритм Add-Del является модернизацией «жадного» алгоритма. В нем параметр, который увеличивает функционал ошибки после его добавления в модель, удаляется из набора параметров, а в «жадном» алгоритме новые пара-

метры только добавляются. Результат работы алгоритма Add-Del на тех же данных, что и у «жадного» алгоритма, показан на рис. 11.

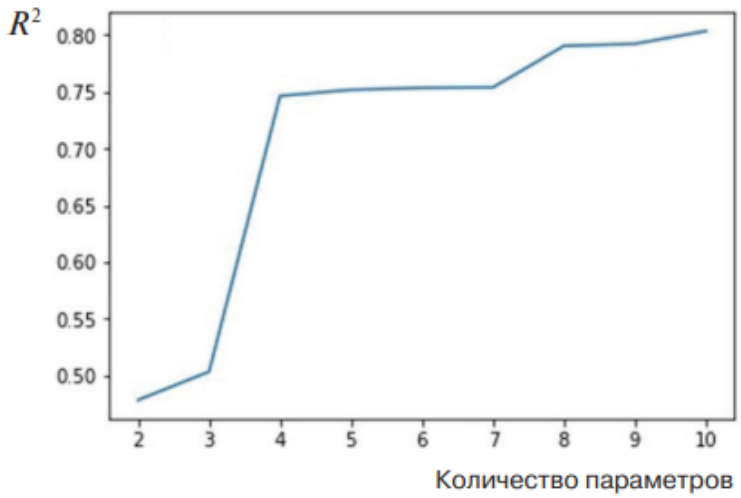


Рис. 11. Пример работы алгоритма Add-Del

6. Метод градиентного спуска

Метод градиентного спуска – это алгоритм оптимизации, который используется для минимизации ошибок в модели машинного обучения, т. е. метод, который находит минимальное значение функции потерь (рис. 12). Возьмем ω_0 – начальную точку градиентного спуска, каждая следующая точка будет выбираться по правилу

$$\omega_{k+1} = \omega_k - \alpha \nabla J(\omega_k),$$

где α – размер шага; ∇ – градиент.

Процесс повторяется много раз, пока алгоритм не сможет предсказать ответ настолько хорошо, насколько это возможно.

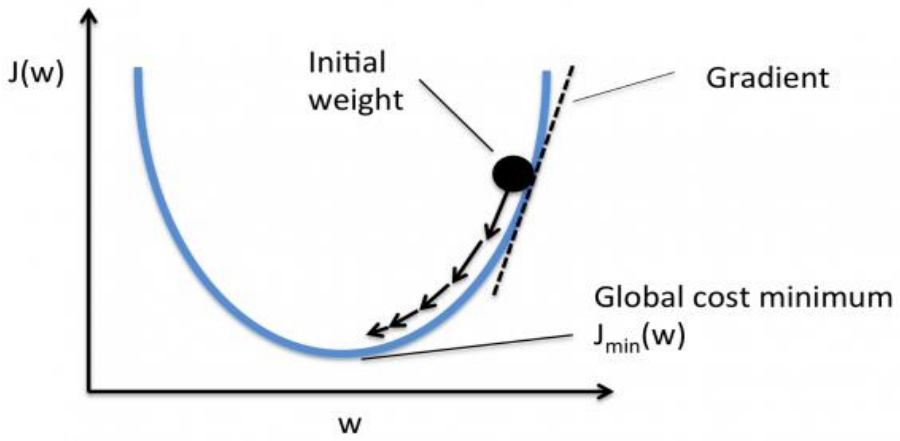


Рис. 12. Функция потерь

Функция потерь предназначена для отслеживания ошибки с каждым шагом обучения, в то время как производная функции потерь относительно одного веса – это то, куда нужно сместить вес, чтобы минимизировать функцию потерь для этого примера обучения.

Рассмотрим пример. Необходимо научить модель предсказывать рост человека по длине его стопы. Есть набор данных с длиной стопы и ростом человека. Программа начинается со случайного предположения о взаимосвязи между данными параметрами: $y = \omega x + b$, где y – рост; x – длина стопы; ω – весовой коэффициент (допустим, она предполагает, что при каждом увеличении длины на 1 сантиметр рост увеличивается на 0,3 метра). Далее для определения функции ошибки возьмем среднеквадратичную ошибку:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - (wx_i + b))^2 .$$

Для обновления параметров ω и b найдем частные производные: $\frac{\partial \text{MSE}}{\partial \omega}$ и $\frac{\partial \text{MSE}}{\partial b}$. Параметры обновляются по следующим формулам: $\omega_{new} = \omega - \eta \cdot \frac{\partial \text{MSE}}{\partial \omega}$, $b_{new} = b - \eta \cdot \frac{\partial \text{MSE}}{\partial b}$, где – η скорость обучения. Продолжаем обновлять параметры до тех пор, пока ошибка не станет достаточно малой или пока не достигнем определенного количества итераций.

```
import numpy as np

# Данные (пример)
foot_length = np.array([22, 24, 26, 28, 30]) height= np.array([150, 160, 170, 180, 190])

# Инициализация параметров
w = 0.3
b = 0
learning_rate = 0.01
num_iterations = 1000

# Функция для вычисления MSE
def compute_mse(y_true, y_pred): return np.mean((y_true y_pred) ** 2)

# Градиентный спуск
for i in range(num_iterations):

# Предсказание
y_pred w foot_length + b

# Вычисление градиентов
dw2 np.mean(foot_length (heighty_pred))
db2 np.mean(height (heighty_pred))
```

```
# Обновление параметров
w learning_rate dw
b = learning_rate * db

# Печать ошибки каждые 100 итераций

if i % 100 == 0:
mse = compute_mse(height, y_pred)
print(f"Итерация (i): MSE {mse}, {w}, b {b}")

# Результаты
print(f"Окончательные параметры: и {w}, b = {b}")
```

7. Логистическая регрессия

Машинное обучение использует теорию вероятности для предсказания и классификации. Можно обучать алгоритмы с помощью статистических закономерностей. Одной из таких относительно простых возможностей является использование теоремы Байеса. Пусть $P(A)$ – априорная вероятность события A; $P(B)$ – априорная вероятность события B; $P(B \mid A)$ – условная вероятность наступления события B при истинности события A. Тогда условная вероятность гипотезы A при наступлении события B вычисляется по формуле Байеса:

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}.$$

В отличие от обычной регрессии, где предсказываются значения числовой переменной, результатом работы логистического регрессора является вероятность того, что объект принадлежит определенному классу. Следовательно, результат логистической регрессии всегда принадлежит отрезку [0, 1].

Основная идея логистической регрессии заключается в том, что пространство исходных значений может быть разделено линейной границей на две соответствующих классам области. В случае двух измерений линейная граница – это просто прямая линия, в случае трех измерений – это плоскость. Если точки исходных данных удовлетворяют этому требованию, то их можно назвать линейно разделяемыми (рис. 13).

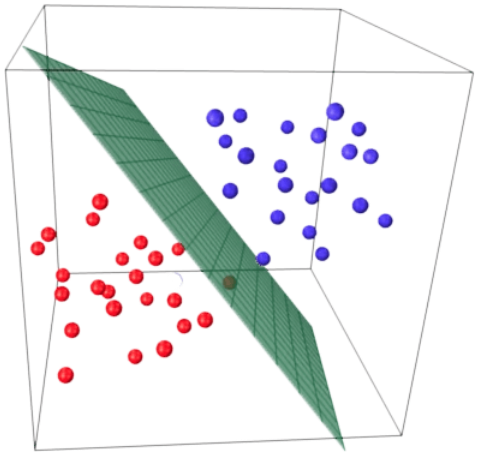


Рис. 13. Плоскость, разделяющая элементы пространства на два класса

Граница задается в зависимости от имеющихся исходных данных и обучающего алгоритма.

Метод максимального правдоподобия (MLE)

Рассмотрим широко используемый метод оценивания параметров в математической статистике – метод максимального правдоподобия. Если есть плотность (вероятность) распределения $p(y \mid \theta)$, известная с точностью до параметра θ, то для его оценки надо максимизировать функцию правдоподобия – произведение значений вероятностей в элементах нашей выборки (предполагаем, что они независимые и распределены по $p(y \mid *)$, где * – неизвестное нам истинное значение параметра θ):

$$p(y \mid \theta) = p(y_1, y_2, \ldots, y_m \mid \theta) = \prod_{i=1}^{m} p(y_i \mid \theta).$$

Точка, в которой правдоподобие достигает максимального значения, называется оценкой максимального правдоподобия (MLE, Maximum Likelihood Estimation). Заметим, что θ может быть вектором (рис. 14).

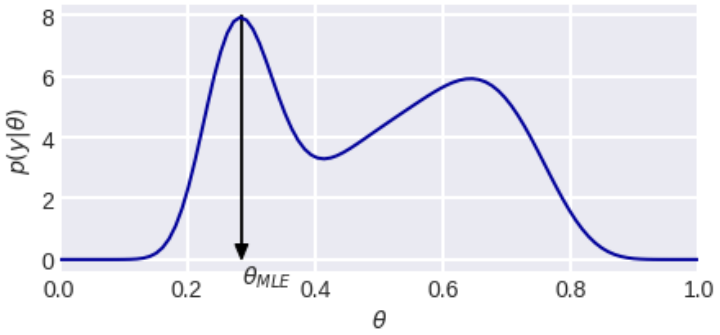


Рис. 14. Оценка максимального правдоподобия

Рассмотрим задачу бинарной классификации. Пусть банк выдает кредиты заемщикам. Данные (кроме персональных), которые необходимо предоставить банку, представлены на рис. 15.

Доход	Наличие жилья	Наличие работы	Число детей	Возврат кредита
10 000	1	1	2	1
10 000	1	1	2	-1
10 000	1	1	2	-1
10 000	1	1	2	1
10 000	1	1	2	-1
10 000	1	1	2	1
15 000	1	0	3	-1
15 000	1	0	3	-1
15 000	1	0	3	-1
15 000	1	0	3	-1
15 000	1	0	3	-1
15 000	1	0	3	1

Рис. 15. Начальная выборка данных

В данном примере для одинаковых объектов целевое значение может меняться: кредит иногда возвращают, а иногда нет. Рассмотрим линейную модель

$$a(x,\omega)=\text{sign}(\langle\omega,x\rangle).$$

Данная модель для одного и того же вектора x возвращает 1 (кредит возвращают) или –1 (кредит не возвращают). Но детерминированная модель (без случайностей) не обладает таким свойством. Мы будет иметь либо все время 1, либо все время –1. В такой ситуации будем возвращать не номер класса 1 или –1, а вероятность появления прогнозируемого класса. Тогда используем вероятностный подход для описания данной задачи:

$$P(y\,|\,x,\omega).$$

Данная условная вероятность показывает вероятность появления прогнозируемого класса модели $a(x,\ \omega)$ при предъявлении конкретного объекта x и вектора весов ω. Например, для объекта $x=[10000,1,1,2]^T$ вероятность равна

$$P(y=+1|\,x,\omega)=\frac{3}{6}=0,5.$$

А противоположную вероятность (для класса –1) можно вычислить так:

$$P(y=-1|\,x,\omega)=1-P(y=+1|\,x,\omega)=1-0,5=0,5.$$

Будем искать вектор весов ω, опираясь на выражение вероятности $P(y\,|\,x,\ \omega)$. Используем метод максимального правдоподобия:

$$\omega_* = \arg\max_{\omega} P\left(y \mid x, \omega\right),$$

т. е. выберем такое ω_*, при котором вероятность достигает максимального значения.

В данном случае мы имеем не одно наблюдение, а целую обучающую выборку, описываемую многомерной величиной:

$$P(y_1, y_2, ..., y_n \mid x_1, x_2, ..., x_n, \omega_1, \omega_2, ..., \omega_n).$$

Для задачи оптимизации нужно конкретизировать это выражение. В машинном обучении делают предположение, что все объекты $\{x_i, y_i\}$ обучающей выборки независимы между собой. Тогда многомерную формулу можно представить как произведение отдельных вероятностей:

$$P(y_1, y_2, ..., y_n \mid x_1, x_2, ..., x_n, \omega_1, \omega_2, ..., \omega_n) = \prod_{i=1}^{l} P(y_i \mid x_i, \omega).$$

Такое упрощение многомерного распределения соответствует задаче, известной под названием **наивный байесовский классификатор (Naive Bayes classifier)**.

В формулу будем подставлять конкретные значения $\{y_i\}$ и $\{x_i\}$, а менять (подбирать) можем только вектор коэффициентов ω. Следовательно, получаем функцию, зависящую только от вектора ω:

$$L(\omega) = \prod_{i=1}^{l} P(y_i \mid x_i, \omega).$$

Такая функция получила название **функции правдоподобия**. Для поиска наилучших значений вектора ω по всей обучающей выборке максимизируем ее:

$$\omega = \arg\max L\left(\omega\right).$$

Чтобы не искать максимум от произведения величин, переходим к логарифму правдоподобия (log-likelihood, log-loss):

$$\log L\left(\omega\right) = \sum \log P\left(y \mid x, \omega\right) \to \max.$$

Такое преобразование можно применить, т. к. логарифмическая функция монотонно возрастающая, следовательно, никак не повлияет на положение точки максимума функции.

Ранее при решении задач классификации мы вводили функционал, аппроксимирующий эмпирический риск некоторой выбранной функцией потерь:

$$Q(a, X^l) = \sum_{i=1}^{l} L_i(x_i, \omega) \to \min_{\omega}.$$

Из преобразований видно сходство этих двух критериев качества. Только в одном случае мы максимизируем, а в другом – минимизируем. Но это легко свести к единой задаче минимизации и записать следующее равенство:

$$Q(a, X^l) = \sum_{i=1}^{l} L_i(x_i, \omega) = -\sum_{i=1}^{l} \log P(y_i \mid x_i, \omega) \to \min_{\omega}.$$

Отсюда следует важный, ключевой вывод: вероятностный взгляд на задачи машинного обучения и взгляд через определение моделей с функциями потерь – это фактически одно и то же. Мы совершенно спокойно можем переходить из модельной плоскости в вероятностную и обратно при решении любых задач машинного обучения.

Если теперь в качестве функции потерь выбрать логарифмическую

$$L(M) = \log(1 + e^{-M}),$$

то сходство обеих сумм станет еще больше, где

$$M = \langle \omega, x_i \rangle \cdot y_i = \omega_i^T \cdot x_i \cdot y_i$$

есть отступ для *i*-го образца, показывающий, насколько далеко он находится от разделяющей гиперплоскости. Для корректного вычисления отступа для обоих классов целевые выходы должны быть $y \in \{-1; 1\}$. Также из формулы отступа автоматически вытекает линейная формула для определения разделяющей гиперплоскости:

$$g(x, \omega) = \omega \cdot x.$$

В результате мы имеем

$$Q(a, X^l) = \sum_{i=1}^{l} \log(1 + e^{-M_i}) = -\sum_{i=1}^{l} \log P(y_i \mid x_i, \omega) \to \min_{\omega}.$$

Откуда следует, что

$$\log\left(1+e^{-M}\right)=-\log P\left(y\,|\,x,\omega\right)$$

или в виде

$$P(y_{\mathrm{i}}\,|\,x_{\mathrm{i}},\omega)=\frac{1}{1+e^{-M_i}},\ i=1,2,...,l\,.$$

Таким образом, логарифмическая функция потерь приводит нас к достаточно простой формуле построения оценок вероятностей для прогнозируемого класса. Функция

$$\sigma(M)=\frac{1}{1+e^{-M_i}}$$

называется сигмоидальной или логистической. Отсюда и пошло название такого класса задач – логистическая регрессия.

На рис. 16 приведен график логистической регрессии.

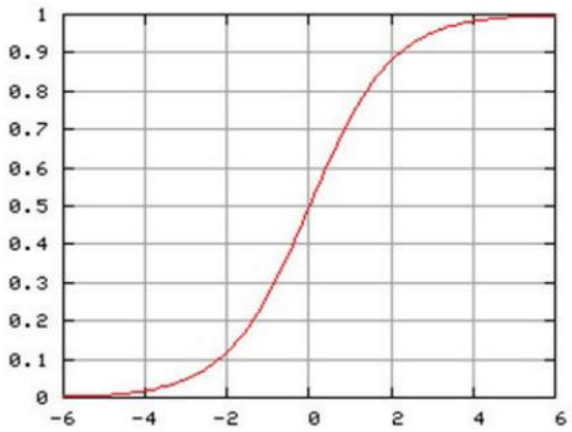


Рис. 16. График логистической регрессии

Из рис. 16 хорошо видно, что чем дальше от разделяющей гиперплоскости находится правильно спрогнозированный класс ($M > 0$), тем выше значение вероятности (уверенности) классификатора, что прогноз верен. И наоборот, если знак отступа отрицательный ($M < 0$), значит, произошла ошибка классификации, и вероятность будет меньше 0,5. Если же образ оказался точно на разделяющей гиперплоскости, то на выходе увидим значение 0,5, т. е. классификатор не уверен, к какому классу отнести текущий вектор x.

В библиотеке scikit-learn есть несколько реализаций наивного байесовского классификатора, отличающихся предположениями о распределении признаков при заданном классе.

8. Дерево принятия решений и случайный лес

Рассмотрим еще одну модель для решения задач классификации и регрессии – дерево принятия решений (решающее дерево, decision tree). Эта модель позволяет решать не только задачу бинарной классификации, но и задачу мультиклассовой классификации.

Рассмотрим пример дерева принятия решений для задачи классификации (рис. 17).

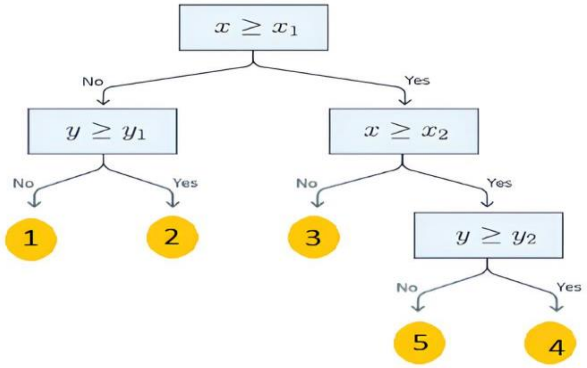


Рис. 17. Пример дерева принятия решений

Дерево принятия решений представляет собой направленный иерархический граф. В каждой вершине дерева (на рис. 17 вершины изображены прямоугольниками) проверяется условие, по которому идет разделение выборки. Если неравенство выполнено, то переходим по правому ребру, а если не выполнено – по левому. Для новой вершины снова проверяем условие и т. д. В каждом листе (на рис. 17 листья изображены кругами) содержится тот класс, к которому отнесен объект после ответа на вопросы – результат предсказания.

Глубиной дерева называется количество вершин в самом длинном пути (листья не учитываются), связывающем корень дерева и его лист. Корнем дерева называется самая первая вершина.

У дерева принятия решений есть геометрическая интерпретация. Каждому вопросу вида «значение фактора $x \geq x_i$» соответствует полупространство $x \geq x_i$. Поэтому каждому пути от корня дерева до конкретного листа соответствует некоторая область многомерного пространства, ограниченная гиперплоскостями. Для всех объектов в этой области дерево принятия решений дает одно и то же предсказание, записанное в соответствующем листе.

На рис. 18 приведена визуализация процесса построения решающих поверхностей, порождаемых деревом принятия решений.

На рис. 19 приведен пример классификации с помощью дерева принятия решений. Все объекты выборки разделены на три класса: синий, желтый и красный.

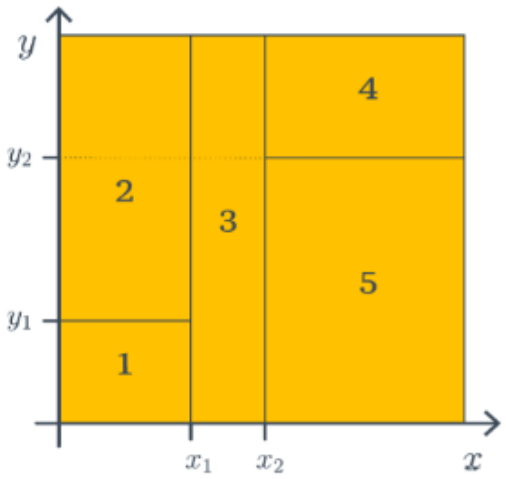


Рис. 18. Решающие поверхности

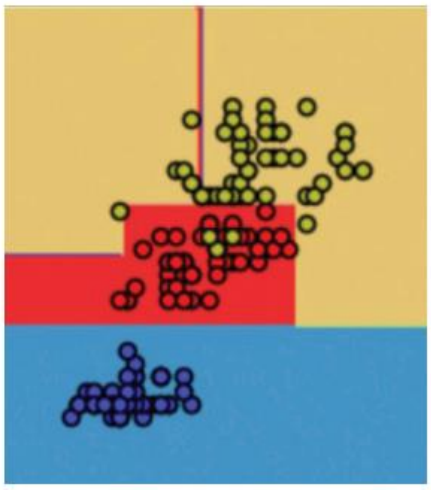


Рис. 19. Классификация деревом принятия решений

Сначала провели разделение на синий и желтый классы, после этого желтый класс разделили на красный и желтый классы. Для этого потребовалось дерево глубиной 2. Заметим, что некоторые объекты желтого класса попали в красный класс (см. рис. 19). Если продолжить разделение выборки по классам, применяя новые признаки, т. е. увеличивая количество вершин в дереве принятия решений, то можно добиться того, что модель будет идеально работать на обучающей выборке. С одной стороны, это хорошо, потому что позволяет дереву принятия решений «видеть» нелинейные зависимости. С другой стороны, это означает, что дерево принятия решений склонно к переобучению, т. е. в результате обучения параметры подбираются таким образом, что точность модели на обучающей выборке будет больше, чем на валидационной. Это может быть связано, в частности, с тем, что в процессе обучения модель подстроилась под выбросы.

Упражнение 1. Дан набор объектов, каждый из которых описывается факторами x_1, x_2, x_3 (табл. 4), и дано дерево принятия решений (рис. 20).

Табл. 4. Значения факторов

x_1	x_2	x_3
–3	2	5
10	10	1
–5	–8	3

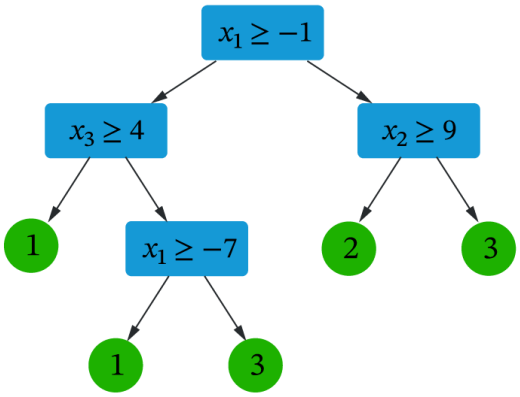


Рис. 20. Дерево принятия решений

Сопоставьте каждый из объектов с классом, который для него предсказывает дерево принятия решений (табл. 5). Может оказаться так, что какому-то классу не принадлежит ни одного объекта. В этом случае оставьте класс без сопоставления.

Табл. 5. Объекты и классы для упражнения 1

Ответ	Объект	Ответ	Класс
1	[−3; 2; 5]	А	1
2	[10; 10; 1]	Б	2
3	[−5; −8; 3]	В	3

Ответ: 1В, 2В, 3А.

Упражнение 2. Сопоставьте каждое из приведенных ниже деревьев принятия решений (рис. 21) с его представлением в виде разбиения плоскости на отдельные области (рис. 22), используя табл. 6. На графиках число внутри каждой области соответствует классу, который дерево принятия решений предсказывает для попавших в нее объектов.

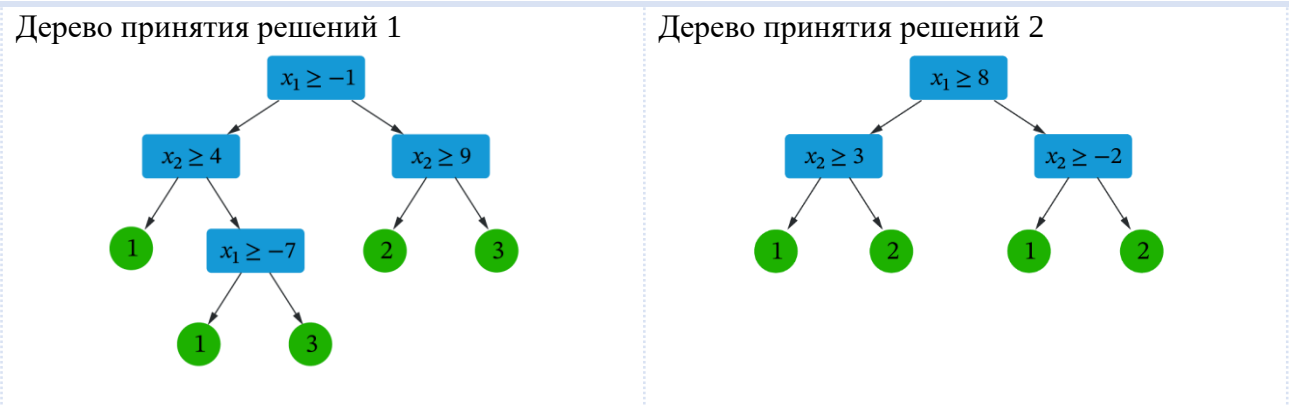


Рис. 21. Деревья принятия решений
для упражнения 2

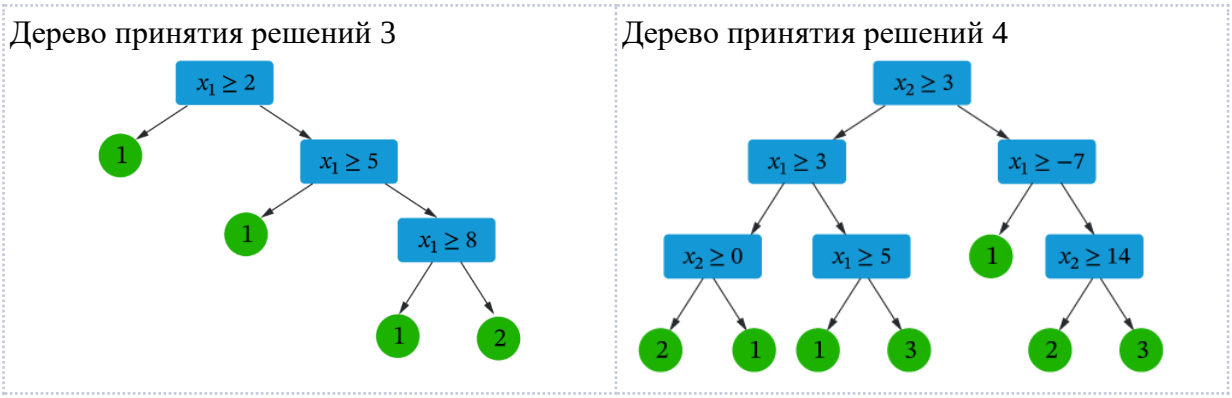


Рис. 21, лист 2

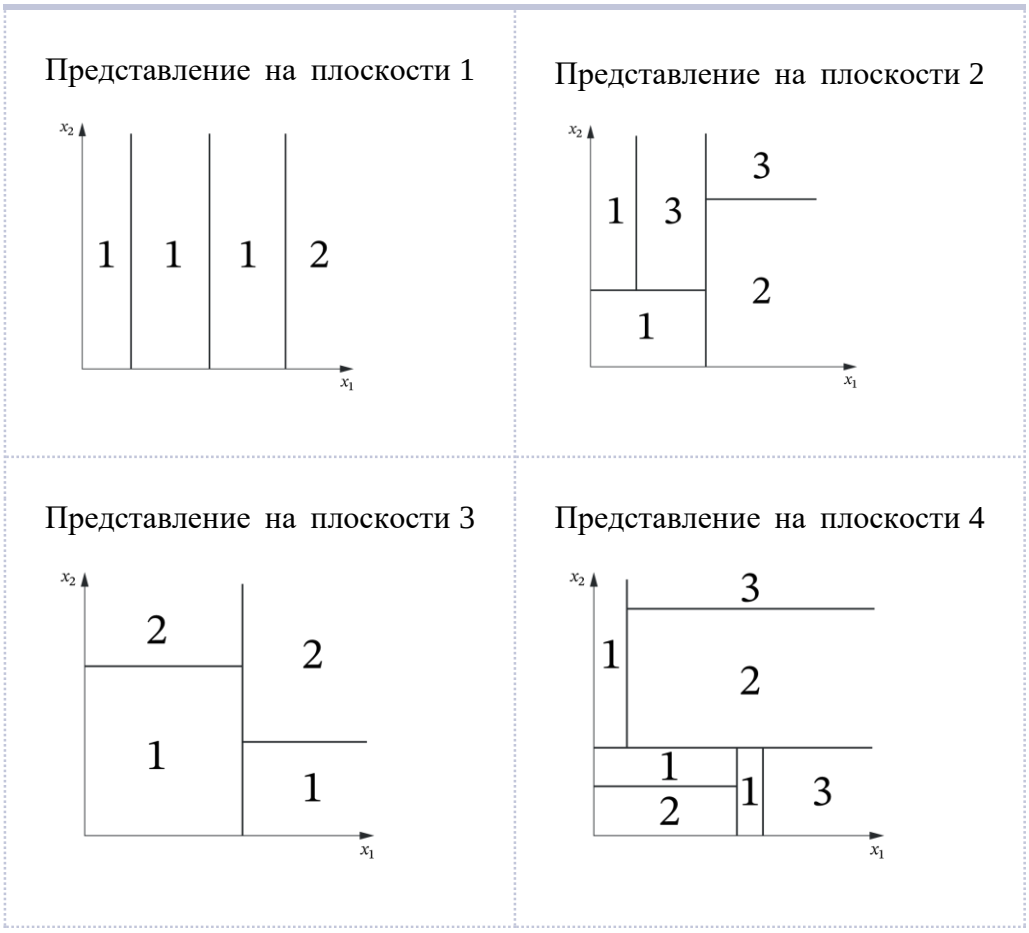


Рис. 22. Представления деревьев принятия решений на плоскости для упражнения 2

Табл. 6. Сопоставление дерева принятия решений и его представления на плоскости

Ответ	Дерево принятия решений	Ответ	Представление на плоскости
1	Дерево принятия решений 1	А	Представление на плоскости 1
2	Дерево принятия решений 2	Б	Представление на плоскости 2
3	Дерево принятия решений 3	В	Представление на плоскости 3
4	Дерево принятия решений 4	Г	Представление на плоскости 4

Ответ: 1Б, 2В, 3А, 4Г.

8.1. Алгоритм построения дерева принятия решений

Построение дерева принятия решений идет путем разделения выборки на части по вводимым факторам. Допустим, необходимо построить очередную вершину дерева. На вход вершине подается набор данных X из обучающей выборки. Выбираем порог c по критерию ошибки Q для фактора i, минимизируя критерий ошибки:

$$Q(x, i, c) \to \min .$$

В каждой вершине проверяется справедливость неравенства $x_i \geq c$. Факторы i и значения порога перебираем так, чтобы значение функции $Q(x, i, c)$ было минимальным. Набор данных X делится на два поднабора X_l и X_r – для первого поднабора неравенство в вершине не выполняется, а для второго выполняется. Далее аналогично построим левую и правую вершины, в качестве обучающей выборки для них будут использованы X_l и X_r.

В какой момент нужно остановить процесс ветвления? То есть когда вместо того, чтобы формировать новую вершину, мы будем формировать лист? Существуют разные критерии:

- все данные, которые пришли на вход, относятся к одному классу;
- ограничение по максимальной глубине дерева (максимальная глубина является гиперпараметром модели). В качестве предсказания в этом листе выберем класс, элементов которого больше всего в выборке, которая пришла на вход;
- ограничение на минимальное количество объектов в листе;
- ограничение на максимальное количество листьев в дереве.

Критерий информативности

Чтобы оценить качество работы алгоритма дерева принятий решений, используют критерий информативности, который показывает, насколько хорошо решающее дерево компонует объекты в своих листьях.

Выбор оптимального разбиения определяется критерием ошибки $Q(x, i, c)$:

$$Q(x, i, c) = \frac{|X_l|}{|X|} H(X_l) + \frac{|X_r|}{|X|} H(X_r),$$

где $|X|, |X_l|, |X_r|$ – количество объектов в выборках X, X_l, X_r соответственно; $H(X)$ – критерий информативности.

Чем меньше значение функции Q, тем более качественным получается разбиение. В случае задач классификации используется критерий информативности Джини. Вычисляется критерий Джини по формуле

$$H(X) = \sum_{c \in \{1, 2. \ldots, k\}} p_c (1 - p_c),$$

где p_c – это доля объектов, которые принадлежат классу c в выборке X:

$$p_c = \frac{1}{X} \sum_{i \in X} [y_i = c].$$

Можно также использовать энтропийный критерий информативности (энтропию):

$$H(X) = -\sum_{c=1}^{k} p_c \ln p_c .$$

Энтропия в дереве принятия решений означает однородность. Если данные полностью однородны, то энтропия равна нулю. Если данные разделены 1 : 1, то энтропия равна 1. Энтропия измеряется в диапазоне от 0 до 1.

Рассмотрим пример. Пусть выборка содержит 100 объектов, принадлежащих классу 0 или классу 1. Если 30 объектов принадлежат классу 0, тогда $p_0 = \frac{3}{10}$. Оставшиеся 70 объектов принадлежат классу 1, тогда $p_1 = \frac{7}{10}$. Энтропия приблизительно равна 0,88, т. е. имеем высокий уровень энтропии, или беспорядка.

Величина, обратная энтропии, – прирост информации. Чем выше прирост информации, тем меньше энтропия, а следовательно, меньше неучтенных данных и лучше результат.

Стрижка деревьев

При стрижке дерева строится переобученное дерево максимальной глубины, например когда в каждом листе содержится по одному объекту. Затем оптимизируется его структура, т. е. убираются лишние листья на основе различных критериев, с целью улучшения обобщающей способности.

8.2. Случайный лес

Объединение нескольких деревьев, обученных на имеющихся данных, дает возможность получить на тестовых данных качество выше, чем могла показать каждая из этих моделей отдельно. Одной из разновидностей композиции решающих деревьев является случайный лес.

В качестве результата работы случайного леса, состоящего из нескольких деревьев принятия решений, берут усредненное значение, которое предсказывают отдельные деревья. Тогда некорректная работа деревьев на отдельных вы-

бросах скомпенсируется, а на данных, отражающих общую зависимость, случайный лес будет работать корректно, поскольку отдельные деревья также в среднем работали корректно на этих данных. Деревья нужно сделать максимально непохожими друг на друга. Этого можно добиться одним из следующих способов:

- каждое дерево обучается на своей выборке;
- в каждой вершине каждого дерева добавляется случайность при выборе фактора и порогового значения.

Большое количество разных обучающих выборок можно получить с помощью подхода *бутстрап*. Его идея в том, чтобы на основе исходной обучающей выборки размером n сгенерировать другую обучающую выборку размером n, которая называется бутстрап-выборкой. Пронумеруем объекты в обучающей выборке от 1 до n и будем генерировать новые выборки по следующему правилу: на каждую из n позиций в бутстрап-выборке будем помещать случайный объект из тестовой выборки. За счет того, что объекты выбираются случайно и независимо, в бутстрап-выборку может попасть несколько одинаковых объектов из исходной выборки. Для построения случайного леса принято генерировать много бустрап-выборок, например порядка 10 000.

Чтобы добавить случайность в построение каждого дерева, нужно в каждой вершине дерева перебирать не все факторы для построения оптимального вопроса, а выбирать случайный набор факторов и перебирать только их значения.

Алгоритм построения случайного леса:

1. Генерируем N бутстрап-выборок на основе обучающей выборки.

2. На каждой бутстрап-выборке строим свое дерево принятия решений. Для каждого дерева выполнены два условия:

- минимальное число элементов в листе равно L;
- в каждой вершине используется K случайных факторов.

Чтобы сделать предсказание с помощью случайного леса для задачи регрессии, нужно усреднить предсказания всех деревьев для конкретного объекта. Для задачи классификации нужно отнести объект к тому классу, за который деревья «отдали» больше всего голосов.

Гиперпараметры случайного леса – числа N, L, K:

- N – количество деревьев, которые используются для предсказания. При увеличении количества используемых деревьев модель не переобучается. Но важно понимать, что чем больше деревьев в случайном лесе, тем сложнее обучать модель;
- L – минимальное количество таких объектов обучающей выборки, которые могут попасть в лист дерева. Этот параметр принято брать небольшим, например 3. Нужно, чтобы каждое дерево хорошо решало поставленную задачу на своей обучающей выборке;
- K – размер случайного подмножества всего множества факторов. Его принято выбирать исходя из задачи. Для задачи регрессии можно поло-

жить $K = m/3$, где m – число всех факторов. Для задачи классификации можно положить $K = \sqrt{m}$. Это только рекомендации, и если есть возможность экспериментировать с гиперпараметрами, то стоит это сделать.

8.3. Bagging

Рассмотрим методы обучения композиции алгоритмов. В методе бэггинг (bagging) алгоритмы обучаются на сгенерированных бутстрапом выборках: каждый алгоритм на своей выборке. Каждое решающее дерево строится до конца, т. е. пока в листе не окажется единственный объект или минимальное количество объектов. Признак, по которому идет разделение в выборки на каждом шаге построения дерева, выбирается случайно. Далее усредняются прогнозы каждого дерева.

8.4. Boosting

В методе бустинг (boosting) деревья обучаются последовательно, причем каждый последующий алгоритм учитывает ошибки предыдущего. Алгоритм бустинга выполняет следующие шаги для обучения:

1. Создает первое дерево принятия решений (базовый алгоритм).
2. Обучает его на всей выборке.
3. Делает предсказание.
4. Вычисляет ошибку.
5. Создает следующее дерево принятия решений и обучает его на ошибках, полученных на предыдущем шаге, и на том же наборе параметров.
6. Суммирует предсказания полученных моделей (деревьев).
7. Повторяет шаги 4–6, пока сумма предсказаний не перестанет меняться или ошибка не станет меньше определенного порога.

Алгоритм бустинга учитывает вклад каждой модели в зависимости от ее точности, т. е. у каждой модели композиции есть свой вес. Модель с лучшими прогнозами будет иметь влияние на окончательное решение.

9. Метод опорных векторов (SVM)

9.1. Метод опорных векторов для линейно разделимого случая

Рассмотрим задачу бинарной классификации образов с учетом концепции разделяющей гиперплоскости. Будем считать, что мы оперируем двумерным признаковым пространством. Тогда каждый образ класса может быть представлен точкой на плоскости. Предположим, что образы обучающей выборки распределены так, как показано на рис. 23.

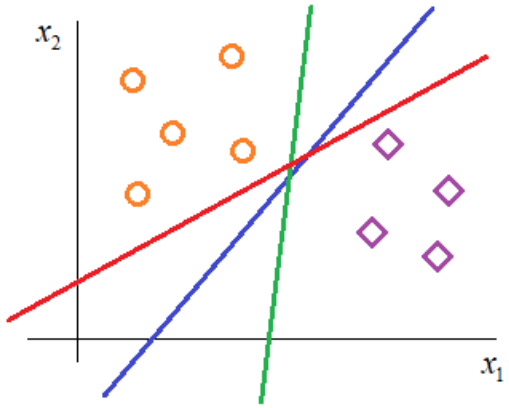


Рис. 23. Различные разделяющие линии для одинаковой выборки

Для данного случая можно построить несколько разнообразных линий разделения (гиперплоскостей), каждая из которых способна правильно отделять один класс от другого. Возникает вопрос о предпочтительности какого-либо из вариантов разделения. В рамках машинного обучения мы должны учитывать, что модель, обученная на конкретной выборке, должна успешно обрабатывать другие произвольные наборы данных из того же распределения. Следовательно, модель должна обладать хорошими обобщающими способностями и не должна быть слишком специфичной для конкретного набора данных. Рассматривая предложенные линии разделения с этой точки зрения, увидим, что более предпочтительной кажется синяя линия. Это объясняется тем, что красная и зеленая линии предполагают дополнительные ограничения относительно распределения образов обоих классов. Красная линия предполагает горизонтальное распределение образов, а зеленая – вертикальное (рис. 24).

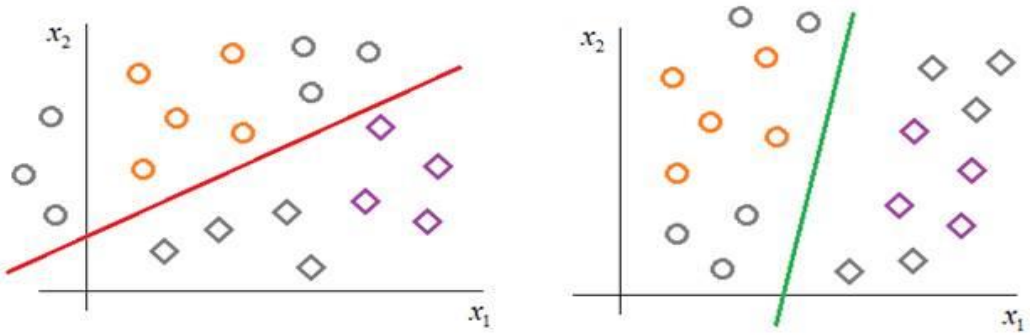


Рис. 24. Предположения линий вне обучающей выборки

В свою очередь, синяя линия, сохраняя исходное распределение в большей степени, также формулирует свои предположения. Однако она ориентирована на визуальное сохранение исходного распределения. Эта концепция разделения гиперплоскостей, которая ориентирована исключительно на обучающую выборку и, по возможности, не делает дополнительных предположений о распределении образов в классах, легла в основу метода опорных векторов (Support Vector Machine – SVM).

Формализуем это интуитивное соображение на математическом уровне. Ответим на вопрос о том, какая же разделяющая гиперплоскость делает минимальные предположения о распределении классов и, следовательно, обеспечивает лучшую обобщающую способность алгоритма классификации. С точки зрения SVM оптимальная разделяющая гиперплоскость – это та, которая образует наиболее широкую полосу между объектами двух классов. При этом сама гиперплоскость точно проходит посередине этой полосы (рис. 25).

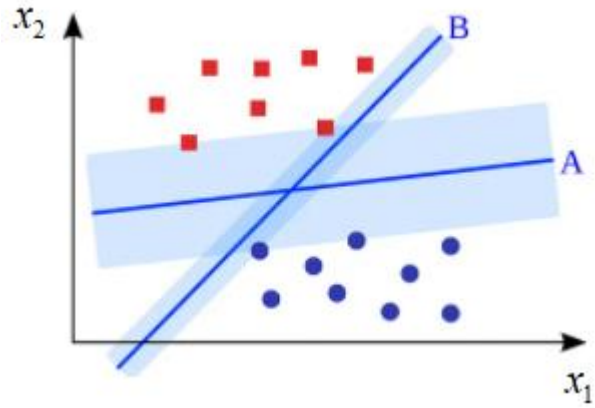


Рис. 25. Различные разделяющие гиперплоскости

Ширина полосы является критическим фактором, поскольку чем шире полоса, тем более надежно классификатор сможет разделять образы разных классов. Для того чтобы сформулировать эту идею в математических терминах, необходимо сначала определить модель классификатора, которая фактически определяет уравнение гиперплоскости в пространстве признаков. Выберем наиболее простую линейную модель:

$$a(x) = \text{sign}\,(\omega^T x - b) = \text{sign}\,(\langle \omega, x \rangle - b) = \text{sign}\,(\omega \cdot x - b).$$

Это уравнение может быть представлено различными способами, но все они описывают линейную комбинацию вектора параметров ω с образом x, дополненную смещением b. В результате работы модель возвращает значения

$$a(x) \in \{-1; +1\}.$$

Сначала предположим, что обучающая выборка состоит из линейно разделимых образов, а затем расширим этот случай на линейно неразделимый.

Ширина полосы будет зависеть от расположения граничных векторов x в пространстве признаков (рис. 26).

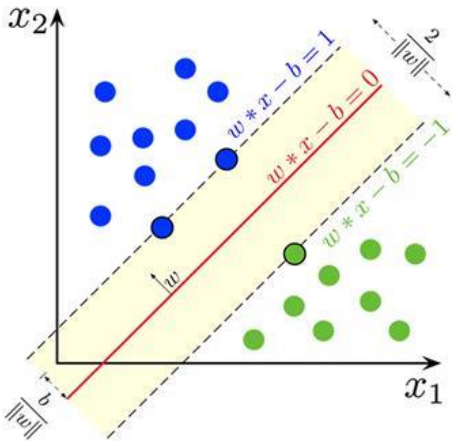


Рис. 26. Общее описание разделяющей гиперплоскости

Пусть любые два образа, принадлежащие разным классам, находятся близко к разделяющей границе, т. е. лежат на границе полосы (рис. 27). Тогда ширину полосы можно вычислить как проекцию вектора x на вектор ω:

$$\langle \omega, x_+ - x_- \rangle = \omega^T \cdot (x_+ - x_-) = |\omega| \cdot |x_+ - x_-| \cdot \cos\alpha .$$

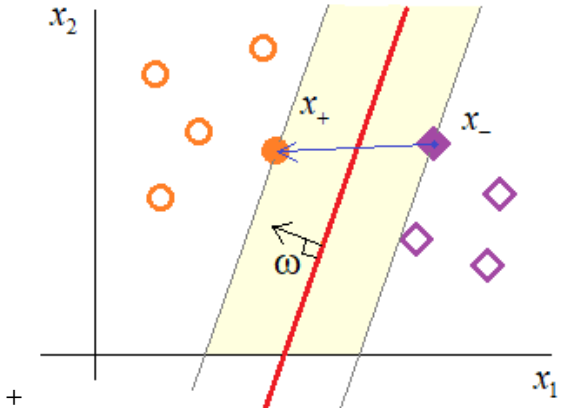


Рис. 27. Граничные элементы выборки

Итак, ширина полосы, умноженная на длину вектора коэффициентов ω, дает нам конечный результат:

$$L = |x_+ - x_-| \cdot \cos\alpha = \frac{\langle \omega, x_+ - x_- \rangle}{|\omega|}.$$

Эту величину нужно максимизировать:

$$L = \frac{\langle \omega, x_+ - x_- \rangle}{|\omega|} \to \max .$$

Для удобства в методе опорных векторов в знаменателе запишем не длину вектора ω, а квадрат его нормы:

$$\|\omega\|^2 = \omega^T \cdot \omega.$$

Таким образом, получим выражение для максимизации ширины полосы:

$$L = \frac{\langle \omega, x_+ - x_- \rangle}{\|\omega\|} \to \max.$$

Такая замена не влияет на суть задачи, но упрощает вычисления, т. к. не нужно проводить операцию извлечения квадратного корня при решении оптимизационной задачи.

В задачах бинарной классификации вводится понятие отступа (margin):

$$M_i = y_i \cdot a(x_i) = y_i\big(\langle \omega, x_i \rangle - b\big), \quad i = 1, 2, ..., l.$$

Эта величина характеризует расстояние от разделяющей гиперплоскости до выбранного образа. Причем

$$\begin{cases} M_i > 0 \text{ при верной классификации;} \\ M_i < 0 \text{ при неверной классификации.} \end{cases}$$

Так как рассматривается случай линейно разделимых образов, то заведомо существуют значения ω и b, такие, что

$$M_i = y_i\big(\langle \omega, x_i \rangle - b\big) > 0, \quad i = 1, 2, ..., l.$$

Можно нормировать значения отступа так, чтобы соблюдалась следующая логика:

$$M_i(x_+) = 1;\ M_i(x_-) = 1.$$

Тогда ширина полосы будет определяться выражением

$$L = \frac{2}{\|\omega\|^2} \to \max.$$

Следовательно, для линейно разделимых образов мы получаем следующую задачу минимизации:

$$\begin{cases} \frac{1}{2}\|\omega\|^2 \to \min\limits_{\omega,\, b,\, \xi}; \\ M_i(\omega, b) \geq 1, \quad i = 1, 2, ..., l. \end{cases}$$

Итак, задача заключается в том, чтобы найти значения ω и b, минимизирующие квадратичную норму весов. В то же время эти значения должны быть такими, чтобы все отступы были больше единицы, за исключением тех образов, которые лежат непосредственно на границах полосы, где отступ должен быть равен единице.

9.2. Метод опорных векторов для линейно неразделимого случая

В общем случае образы в обучающих выборках редко бывают линейно разделимыми. Поэтому при линейно неразделимой выборке мы не можем найти параметры ω и b, которые удовлетворяли бы линейным ограничениям на отступы:

$$M_i(\omega, b) \geq 1, \quad i = 1, 2, ..., l.$$

Для решения этой проблемы был продложен подход, разрешающий классификатору допускать ошибки на некоторую величину (slack variables):

$$\xi_i \geq 0, \quad i = 1, 2, ..., l$$

для каждого i-го образа:

$$M_i(\omega, b) \geq 1 - \xi_i, \quad i = 1, 2, ..., l\,.$$

Величины slack variables можно рассматривать как некоторый штраф за нарушение исходного неравенства. Очевидно, если все slack variables стремятся к бесконечности, то можно взять любые веса, в том числе 0, и оптимизационная задача будет решена. Однако это не соответствует требуемым целям. Разрешается ошибаться, но величина этой ошибки должна быть как можно меньше. Таким образом, нужно найти такие значения ω и b, чтобы

$$\xi_i \to 0, \quad i = 1, 2, ..., l.$$

Это условие нужно учесть в алгоритме минимизации, записав его следующим образом:

$$\begin{cases} \frac{1}{2}\|\omega\|^2 + C \cdot \sum_{i=1}^{l} \xi_i \to \min\limits_{\omega, b, \xi}; \\ M_i(\omega, b) \geq 1 - \xi_i; \\ \xi_i \geq 0, \quad i = 1, 2, ..., l, \end{cases}$$

где C – гиперпараметр, определяющий степень минимизации величин ξ_i.

В результате математических преобразований исходная система становится эквивалентной безусловной задаче минимизации:

$$\sum_{i=1}^{l} (1 - M_i(\omega, b)) + \frac{1}{2C}\|\omega\|^2 \to \min\limits_{\omega, b}.$$

Функция потерь здесь имеет следующий вид (зеленый график) и называется *hinge loss* (рис. 28).

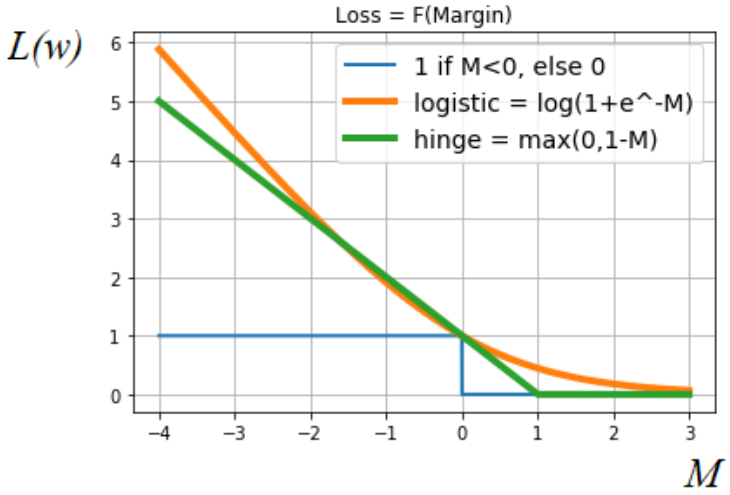


Рис. 28. Сравнение функций потерь

Для сравнения здесь приведена логарифмическая функция потерь. Суть заключается в том, что SVM фактически представляет собой решение оптимизационной задачи при использовании функции потерь *hinge*. Эта функция начинает «наказывать», если минимальный отступ для объекта становится меньше единицы, и не «наказывает» (нулевой штраф), если отступ больше или равен 1. Следовательно, для функции потерь *hinge* ширина полосы между образами двух классов имеет важное значение. В этом отличие от логистической функции потерь, которая стремится максимально раздвинуть образы классов относительно разделяющей гиперплоскости.

Также из метода опорных векторов очень хорошо виден геометрический смысл L2-регуляризатора. Фактически этот регуляризатор отвечает за максимизацию ширины полосы между соседними образами двух разных классов, что в итоге приводит к улучшению обобщающих способностей полученного классификатора.

9.3. Реализация метода опорных векторов (SVM)

Для реализации метода опорных векторов используем *условие Каруша – Куна – Таккера с поиском седловой точки функции Лагранжа*. Коэффициенты ω могут быть вычислены по формуле

$$\omega = \sum_{i=1}^{l} \lambda_i \cdot y_i \cdot x_i ,$$

где $\{\lambda_i\}$ $(i = 1, \ldots, l)$ – некоторые коэффициенты, которые вычисляются по ходу решения данной оптимизационной задачи.

Оптимальный вектор ω представляется в виде линейной комбинации наблюдений из обучающей выборки. Обычно нулевых значений λ достаточно много, оставшиеся используются при расчете коэффициентов. Такие наблюдения (векторы) получили название *опорных*. Отсюда и пошло название *метод опорных векторов*.

Значения коэффициентов λ можно интерпретировать следующим образом:

$$\lambda_i = 0;\ \xi_i = 0;\ M_i \geq 1;$$
$$0 < \lambda_i < C;\ \xi_i = 0;\ M_i = 1;$$
$$\lambda_i = C;\ \xi_i > 0;\ M_i < 1.$$

Первые называются периферийными объектами, следующие – опорными граничными объектами, последние – опорными ошибочными объектами (рис. 29).

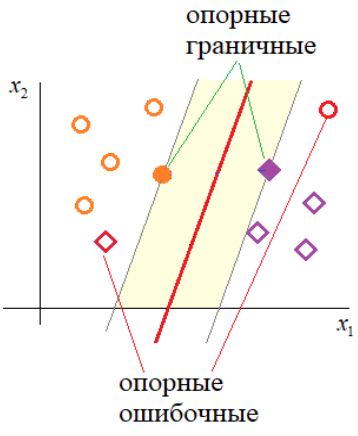


Рис. 29. Разграничение опорных векторов по степени значимости

9.4. Реализация SVM на Python

Представленный ниже код реализует использование метода опорных векторов (SVM) для классификации двух классов данных в двумерном пространстве признаков с учетом, что данные линейно разделимы. Программа начинается с импорта необходимых библиотек: **numpy** для работы с массивами данных, **matplotlib.pyplot** для визуализации данных и построения графиков, **sklearn.svm** для использования метода опорных векторов.

Затем программа создает обучающие данные, задавая массив **X**, содержащий координаты точек в двумерном пространстве, и массив **y**, содержащий метки классов для каждой точки. После этого выполняется обучение модели SVM с помощью классификатора **svm.SVC** с линейным ядром, который обучается на обучающих данных с помощью метода **fit**.

Полученные параметры разделяющей гиперплоскости – коэффициенты гиперплоскости и смещение – извлекаются из атрибутов **coef_** и **intercept_** модели. Затем данные и разделяющая гиперплоскость визуализируются на графике: данные отображаются с использованием функции **scatter**, разделяющая гиперплоскость строится с помощью функции **plot**, а опорные векторы выделяются на графике кружками. График отображается с помощью функции **show**.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

X = np.array([[1, 2], [2, 3], [3, 3], [2, 1], [3, 2],
          [8, 9], [9, 10], [10, 10], [9, 8], [10, 9]])
y = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])

clf = svm.SVC(kernel='linear')
clf.fit(X, y)
w = clf.coef_[0]
b = clf.intercept_[0]

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)

x_axis = np.linspace(0, 12)
y_axis = -w[0] / w[1] * x_axis - b / w[1]
plt.plot(x_axis, y_axis, 'k-')

plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
        s=100, facecolors='none', edgecolors='k')
plt.show()
```

Результаты выполнения программы представлены на рис. 30.

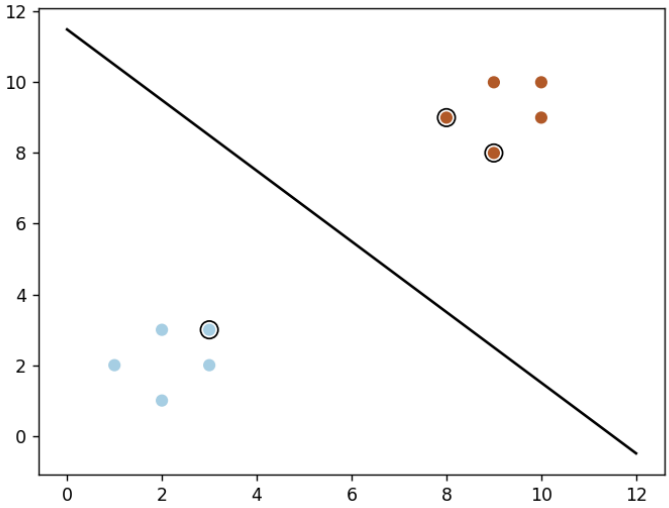


Рис. 30. Графическое изображение линейно разделимой выборки

Следующий пример предполагает, что данные линейно неразделимы. Код приведен ниже. Сначала создается случайная выборка данных, состоящая из 100 точек, где первые 50 точек принадлежат к классу 0, а остальные 50 – к классу 1. Данные распределены вдоль двух гауссиан с некоторым смещением.

Модель SVM с линейным ядром обучается на этих данных. Разделяющая гиперплоскость визуализируется на графике, а также отображаются опорные векторы, которые являются точками данных, ближайшими к разделяющей гиперплоскости.

На графике точки обозначены с использованием цветовой схемы plt.cm.Paired: точки класса 0 отображаются синим цветом, а точки класса 1 – красным (рис. 31). Разделяющая гиперплоскость изображается линией, а опорные векторы – черными ободками.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

np.random.seed(0)
X = np.random.randn(100, 2) * 2
X[:50] += 3
y = np.concatenate([np.zeros(50), np.ones(50)])

clf = svm.SVC(kernel='linear')
clf.fit(X, y)

w = clf.coef_[0]
b = clf.intercept_[0]
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)

x_axis = np.linspace(-4, 8)
y_axis = -w[0] / w[1] * x_axis - b / w[1]
plt.plot(x_axis, y_axis, 'k-')

plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
        s=100, facecolors='none', edgecolors='k')

plt.show()
```

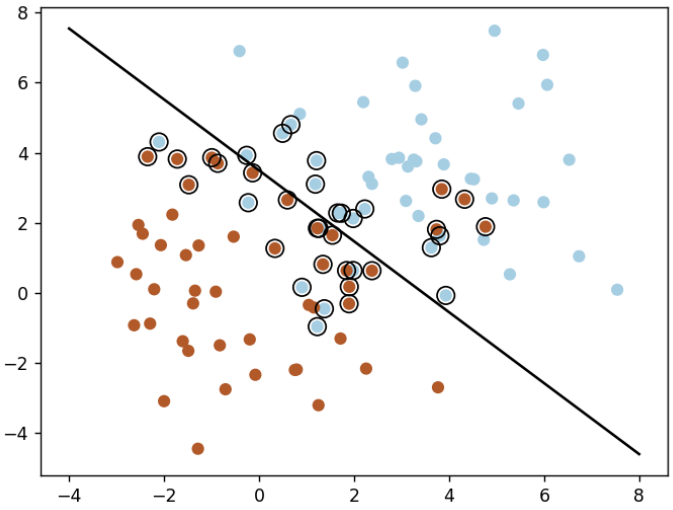


Рис. 31. Графическое изображение линейно неразделимой выборки

9.5. SVM с нелинейными ядрами

Вернемся к общей формуле модели линейного классификатора:

$$a(x) = \text{sign}\left(\sum_{i=1}^{h} \lambda_i y_i \langle x_i, x\rangle - b\right).$$

Линейные преобразования в исходном признаковом пространстве можно заменять и другими. Например, возвести скалярное произведение в квадрат:

$$\langle x_i, x\rangle^2 = \left(x_i^T \cdot x\right)^2, \quad i = 1, 2, ..., h.$$

Для указанного преобразования и некоторого ряда других преобразований решение системы остается неизменным.

На рис. 32 изображены разделяющие гиперплоскости для разных типов ядер.

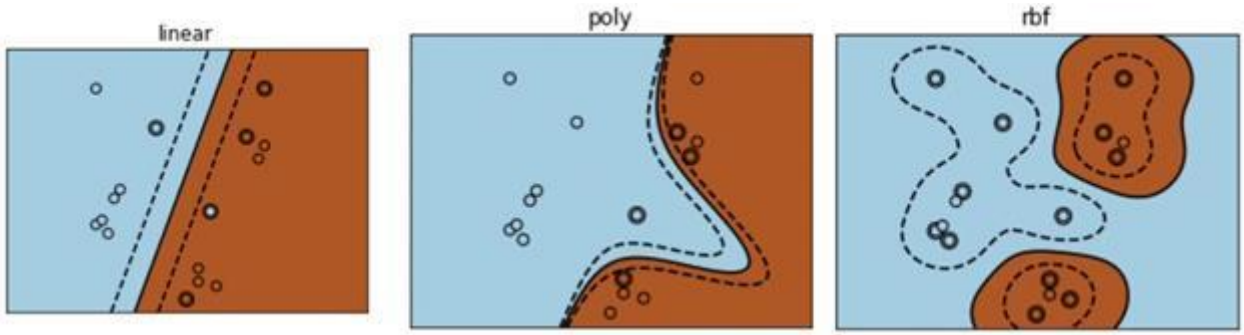


Рис. 32. Разделяющие гиперплоскости для разных типов ядер

Здесь linear – обычное скалярное произведение, poly – полиномиальное ядро, rbf – радиальные ядра, а также ядро вида th. Перечисленные ядра можно представить следующим образом:

$$K(x, x') = \langle x, x' \rangle^d;$$
$$K(x, x') = (\langle x, x' \rangle + 1)^d;$$
$$K(x, x') = \exp(-\gamma \|x - x'\|^2);$$
$$K(x, x') = \text{th}(k_1 \langle x, x' \rangle - k_0), \quad k_0, k_1 \geq 0.$$

На основе ядра вида th получается аналог двухслойной нейронной сети с сигмоидальными функциями активации.

9.6. Способы синтеза ядер

Существует несколько простых правил синтеза ядер для метода опорных векторов. К основным можно отнести следующие подходы:

- $K(x, x') = \langle x, x' \rangle$ – скалярное произведение;
- $K(x, x') = 1$ – константа;
- $K(x, x') = K_1(x, x') \cdot K_2(x, x')$ – произведение ядер (подходящих для SVM);
- $K(x, x') = \langle \psi(x), \psi(x') \rangle, \quad \nabla \psi : X \to R$ – применение функции;
- $K(x, x') = \alpha_1 K_1(x, x') + \alpha_2 K_2(x, x'), \quad \alpha_1, \alpha_2 > 0$ – сумма ядер.

На практике обычно используется ограниченный набор ядер для метода опорных векторов (SVM): линейные, полиномиальные, радиальные и гиперболический тангенс. Эти ядра встроены в библиотеку scikit-learn и обычно их достаточно для большинства задач. Важно отметить, что для каждого типа ядра вычислительная схема SVM немного изменяется. Именно поэтому нельзя создать универсальный функционал для произвольных ядер без использования субградиентных методов.

9.7. Преимущества и недостатки SVM

Преимущества SVM:

- решает задачу квадратичного программирования, которая имеет единственное оптимальное решение;
- позволяет выявить опорные векторы, включая выбросы, присутствующие в обучающей выборке. Иногда SVM применяется для обнаружения выбросов;
- в отличие от нейронных сетей SVM позволяет определить необходимое число опорных векторов, аналогичное числу нейронов скрытого слоя.

Недостатки SVM:

- нет общих методов оптимизации для произвольных ядер;
- нет встроенного механизма выбора наиболее информативных признаков, аналогичного L1-регуляризации;
- необходимо настраивать параметр *C* для каждой задачи.

Тем не менее SVM считается одним из лучших методов классификации среди линейных классификаторов. Благодаря максимизации ширины полосы между классами этот метод обычно демонстрирует лучшие обобщающие способности на реальных данных. SVM также относительно легко адаптировать для работы с нелинейными данными, используя различные ядра, его возможности могут быть расширены с использованием L1-регуляризации.

10. Нейронные сети

10.1. Принципы действия нейронной сети

Нейронная сеть – это математическая модель, ее программное или аппаратное воплощение, построенная по принципу организации и функционирования биологических нейронных сетей. Здесь реализованы идеи моделирования мыслительных или поведенческих явлений в сетях из связанных между собой простых элементов. Существует несколько видов нейронных сетей:

- искусственные нейронные сети (Artificial Neural Networks, ANN). Используются для решения широкого спектра задач, в том числе задач распознавания образов и классификации. ANN применяются, например, для анализа медицинских изображений (рентгеновских снимков, сканов компьютерной томографии и т. д.);
- сверточные нейронные сети (Convolutional Neural Networks, CNN). Они оптимизированы для обработки изображений и видео. CNN используют специальные фильтры, чтобы выяснить, что изображено на картинке и какие детали существенны. Применяются, например, для распознавания лиц при разблокировке смартфонов;
- рекуррентные нейронные сети (Recurrent Neural Networks, RNN). Используются для работы с последовательными данными: текстами, временными

рядами, аудио. RNN можно применять, например, для автоматического анализа тональности текстов в соцмедиа.

В настоящее время существенно расширяется перечень задач, решаемых с помощью нейронных сетей, и этот перечень неисчерпаем.

При построении нейронных сетей были учтены важные особенности их естественных аналогов. А именно: ошибка в срабатывании отдельного нейрона остается незаметной в общей массе взаимодействующих клеток. Следовательно, нейронная сеть является устойчивой системой, в которой отдельные сбои не оказывают существенного влияния на результаты ее функционирования. Также важной особенностью является высокая скорость функционирования нейронной сети. Достигается она благодаря параллельной обработке информации огромным количеством нейронов, соединенных многочисленными связями.

Искусственные нейронные сети обучаются на основе опыта, обобщают прецеденты для новых случаев и извлекают существенные свойства из сведений, содержащих излишние данные, т. е. они обладают свойствами обучения, обобщения и абстрагирования. Нейронная сеть способна менять свое поведение в зависимости от внешней среды. После получения входных сигналов вместе с требуемыми выходами они самонастраиваются с целью обеспечить ожидаемую реакцию, т. е. обучаются. Результат, полученный после обучения, может быть в некоторой степени нечувствителен к небольшим изменениям входных сигналов. Сеть делает обобщение автоматически благодаря своей стуктуре. И если на вход нейронной сети поступает несколько искаженных вариантов образов, то сеть сама может создать на выходе новый для нее идеальный образ.

Нейронная сеть обучается следующим образом: ей предъявляются объекты с указанием их принадлежности определенному образу или классу, и сеть должна приобрести способность реагировать одинаково на все объекты одного образа или класса и по-разному реагировать на объекты различных образов или классов. После обучения следует процесс распознавания новых объектов, характеризующий действия уже обученной системы. По мере накопления опыта нейронная сеть может повышать точность результатов и адаптироваться к происходящим изменениям.

10.2. Нейронная сеть в виде перцептрона

Перцептрон – это нейронная сеть прямого распространения сигнала (без обратных связей), в которой входной сигнал преобразуется в выходной, проходя последовательно через один или несколько слоев.

Слой – это один или несколько нейронов, на входы которых подается один и тот же общий сигнал. В рамках одного слоя данные обрабатываются параллельно, а в масштабах всей сети – последовательно, от слоя к слою. Нейроны текущего слоя передают результаты своей работы на входы нейронам следующего слоя. В конце стоит один искусственный нейрон, который агрегирует поступающую на него информацию и возвращает одно итоговое значение.

На рис. 33 приведена нейронная сеть в виде многослойного перцептрона, состоящего из трех слоев. В первом слое этого перцептрона находится один искусственный нейрон, во втором слое – два нейрона, в третьем – один нейрон. Первый слой называется входным или сенсорным, внутренние слои называются скрытыми или ассоциативными, последний слой называется выходным или результативным.

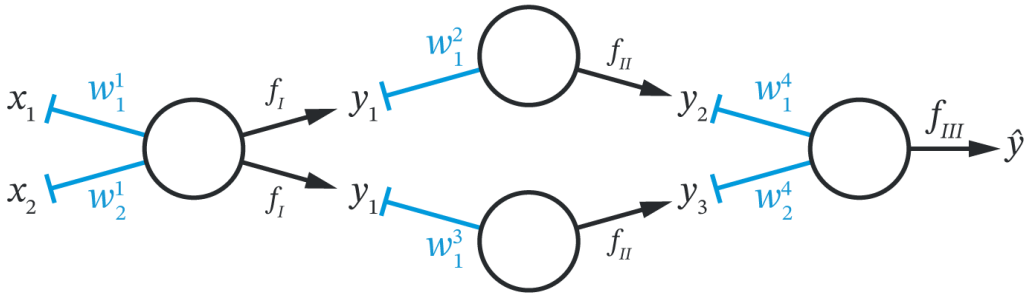


Рис. 33. Нейронная сеть в виде многослойного перцептрона

В основе работы искусственной нейронной сети лежит искусственный нейрон. В рамках сети нейроны связаны между собой по принципу, аналогичному связям биологической нейронной сети.

Перцептрон – это модель машинного обучения, которая является прародителем современных методов глубокого обучения. Модель перцептрона была вдохновлена моделью из реальной жизни, в частности строением реального физического нейрона (рис. 34).

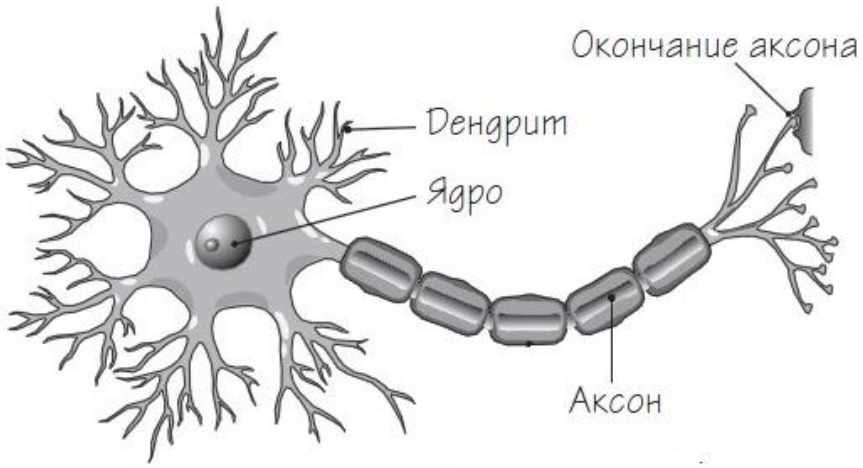


Рис. 34. Физический нейрон

В 1943 г. американские кибернетики и нейробиологи Уоррен Мак-Каллок и Уолтер Питтс предложили математическую модель, которая повторяла строение физического нейрона. Эта модель получила название «искусственный нейрон» (рис. 35).

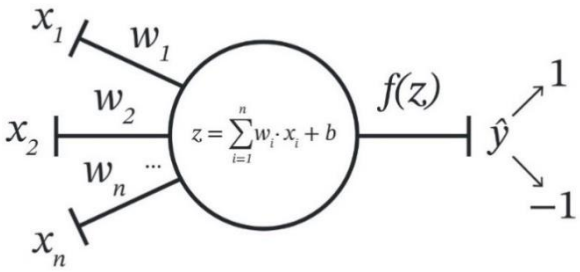


Рис. 35. Искусственный нейрон

Искусственный нейрон принимает на вход сигнал по нескольким каналам (по аналогии с физическим нейроном они называются дендритами) от произвольного числа других нейронов. Связи, по которым выходные сигналы одних нейронов поступают на вход других, называют синапсами, как и в естественной нейронной сети. На вход дендрит с номером i получает значение фактора x_i. Каждому дендриту соответствует вес ω_i.

Процесс обработки искусственным нейроном входящих сигналов выглядит так: нужно взять каждый входящий сигнал x_i , домножить на соответствующий вес ω_i и сложить:

$$\hat{y} = f(z) - f(\sum_{i=1}^{n} \omega_i x_i + b).$$

В этой формуле к взвешенной сумме входящих в нейрон сигналов добавлена константа b, которая нужна для математической корректности этого выражения. На основе полученной суммы нейрон должен принять решение, передавать свой сигнал дальше или нет. Если нейрон передает сигнал дальше, говорят, что нейрон прошел активацию.

Взвешенная сумма z передается заданной для этого нейрона функции f, которая называется функцией активации. В качестве результата работы нейрона возвращается значение

$$\hat{y} = f(z) - f(\sum_{i=1}^{n} \omega_i x_i + b).$$

Если $\hat{y} = 1$, будем говорить, что нейрон прошел активацию, если $\hat{y} = -1$ – что не прошел. Чтобы нейрон мог работать по этому правилу, достаточно взять функцию

$$\hat{y} = f(z) - f(\sum_{i=1}^{n} \omega_i x_i + b).$$

Перцептрон – *обучающаяся* модель, построенная на основе искусственного нейрона. Обучать перцептрон можно за счет изменения его параметров, например его весов.

Рассмотрим принцип, согласно которому можно обучить перцептрон решать задачу классификации (табл. 7). Данные представляют собой набор объектов (например, объекты – точки на плоскости с двумя координатами), каждый из которых классифицирован – отнесен либо к классу 1 (его обычно называют положительным классом), либо к классу −1 (его называют отрицательным).

Табл. 7. Данные для задачи классификации

x_1	x_2	y
0,20	0,19	1
0,33	0,37	−1
...	...	...
0,81	0,77	−1

Преположим, что есть искусственный нейрон с двумя входами. Для каждой точки по очереди можно подать одну ее координату на один вход нейрона, а другую координату – на другой вход нейрона. В результате своей работы такой нейрон выдаст предсказание для точки: либо +1, либо –1, т. е. вычисляет значение $\hat{y}$. Будем считать, если он выдал +1, то точка принадлежит положительному классу, если –1, то к отрицательному. И, соответственно, для каждой из точек можем понять, насколько нейрон ошибся. То есть мы знаем реальные значения класса – y – и мы знаем предсказанные значения класса – $\hat{y}$ (рис. 36).

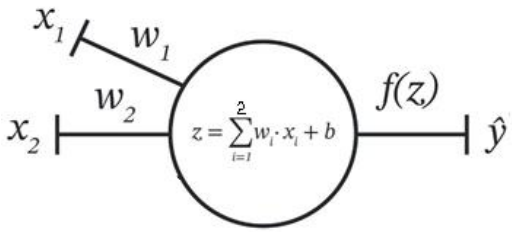


Рис. 36. Искусственный нейрон с двумя входами

Зная разницу между реальными значениями и предсказанными, можно изменить параметры нейрона для того, чтобы он научился лучше классифицировать какой-то конкретный объект.

Возможны следующие четыре комбинации соотношения $\hat{y}$ и y (табл. 8), для каждой из них предусмотрена соответствующая настройка параметров перцептрона.

Табл. 8. Настройка параметров перцептрона для задачи классификации

$\hat{y}$	y	Настройка параметров
1	1	Не нужно менять параметры
−1	−1	Не нужно менять параметры
−1	1	Нужно увеличить $\sum_{i=1}^{n} \omega_i \, x_i$
1	−1	Нужно уменьшить $\sum_{i=1}^{n} \omega_i \, x_i$

Будем смотреть на результат классификации нейроном какого-то конкретного объекта, но помним, что в обучающей выборке объектов много.

Рассмотрим различные ситуации. Если нейрон предсказал для какого-то конкретного объекта положительный (отрицательный) класс, и этот объект действительно принадлежит положительному (отрицательному) классу. В этом случае нейрон не ошибся, и у него хорошие веса. Если объект принадлежит положительному классу, а нейрон предсказал, что отрицательному, то в этой ситуации нужно как-то изменить веса нейрона. При этом вспомним, что если нейрон предсказал отрицательный класс, это значит, что аргументом в функцию активации нейрона было передано какое-то отрицательное число, и, соответственно, это число должно быть положительным, чтобы нейрон предсказал не –1, а +1. Что для этого нужно сделать? Нужно каким-то образом увеличить значение аргумента функции активации, т. е. увеличить взвешенную сумму сигналов, которые пришли в нейрон. Другая ситуация: если предсказан класс +1, а объект принадлежит отрицательному классу. Это ошибка в классификации. Надо поменять параметры: нужно уменьшить аргумент функции активации, чтобы вывести его из положтельной области в отрицательную.

Чтобы пересчитать веса перцептрона в соответствии с этими правилами, пользуются следующей формулой:

$$\omega_i^{'} = \omega_i + (y - \hat{y}) x_i \, .$$

Эта формула применяется для весов всех дендритов. Если классификация объекта прошла правильно, веса не поменяются, в противном случае все веса ω_i заменятся на новые веса $\omega_i^{'}$. Важно отметить, что объектов в выборке много, поэтому при пересчете весов нет задачи четко подогнать итоговый сигнал $\hat{y}$ под верный ответ *y*, достаточно немного сместить $\sum_{i=1}^{n} \omega_i \, x_i$ в нужную сторону.

Обычно в формулу пересчета весов добавляют параметр λ – *константу скорости обучения*:

$$\omega_i^{'} = \omega_i + \lambda (y - \hat{y}) x_i \, .$$

За счет нее можно регулировать, насколько много информации каждый объект обучающей выборки будет вносить в изменение весов. Увеличение значения параметра может привести к тому, что перцептрон обучится быстрее и при этом все еще будет решать задачу достаточно точно. Однако увеличение значения параметра может привести и к потере точности. Поэтому выбирать константу нужно аккуратно: обычно ее берут $\lambda = 0{,}1$ или $\lambda = 0{,}01$.

Помимо весов дендритов можно обучать и константу b, которая добавляется к взвешенной сумме входящих в перцептрон сигналов. Для этого в перцептрон добавляют искусственный дендрит с весом ω_b, а в данные – искусственно созданный фактор x_b, равный единице для всех объектов. Взвешенная сумма будет выглядеть так:

$$z = \sum_{i=1}^{n} \omega_i x_i + \omega_b x_b \ .$$

Теперь смысл константы b закодирован весом ω_b. Это по-прежнему константа, она одинакова для всех данных, как и b, но ее можно обучать наравне с другими весами.

Одной *итерацией обучения* перцептрона называется следующая последовательность действий. Для каждого объекта из обучающей выборки нужно:

1. Вычислить предсказание перцептрона.
2. Сравнить предсказание с реальным значением предсказываемой величины.
3. Обновить веса, если предсказание не совпало с реальностью.

Чтобы получить хорошее качество предсказания классов объектов, может потребоваться несколько итераций. Чтобы понять, в какой момент прекратить обучение перцептрона, нужны четкие критерии. Процесс обучения останавливается, если выполнено одно из двух условий:

1. Завершилось достаточно много итераций обучения (например, 1000).
2. Значение показателя качества перцептрона стало больше (или меньше, в зависимости от показателя) порогового значения.

10.3. Пример обучения перцептрона

Рассмотрим пример обучения перцептрона с двумя обычными дендритами (с весами ω_1 и ω_2) и одним искусственным дендритом (с весом ω_b), соответствующим константе b, которая добавляется к взвешенной сумме входящих в нейрон сигналов. Функция активации в примере используется стандартная:

$$f(x) = \operatorname{sign} x = \begin{cases} 1, & x > 0, \\ -1, & x \le 0. \end{cases}$$

Перцептрон обучается на наборе данных, состоящем из четырех объектов (табл. 9).

Табл. 9. Набор данных для обучения перцептрона

Точки	x_1	x_2	x_b	y
A_1	1	–1	1	1
A_2	–1	–1	1	–1
A_3	–3	1	1	–1
A_4	–1	2	1	1

В данные уже добавлен дополнительный столбец, чтобы свободный член взвешенной суммы можно было обучать. Данные удобно изобразить в виде точек на плоскости (рис. 37). Ось абсцисс – это x_1, ось ординат – x_2.

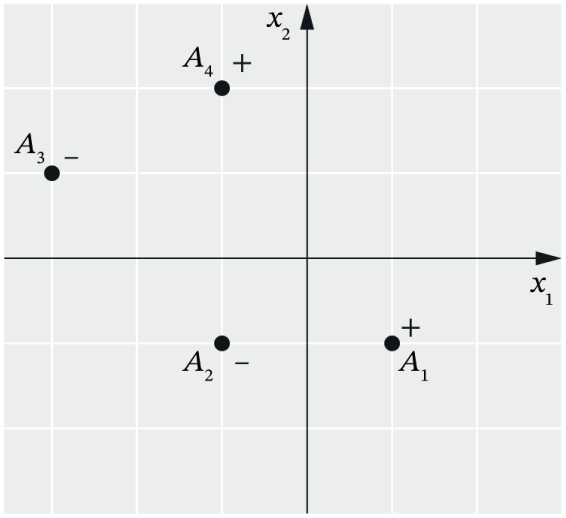


Рис. 37. Изображение объектов на плоскости

Зададим начальные параметры перцептрона:

$$\omega_1 = 0; \omega_2 = 0; \omega_b = 0; \lambda = \frac{1}{2}.$$

Проведем первую итерацию обучения: будем последовательно идти от точки A_1 к точке A_4.

Рассмотрим точку A_1 и посчитаем результат предсказания перцептрона для нее. При этом по умолчанию веса у перцептрона все равны нулю, константа скорости обучения равна $\frac{1}{2}$.

$$A_1 : \omega_1 x_1 + \omega_2 x_2 + \omega_b x_b = 0 \cdot 1 + 0 \cdot (-1) + 0 \cdot 1 = 0.$$

Теперь найдем значение функции активации от найденного значения 0:

$$f(\omega_1 x_1 + \omega_2 x_2 + \omega_b x_b) = f(0) = -1.$$

Это значение не совпадает с реальным (точка A_1 принадлежит положительному классу), поэтому веса перцептрона нужно пересчитать по формуле

$$\omega_i^{'} = \omega_i + \lambda(y - \hat{y})x_i .$$

Пересчет весов выглядит так:

$$\omega_1^{'} = \omega_1 + \lambda(y - \hat{y})x_1 = 0 + \frac{1}{2}[1-(-1)]\cdot 1 = 1;$$
$$\omega_2^{'} = \omega_2 + \lambda(y - \hat{y})x_2 = 0 + \frac{1}{2}[1-(-1)]\cdot(-1) = -1;$$
$$\omega_b^{'} = \omega_b + \lambda(y - \hat{y})x_b = 0 + \frac{1}{2}[1-(-1)]\cdot 1 = 1.$$

Теперь, используя веса, которые мы получили в результате обучения нейрона на первой точке, посмотрим на результат предсказания нейрона для второй точки. Результат предсказания перцептрона для точки $A_2(-1;-1)$:

$$f(\omega_1 x_1 + \omega_2 x_2 + \omega_b x_b) = f(1\cdot(-1) + (-1)\cdot(-1) + 1\cdot 1) = f(1) = +1.$$

Это значение не совпадает с реальным, поэтому веса перцептрона снова нужно пересчитать:

$$\omega_1^{'} = \omega_1 + \lambda(y - \hat{y})x_1 = 1 + \frac{1}{2}(-1-1)\cdot(-1) = 2;$$
$$\omega_2^{'} = \omega_2 + \lambda(y - \hat{y})x_2 = -1 + \frac{1}{2}(-1-1)\cdot(-1) = 0;$$
$$\omega_b^{'} = \omega_b + \lambda(y - \hat{y})x_b = 1 + \frac{1}{2}(-1-1)\cdot 1 = 0.$$

Повторим процесс для точек A_3 и A_4. После этого завершится первая итерация обучения перцептрона. Веса перцептрона после пересчета для точек A_3 и A_4 будут

$$\omega_1 = 1; \omega_2 = 2; \omega_b = 1.$$

Аналогично проведем вторую итерацию. По ее итогу перцептрон обучится решать задачу классификации представленного набора точек, итоговые веса будут $\omega_1 = 2, \omega_2 = 1, \omega_b = 2.$

Важно, что к концу второй итерации перестали меняться веса перцептрона. Это значит, что все объекты, что есть в обучающей выборке классифицируются правильно. Это признак того, что обучение можно остановить (веса не меняются, и лучшего результата уже не получить).

У перцептрона есть важная **геометрическая интерпретация**. Перцептрон передает функции активации взвешенную сумму координат каждой точки, а функция сравнивает эту сумму с нулем. Пограничное значение 0 достигается для точек на прямой, которая задается уравнением $\omega_1 x_1 + \omega_2 x_2 + \omega_b x_b = 0$. То есть перцептрон строит на плоскости прямую и проверяет, в какой полуплоскости относительно нее лежит каждая точка. Точки с одной стороны от прямой перцептрон классифицирует как объекты положительного класса, а с другой – как объекты отрицательного класса.

Для четырех точек из примера перцептрон построит прямую, задаваемую уравнением $2x_1 + x_2 + 2 = 0$ (рис. 38).

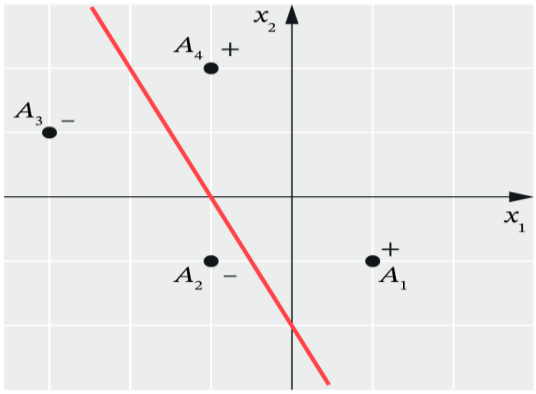


Рис. 38. Геометрическая интерпретация перцептрона

10.4. SVM как двухслойная нейронная сеть

SVM можно представить в виде следующей вычислительной структуры. Так как выход модели при произвольных ядрах вычисляется по формуле

$$x_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \dots \\ x_{il} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots \\ x_{l1} & x_{l2} & \dots & x_{ln} \end{bmatrix},$$

то имеем сеть, представленную на рис. 39.

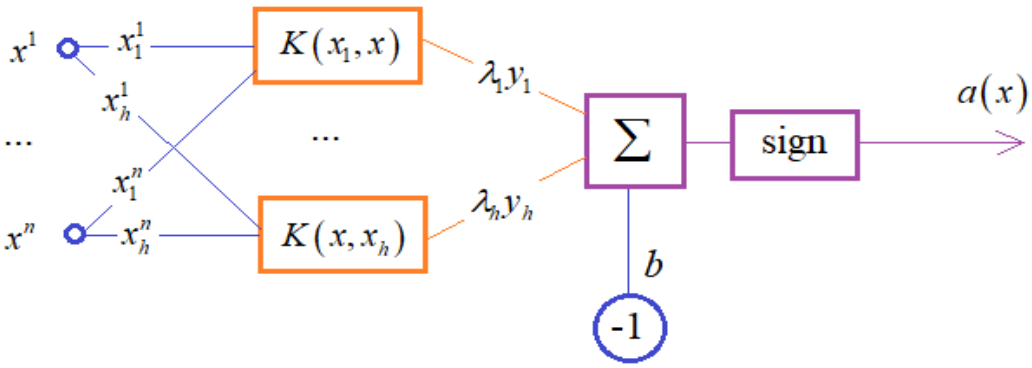


Рис. 39. Двухслойная нейронная сеть

На скрытом слое вычисляются свертки входного вектора x с опорными векторами $x_1, ..., x_h$ с учетом выбранного ядра $K(x, x')$. Затем все эти значения умножаются на весовые коэффициенты $\lambda_1 y_1, ..., \lambda_h y_h$, суммируются и пропускаются через знаковую функцию активации. SVM определяет необходимое число нейронов скрытого слоя и значения весовых коэффициентов.

ПРАКТИЧЕСКИЙ РАЗДЕЛ

Примеры

Пример 1. Расчет коэффициентов разделяющей линии и вычисление отступа для объектов разных классов.

1. Используя данные графика (рис. 40), вычислите коэффициенты

$$\omega = [\omega_0, \omega_1, \omega_2]^T$$

разделяющей линии, которая определяется выражением

$$\omega_1 \cdot x_1 + \omega_2 \cdot x_2 + \omega_0 = 0.$$

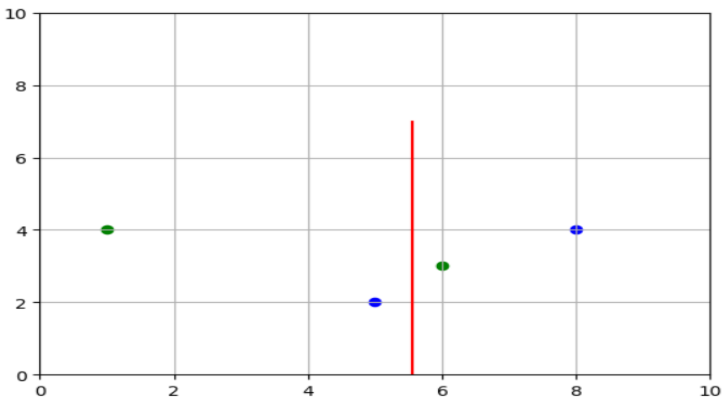


Рис. 40. Графическое изображение объектов

2. Вычислите отступы для зеленых и синих точек. Зеленые точки относятся к классу +1, синие точки относятся к классу –1. Отступ вычисляется по формуле

$$M_i = y_i \cdot \langle \omega, x_i \rangle, \quad i = 1, 2, 3, 4,$$

где $y_i \in \{-1; +1\}$ – метка класса объекта x_i.

Координаты вектора ω должны удовлетворять условию: отступ для точек, расположенных дальше от разделяющей линии, должен быть положительным, а для точек, расположенных близко к разделяющей линии, – отрицательным (см. рис. 40).

Решение. Приведем код на языке Python:

```
import numpy as np
import matplotlib.pyplot as plt

x_train = np.array([[1, 4], [5, 2], [6, 3], [8, 4]])
y_train = np.array([1, -1, 1, -1])

n_train = len(x_train)                          # размер обучающей выборки
w = [1, 0, -1]                                    # начальное значение вектора w
a = lambda x: np.sign(x[0]*w[0] + w[2])    # решающее правило
N = 1000                                          # максимальное число итераций
L = 0.01                                         # шаг изменения веса
e = 0.01                                         # небольшая добавка для w0, чтобы был зазор между разделяющей линией и граничным образом

last_error_index = -1                          # индекс последнего ошибочного наблюдения

if last_error_index > -1:
    w[0] = w[0] + e * y_train[last_error_index]
print(w)
line_y = list(range(max(x_train[:, 0])))    # формирование графика разделяющей линии
line_x = [-w[2]/w[0] for x in line_y]
x_0 = x_train[y_train == 1]                 # формирование точек для 1-го
x_1 = x_train[y_train == -1]                # и 2-го классов
plt.scatter(x_0[:, 0], x_0[:, 1], color='green')
plt.scatter(x_1[:, 0], x_1[:, 1], color='blue')
plt.plot(line_x, line_y, color='red')
plt.xlim([0, 10])
plt.ylim([0, 10])
plt.grid(True)
plt.show()
# Добавляемые координаты
new_coordinates = np.array([[0], [0], [0], [0]])
# Добавляем новые координаты к каждому вектору в x_train
x_train_extended = np.concatenate((x_train, new_coordinates), axis=1)
for i in range(n_train):
    p=y_train[i]*np.dot(w, x_train_extended[i])
    if y_train[i]<0: print('blue dot margin:')
    else : print('green dot margin')
    print(p)
```

```
green dot margin
0.17999999999999927
blue dot margin:
-0.8999999999999964
green dot margin
1.0799999999999956
blue dot margin:
-1.4399999999999942
```

Пример 2. Задача бинарной классификации (метод градиентного спуска). Даны обучающие выборки:

x_train = [[4.9, 3.3], [5.6, 4.5], [6.4, 4.3], [6.7, 5.7], [6.3, 5.0], [5.2, 3.9], [5.5, 3.7], [5.6, 3.6], [5.5, 3.8], [6.1, 4.7], [7.4, 6.1], [6.0, 5.1], [5.5, 4.4], [5.9, 5.1], [6.5, 5.8], [6.5, 4.6], [6.7, 4.4], [6.3, 5.6], [5.9, 4.8], [6.0, 4.5], [5.6, 4.1], [5.6, 4.9], [4.9, 4.5], [6.2, 4.5], [6.1, 4.7], [6.1, 4.9], [6.2, 5.4], [5.7, 4.2], [6.1, 5.6], [5.8, 4.0], [6.6, 4.6], [5.6, 4.2], [7.2, 6.1], [7.7, 6.7], [5.6, 3.9], [7.7, 6.9], [6.0, 4.0], [6.1, 4.0], [7.6, 6.6], [5.1, 3.0], [6.3, 6.0], [6.7, 5.7], [6.8, 5.9], [6.4, 5.5], [7.0, 4.7], [5.8, 5.1], [5.8, 5.1], [6.4, 5.3], [6.3, 4.9], [6.4, 5.3], [5.7, 3.5], [7.2, 5.8], [6.4, 5.6], [5.7, 4.5], [6.0, 4.5], [7.7, 6.1], [6.2, 4.3], [7.1, 5.9], [7.3, 6.3], [5.0, 3.3], [6.3, 5.1], [5.8, 3.9], [6.4, 4.5], [6.3, 5.6], [6.8, 5.5], [6.9, 5.4], [5.5, 4.0], [5.7, 4.1], [6.5, 5.5], [6.3, 4.7], [5.0, 3.5], [6.7, 5.8], [6.9, 4.9], [7.7, 6.7], [5.8, 4.1], [6.4, 5.6], [6.7, 5.2], [6.7, 4.7], [5.4, 4.5], [6.8, 4.8], [5.7, 4.2], [5.5, 4.0], [6.3, 4.9], [6.5, 5.2], [5.8, 5.1], [6.0, 4.8], [6.2, 4.8], [6.5, 5.1], [7.9, 6.4], [6.7, 5.0], [6.7, 5.6], [6.0, 5.0], [6.1, 4.6], [5.7, 5.0], [7.2, 6.0], [6.3, 4.4], [5.9, 4.2], [6.9, 5.1], [6.6, 4.4], [6.9, 5.7]]

y_train = [-1, -1, -1, 1, 1, -1, -1, -1, -1, -1, 1, -1, -1, 1, 1, -1, -1, 1, -1, -1, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1, -1, -1, 1, 1, -1, 1, -1, -1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, -1, 1, -1, -1, 1, 1, 1, -1, -1, 1, -1, -1, 1, -1, 1, -1, 1, 1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1, 1, 1, -1, -1, 1, -1, 1]

Модель линейного алгоритма, называемая линейным классификатором, должна иметь вид

$$a(x) = \text{sign}\left(\langle \omega, x \rangle\right),$$

где $\omega = \left[\omega_0, \omega_1, \omega_2\right]^T$ – вектор весовых коэффициентов модели (определяют положение разделяющей линии); $x = \left[1, x_1, x_2\right]^T$ – вектор, составленный из значений факторов объекта и единицы; $\text{sign}(v) = \begin{cases} -1, v < 0 \\ +1, v > 0 \end{cases}$ – знаковая функция.

Метки классов принимают значения $Y \in \{-1, 1\}$. Необходимо обучить модель $a(x)$, т. е. найти значения весовых коэффициентов

$$\omega = \begin{pmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{pmatrix}$$

с помощью алгоритма градиентного спуска (программы, написанной на языке Python), который должен минимизировать эмпирический риск:

$$Q(X^l) = \sum_{i=1}^{l} [y_i \neq a(x_i)] \to \min_{\omega},$$

где [•] – нотация Айверсона.

Если условие в скобках истинно, то нотация Айверсона возвращает **1**, если условие в скобках ложно, то **0**. То есть эмпирический риск $Q(X^l)$ показывает число неверных классификаций.

Так как градиентный алгоритм может минимизировать только гладкие, дифференцируемые функции, то величину $Q(X^l)$ следует сверху ограничить именно таким функционалом:

$$Q(X^l) \le \tilde{Q}(X^l) = \sum_{i=1}^{l} L\left(a\left(x_i\right), y_i\right) \to \min_{\omega},$$

где $L(a(x_i), y_i) = L(M_i)$ – выбранная функция потерь (здесь $M_i = y_i \cdot \langle \omega, x_i \rangle$ – отступ).

Функция потерь $Q(M) = (1 - M)^2$ квадратичная, ее частные производные имеют вид

$$\frac{\partial Q(M)}{\partial \omega} = -2 \cdot \left(1 - \omega^T \cdot x \cdot y\right) \cdot x^T \cdot y.$$

В качестве начальных значений весовых коэффициентов можно взять следующие:

$$\omega_0 = 0;\ \omega_1 = 0;\ \omega_2 = 1.$$

Шаг в градиентном алгоритме для коэффициента ω_0 целесообразно выбрать побольше, а для коэффициентов ω_1, ω_2 – поменьше.

Решение. Приведем код на языке Python. Графически решение представлено на рис. 41.

```
import numpy as np
import matplotlib.pyplot as plt
# квадратичная функция потерь
def loss(w, x, y):
    M = np.dot(w, x) * y
    return (1 - M) ** 2
# производная квадратичной функции потерь по вектору w
def df(w, x, y):
    M = np.dot(w, x) * y
    return -2 * (1 - M) * x * y
x_train = [[4.9, 3.3], [5.6, 4.5], [6.4, 4.3], [6.7, 5.7], [6.3, 5.0], [5.2, 3.9], [5.5, 3.7], [5.6, 3.6], [5.5, 3.8],
           [6.1, 4.7], [7.4, 6.1], [6.0, 5.1], [5.5, 4.4], [5.9, 5.1], [6.5, 5.8], [6.5, 4.6], [6.7, 4.4], [6.3, 5.6],
           [5.9, 4.8], [6.0, 4.5], [5.6, 4.1], [5.6, 4.9], [4.9, 4.5], [6.2, 4.5], [6.1, 4.7], [6.1, 4.9], [6.2, 5.4],
           [5.7, 4.2], [6.1, 5.6], [5.8, 4.0], [6.6, 4.6], [5.6, 4.2], [7.2, 6.1], [7.7, 6.7], [5.6, 3.9], [7.7, 6.9],
           [6.0, 4.0], [6.1, 4.0], [7.6, 6.6], [5.1, 3.0], [6.3, 6.0], [6.7, 5.7], [6.8, 5.9], [6.4, 5.5], [7.0, 4.7],
           [5.8, 5.1], [5.8, 5.1], [6.4, 5.3], [6.3, 4.9], [6.4, 5.3], [5.7, 3.5], [7.2, 5.8], [6.4, 5.6], [5.7, 4.5],
           [6.0, 4.5], [7.7, 6.1], [6.2, 4.3], [7.1, 5.9], [7.3, 6.3], [5.0, 3.3], [6.3, 5.1], [5.8, 3.9], [6.4, 4.5],
           [6.3, 5.6], [6.8, 5.5], [6.9, 5.4], [5.5, 4.0], [5.7, 4.1], [6.5, 5.5], [6.3, 4.7], [5.0, 3.5], [6.7, 5.8],
           [6.9, 4.9], [7.7, 6.7], [5.8, 4.1], [6.4, 5.6], [6.7, 5.2], [6.7, 4.7], [5.4, 4.5], [6.8, 4.8], [5.7, 4.2],
           [5.5, 4.0], [6.3, 4.9], [6.5, 5.2], [5.8, 5.1], [6.0, 4.8], [6.2, 4.8], [6.5, 5.1], [7.9, 6.4], [6.7, 5.0],
           [6.7, 5.6], [6.0, 5.0], [6.1, 4.6], [5.7, 5.0], [7.2, 6.0], [6.3, 4.4], [5.9, 4.2], [6.9, 5.1], [6.6, 4.4],
           [6.9, 5.7]]
y_train = [-1, -1, -1, 1, 1, -1, -1, -1, -1, -1, 1, -1, -1, 1, 1, -1, -1, 1, -1, -1, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1,
           -1, -1, 1, 1, -1, 1, -1, -1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, -1, 1, -1,
           -1, 1, 1, 1, -1, -1, 1, -1, -1, 1, -1, 1, -1, 1, 1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1,
           1, 1, -1, -1, 1, -1, 1]
# обучающая выборка с тремя признаками (третий - константа +1)
x_train = [x + [1] for x in x_train]
x_train = np.array(x_train, dtype=float)
y_train = np.array(y_train, dtype=float)

n_train = len(x_train)  # размер обучающей выборки
w = [0.0, 0.0, 1.0]  # начальные весовые коэффициенты
nt = 0.0005  # шаг сходимости SGD
lm = 0.001  # скорость "забывания" для Q
N = 10000  # число итераций SGD
Q = np.mean([loss(w, x, y) for x, y in zip(x_train, y_train)])  # показатель качества
# Q_plot = [Q]
for i in range(N):
    k = np.random.randint(0, n_train - 1)  # случайный индекс
    ek = loss(w, x_train[k], y_train[k])  # вычисление потерь для выбранного вектора
    w = w - nt * df(w, x_train[k], y_train[k])  # корректировка весов по SGD
    Q = lm * ek + (1 - lm) * Q  # пересчет показателя качества
    # Q_plot.append(Q)
print(w)
print(Q)
# print(Q_plot)
x_min = min(x_train[:, 0])
x_max = max(x_train[:, 0])
line_x = np.linspace(0, 10, 100)  # Adjust the range as needed
line_y = [-x * w[0] / w[1] - w[2] / w[1] for x in line_x]
# line_x = list(range(int(max(x_train[:, 0]))))  # формирование графика разделяющей линии
# line_y = [-x * w[0] / w[1] - w[2] / w[1] for x in line_x]
x_0 = x_train[y_train == 1]  # формирование точек для 1-го
x_1 = x_train[y_train == -1]  # и 2-го классов
plt.scatter(x_0[:, 0], x_0[:, 1], color='red')
plt.scatter(x_1[:, 0], x_1[:, 1], color='blue')
plt.plot(line_x, line_y, color='green')
plt.xlim([4, 9])
plt.ylim([2.5, 7.5])
plt.ylabel("Length")
plt.xlabel("Width")
plt.grid(True)
plt.show()
```

```
[-1.06518974  1.30493707  0.37517423]
0.44053389975997104
```

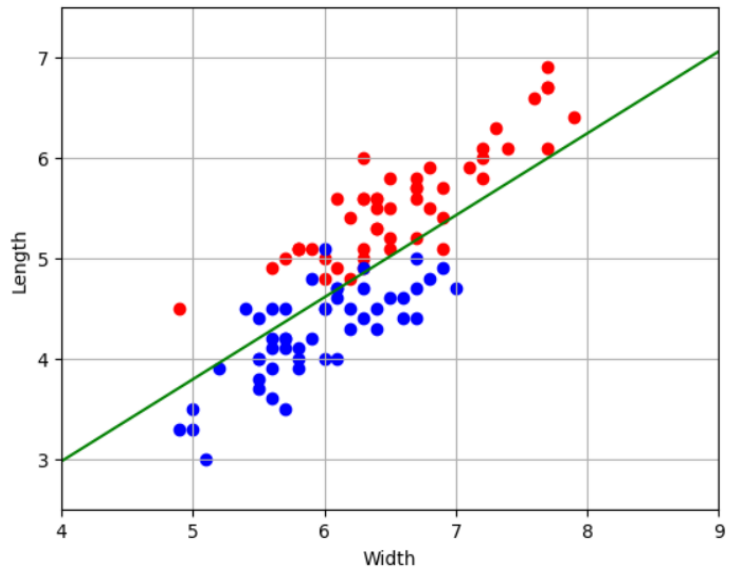


Рис. 41. Решение задачи бинарной классификации методом градиентного спуска

Пример 3. Задача классификации (метод опорных векторов).

Реализуйте на языке Python (с применением пакета scikit-learn) линейный вариант метода опорных векторов для данных обучающей выборки из примера 2. Вычислите количество и долю неверных классификаций для данной обучающей выборки. Отобразите на плоскости объекты обучающей выборки и разделяющую линию, полученную в результате обучения (точки, изображающие объекты разных классов, должны иметь разные маркеры и цвет).

Решение. Приведем код на языке Python. Графически решение представлено на рис. 42.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

# обучающая выборка с тремя признаками (третий - константа +1)
x_train = [(4.9, 3.3), (5.6, 4.5), (6.4, 4.3), (6.7, 5.7), (6.3, 5.0), (5.2, 3.9), (5.5, 3.7), (5.6, 3.6), (5.5, 3.8),
(6.1, 4.7), (7.4, 6.1), (6.0, 5.1), (5.5, 4.4), (5.9, 5.1), (6.5, 5.8), (6.5, 4.6), (6.7, 4.4), (6.3, 5.6),
(5.9, 4.8), (6.0, 4.5), (5.6, 4.1), (5.6, 4.9), (4.9, 4.5), (6.2, 4.5), (6.1, 4.7), (6.1, 4.9), (6.2, 5.4),
(5.7, 4.2), (6.1, 5.6), (5.8, 4.0), (6.6, 4.6), (5.6, 4.2), (7.2, 6.1), (7.7, 6.7), (5.6, 3.9), (7.7, 6.9),
(6.0, 4.0), (6.1, 4.0), (7.6, 6.6), (5.1, 3.0), (6.3, 6.0), (6.7, 5.7), (6.8, 5.9), (6.4, 5.5), (7.0, 4.7),
(5.8, 5.1), (5.8, 5.1), (6.4, 5.3), (6.3, 4.9), (6.4, 5.3), (5.7, 3.5), (7.2, 5.8), (6.4, 5.6), (5.7, 4.5),
(6.0, 4.5), (7.7, 6.1), (6.2, 4.3), (7.1, 5.9), (7.3, 6.3), (5.0, 3.3), (6.3, 5.1), (5.8, 3.9), (6.4, 4.5),
(6.3, 5.6), (6.8, 5.5), (6.9, 5.4), (5.5, 4.0), (5.7, 4.1), (6.5, 5.5), (6.3, 4.7), (5.0, 3.5), (6.7, 5.8),
(6.9, 4.9), (7.7, 6.7), (5.8, 4.1), (6.4, 5.6), (6.7, 5.2), (6.7, 4.7), (5.4, 4.5), (6.8, 4.8), (5.7, 4.2),
(5.5, 4.0), (6.3, 4.9), (6.5, 5.2), (5.8, 5.1), (6.0, 4.8), (6.2, 4.8), (6.5, 5.1), (7.9, 6.4), (6.7, 5.0),
(6.7, 5.6), (6.0, 5.0), (6.1, 4.6), (5.7, 5.0), (7.2, 6.0), (6.3, 4.4), (5.9, 4.2), (6.9, 5.1), (6.6, 4.4),
(6.9, 5.7)]
y_train = [-1, -1, -1, 1, 1, -1, -1, -1, -1, -1, 1, -1, -1, 1, 1, -1, -1, 1, -1, -1, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1,
-1, -1, -1, 1, 1, -1, 1, -1, -1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, -1, 1, -1,
-1, 1, 1, 1, -1, -1, 1, -1, -1, 1, -1, 1, -1, 1, 1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1,
1, 1, -1, -1, 1, -1, 1]
x_train = [x + (1,) for x in x_train]
```

```
clf = svm.SVC(kernel='linear') # SVM с линейным ядром
clf.fit(x_train, y_train) # нахождение вектора w по обучающей выборке

y_pr = clf.predict(x_train) # проверка на обучающей выборке
errors = np.array(y_train) - np.array(y_pr) # Вычисление ошибок
v = clf.support_vectors_ # выделение опорных векторов
# Формирование точек для визуализации
x_train = np.array(x_train)
y_train = np.array(y_train)
x_0_correct = x_train[(y_train == 1) & (errors == 0)] # Правильно классифицированные точки класса 1
x_1_correct = x_train[(y_train == -1) & (errors == 0)] # Правильно классифицированные точки класса -1
x_0_wrong = x_train[(y_train == 1) & (errors != 0)] # Неправильно классифицированные точки класса 1
x_1_wrong = x_train[(y_train == -1) & (errors != 0)] # Неправильно классифицированные точки класса -1
plt.scatter(x_0_correct[:, 0], x_0_correct[:, 1], color='lightblue', label='Correctly Classified (1)')
plt.scatter(x_1_correct[:, 0], x_1_correct[:, 1], color='lightcoral', label='Correctly Classified (-1)')
plt.scatter(x_0_wrong[:, 0], x_0_wrong[:, 1], color='blue', marker='x', label='Misclassified (1)')
plt.scatter(x_1_wrong[:, 0], x_1_wrong[:, 1], color='red', marker='x', label='Misclassified (-1)')
len_wrong = len(x_1_wrong) + len(x_0_wrong)
print(str(len_wrong) + " / " + str(len(x_train)))
print(str(int(len_wrong / len(x_train) * 100)) + "%")
# plt.scatter(v[:, 0], v[:, 1], s=70, edgecolor='black', linewidths=1, marker='s', label='Support Vec-tors')
support_indices = clf.support_
support_errors = errors[support_indices]
correct_support_indices = support_indices[support_errors == 0]
correct_support_vectors = clf.support_vectors_[support_errors == 0]
plt.scatter(correct_support_vectors[:, 0], correct_support_vectors[:, 1], s=150, facecolors='none', edgecolors='k',
label='Correct Support Vectors')
# Построение линии регрессии
w = clf.coef_[0]
b = clf.intercept_[0]
x_values = np.linspace(0, 10, 100)
y_values = -(w[0] * x_values + b) / w[1]
plt.plot(x_values, y_values, label='Regression', color='green')
print(w)
plt.xlim([4, 9])
plt.ylim([2.5, 7.5])
plt.ylabel("Length")
plt.xlabel("Width")
plt.legend()
plt.grid(True)
plt.show()
```

```
7 / 100
7%
[-6.45394222e-01  3.22510294e+00 -1.77635684e-15]
```

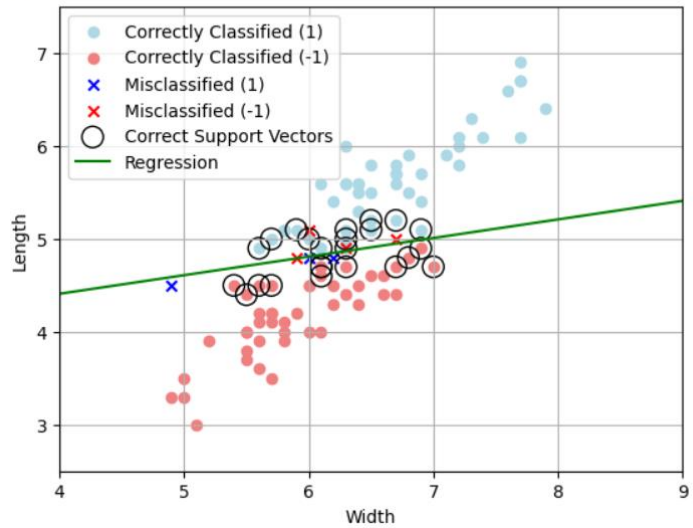


Рис. 42. Решение задачи бинарной классификации методом опорных векторов

Пример 4. Задача классификации (наивный байесовский классификатор).

Реализуйте на языке Python наивный байесовский классификатор на основе данных обучающей выборки из примера 2. Будем считать, что признаки независимы и распределены по гауссовскому закону (нормальной плотности распределения вероятностей). Посчитайте количество и долю неверных классификаций для данной обучающей выборки. Отобразите на плоскости объекты обучающей выборки (точки, изображающие объекты разных классов, должны иметь разные маркеры и цвет).

Решение. Приведем код на языке Python. Графически решение представлено на рис. 43.

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.naive_bayes import GaussianNB

x_train = [(4.9, 3.3), (5.6, 4.5), (6.4, 4.3), (6.7, 5.7), (6.3, 5.0), (5.2, 3.9), (5.5, 3.7), (5.6, 3.6), (5.5, 3.8),
(6.1, 4.7), (7.4, 6.1), (6.0, 5.1), (5.5, 4.4), (5.9, 5.1), (6.5, 5.8), (6.5, 4.6), (6.7, 4.4), (6.3, 5.6),
(5.9, 4.8), (6.0, 4.5), (5.6, 4.1), (5.6, 4.9), (4.9, 4.5), (6.2, 4.5), (6.1, 4.7), (6.1, 4.9), (6.2, 5.4),
(5.7, 4.2), (6.1, 5.6), (5.8, 4.0), (6.6, 4.6), (5.6, 4.2), (7.2, 6.1), (7.7, 6.7), (5.6, 3.9), (7.7, 6.9),
(6.0, 4.0), (6.1, 4.0), (7.6, 6.6), (5.1, 3.0), (6.3, 6.0), (6.7, 5.7), (6.8, 5.9), (6.4, 5.5), (7.0, 4.7),
(5.8, 5.1), (5.8, 5.1), (6.4, 5.3), (6.3, 4.9), (6.4, 5.3), (5.7, 3.5), (7.2, 5.8), (6.4, 5.6), (5.7, 4.5),
(6.0, 4.5), (7.7, 6.1), (6.2, 4.3), (7.1, 5.9), (7.3, 6.3), (5.0, 3.3), (6.3, 5.1), (5.8, 3.9), (6.4, 4.5),
(6.3, 5.6), (6.8, 5.5), (6.9, 5.4), (5.5, 4.0), (5.7, 4.1), (6.5, 5.5), (6.3, 4.7), (5.0, 3.5), (6.7, 5.8),
(6.9, 4.9), (7.7, 6.7), (5.8, 4.1), (6.4, 5.6), (6.7, 5.2), (6.7, 4.7), (5.4, 4.5), (6.8, 4.8), (5.7, 4.2),
(5.5, 4.0), (6.3, 4.9), (6.5, 5.2), (5.8, 5.1), (6.0, 4.8), (6.2, 4.8), (6.5, 5.1), (7.9, 6.4), (6.7, 5.0),
(6.7, 5.6), (6.0, 5.0), (6.1, 4.6), (5.7, 5.0), (7.2, 6.0), (6.3, 4.4), (5.9, 4.2), (6.9, 5.1), (6.6, 4.4),
(6.9, 5.7)]
y_train = [-1, -1, -1, 1, 1, -1, -1, -1, -1, -1, 1, -1, -1, 1, 1, -1, -1, 1, -1, -1, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1,
-1, -1, 1, 1, -1, 1, -1, -1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, -1, 1, -1,
-1, 1, 1, 1, -1, -1, 1, -1, -1, 1, -1, 1, -1, 1, 1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1,
1, 1, -1, -1, 1, -1, 1]
x_train = np.array(x_train, dtype=float)
y_train = np.array(y_train, dtype=float)
# Обучение наивного байесовского классификатора
model = GaussianNB()
model.fit(x_train, y_train)
x_0 = x_train[y_train == 1] # формирование точек для 1-го
x_1 = x_train[y_train == -1] # и 2-го классов
test = [(7, 7), (5, 3), (6, 7), (5, 4)]
test = np.array(test)
plt.scatter(x_0[:, 0], x_0[:, 1], color='red', label='Class 1')
plt.scatter(x_1[:, 0], x_1[:, 1], color='blue', label='Class -1')
plt.scatter(test[:, 0], test[:, 1], color="green", label='Test data', s=100)
# Предсказание классов для новых данных
predictions = model.predict(test)
# Вывод предсказанных классов
combined = np.column_stack((test, predictions))
print(combined)
plt.xlim([4, 9])
plt.ylim([2.5, 7.5])
plt.ylabel("Length")
plt.xlabel("Width")
plt.legend()
plt.grid(True)
plt.show()
```

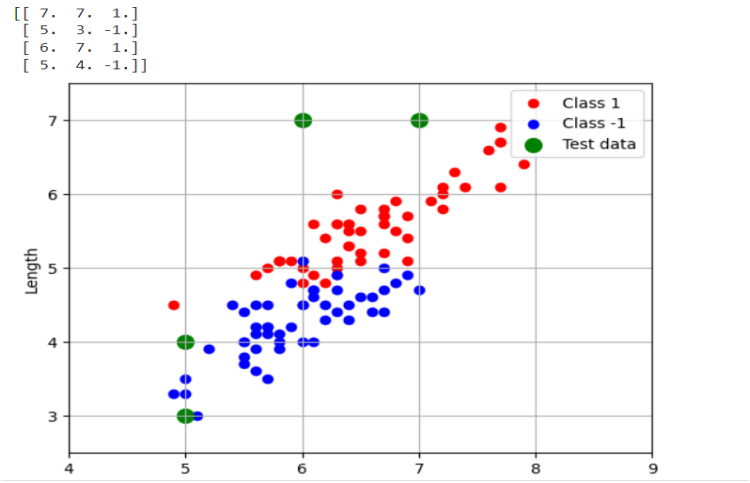


Рис. 43. Решение задачи классификации (наивный байесовский классификатор)

Пример 5. Исследование работы L2-регуляризатора в задачах регрессии. Дана функция

$$y(x) = -x^4 + 100x^2 + x, \qquad x \in [0;10;0{,}1].$$

1. Аппроксимируйте функцию $y(x)$ с помощью линейной модели

$$a(x) = \omega_0 + \sum_{i=1}^{13} \omega_i x^i,$$

т. е. полиномом 13-й степени. Здесь ω_i (i = 0, 1, 2, …, 13) – весовые коэффициенты, которые требуется найти с помощью градиентного алгоритма по обучающему набору данных.

2. Обучающую выборку составьте из всех четных индексов сгенерированных значений функции

$$X^l : \left\{ \left(x_{2i}, y = f\left(x_{2i}\right) \right)_i^{\frac{l}{2}} \right\}.$$

То есть сначала формируется первое значение x_0 с целевым значением $y_0 = f(x_0)$, затем второе $x_2, y_2 = f(x_2)$ и так до конца диапазона.

3. Вычислите значения весовых коэффициентов ω_i (i = 0, 1, 2, …, 13) для квадратичной функции потерь (в задачах регрессии обычно используют именно такую функцию потерь), минимизирующую эмпирический риск:

$$Q(X^l) = \frac{1}{2} \sum_{i=1}^{l} (y_i - a(x_i))^2 \to \min_{\omega}.$$

Весовые коэффициенты вычисляются по формуле

$$\omega_* = \left(X^T \cdot X\right)^{-1} \cdot X^T \cdot Y,$$

где X – входные векторы обучающей выборки; Y – вектор (или матрица) целевых значений обучающей выборки:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_l \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots \\ x_{l1} & x_{l2} & \dots & x_{ln} \end{bmatrix}, Y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_l \end{bmatrix}.$$

4. Вычислите прогнозы функции с помощью полученной модели $a(x)$ для всего диапазона значений. В отсчетах, не участвующих в выборке, значения модели должны сильно расходиться с целевыми.

5. Вычислите коэффициенты вектора ω с L2-регуляризатором по формуле

$$\omega_* = \left(X^T \cdot X + \lambda \cdot I\right)^{-1} \cdot X^T \cdot Y,$$

где $\lambda > 0$ – коэффициент регуляризации; $I_{n \times n}$ – единичная матрица.

6. Для новой модели $a(x)$ повторите вычисление прогнозов функции для всего диапазона значений.

Решение. Приведем код на языке Python. Графически решение представлено на рис. 44.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 10.1, 0.1)
y = np.array([-a ** 4 + 100 * a ** 2 + a for a in x])
x_train, y_train = x[::2], y[::2]
N = 14
L = 500
X = np.array([[a ** n for n in range(N)] for a in x])
IL = np.array([[L if i == j else 0 for j in range(N)] for i in range(N)])
# print(IL)
IL[0][0] = 0
X_train = X[::2]
Y = y_train

A = np.linalg.inv(X_train.T @ X_train + IL)
w = Y @ X_train @ A
print(w)

yy = [np.dot(w, x) for x in X]
plt.plot(x, yy, color = "red")
plt.plot(x, y , color = "green")
plt.grid(True)
plt.show()
```

```
[ 9.99394431e+01  1.24601700e+00  2.59378355e+00  4.42380374e+00
  6.15952795e+00  5.86943806e+00  1.30197721e+00 -3.90424302e+00
  1.74840266e+00 -3.83371938e-01  4.82524391e-02 -3.56110232e-03
  1.43795506e-04 -2.45907236e-06]
```

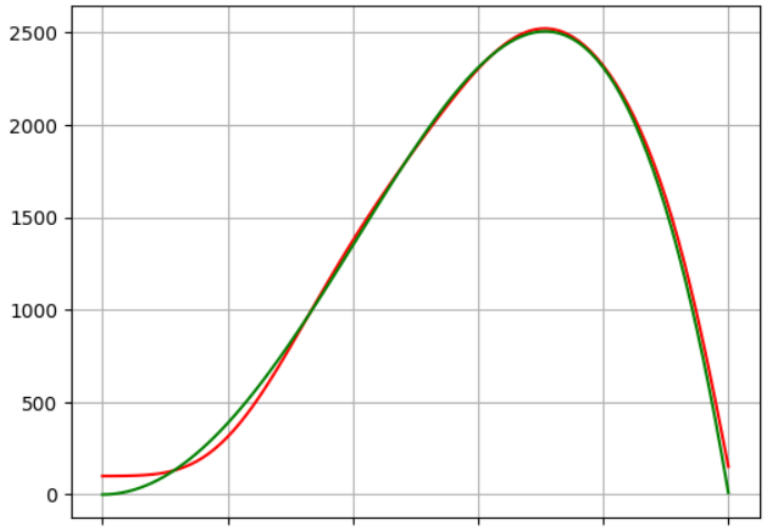


Рис. 44. Пример работы L_2

Задания для самостоятельного решения

Задание 1. Основы языка программирования Python.

Вариант 1. 1. Выведите на экран все простые числа в заданном диапазоне (диапазон вводится с клавиатуры). 2. С клавиатуры вводится текст. Определите, сколько в нем гласных, а сколько – согласных. Посчитайте количество слов в тексте.

Вариант 2. 1. Выведите на экран 1001 простое число. 2. Вводится строка, содержащая буквы, целые неотрицательные числа и иные символы. Все числа, которые встречаются в строке, отдельно выведите на экран.

Вариант 3. 1. Определите, сколько в числе четных цифр, а сколько – нечетных. Число вводится с клавиатуры. 2. В кортеже целых чисел найдите максимальный и минимальный элементы.

Вариант 4. 1. Вычислите сумму цифр введенного натурального числа. 2. Преобразуйте текст в кортеж слов с удалением знаков препинания.

Вариант 5. 1. Найдите произведение элементов списка с нечетными номерами. Найдите наибольший элемент списка. 2. С клавиатуры вводится текст. Определите, сколько в нем гласных, а сколько – согласных.

Вариант 6. 1. Создайте кортеж из 10 случайных чисел. Найдите его максимальный и минимальный элементы. 2. С клавиатуры вводится строка. Посчитайте количество слов, которые имеют нечетное количество букв.

Вариант 7. 1. Найдите сумму нечетных цифр введенного натурального числа. 2. С клавиатуры вводится строка. Посчитайте количество слов, которые имеют нечетное количество букв.

Вариант 8. 1. Выведите на экран все делители числа. Число вводится с клавиатуры. 2. С клавиатуры вводится строка. Посчитайте количество слов, которые имеют четное количество букв.

Вариант 9. 1. Дан список чисел. Посчитайте, сколько в нем пар элементов, равных друг другу. 2. С клавиатуры вводится строка. Выведите на экран самое длинное слово.

Вариант 10. 1. Дан список целых чисел. Выведите на экран кортеж уникальных элементов списка в обратном порядке. 2. Найдите сумму нечетных цифр введенного натурального числа.

Задание 2. Расчет коэффициентов разделяющей линии и вычисление отступа (margin) для объектов разных классов.

1. Используя данные графиков (табл. 10), вычислите коэффициенты

$$\omega = [\omega_0, \omega_1, \omega_2]^T$$

разделяющей линии, которая определяется выражением

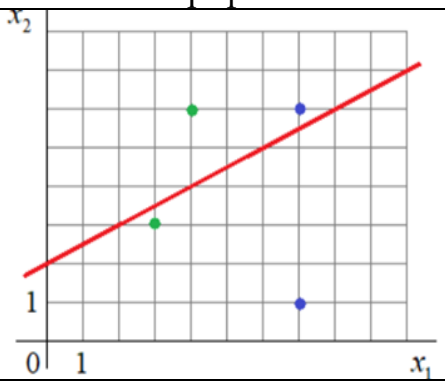
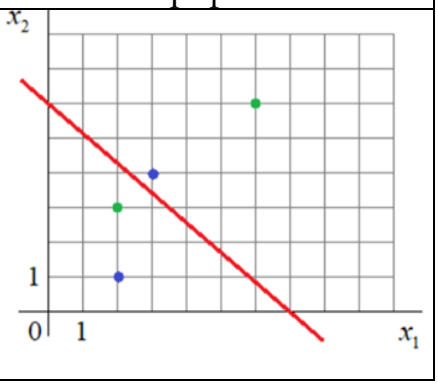
$$\omega_1 \cdot x_1 + \omega_2 \cdot x_2 + \omega_0 = 0.$$

2. Вычислите отступы для зеленых и синих точек. Зеленые точки относятся к классу +1, синие точки относятся к классу –1. Отступ вычисляется по формуле

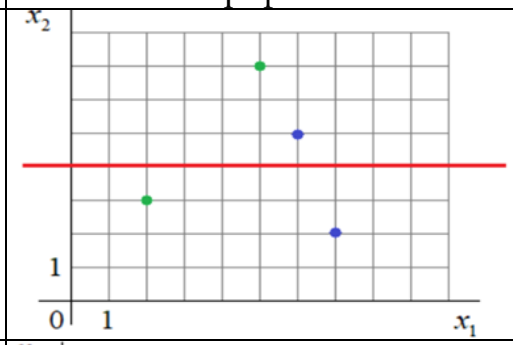
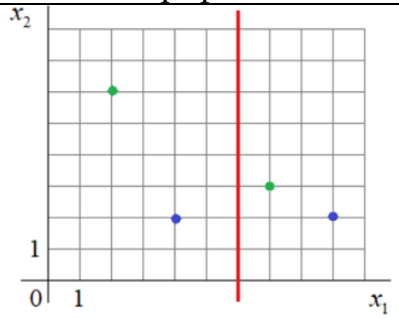
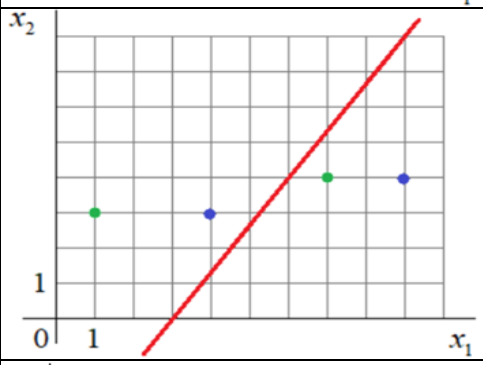
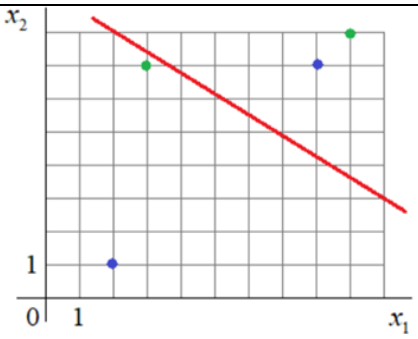
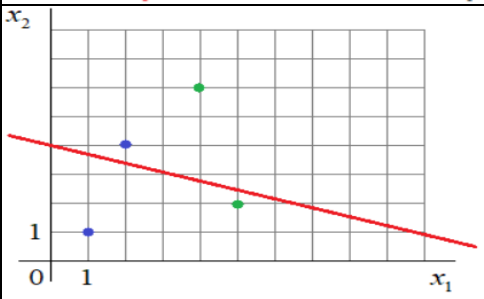
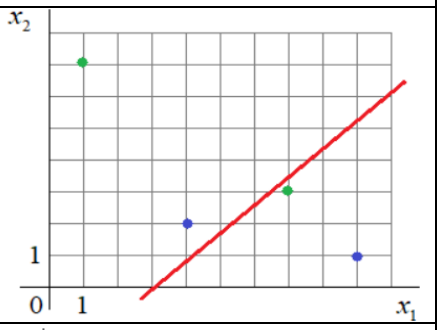
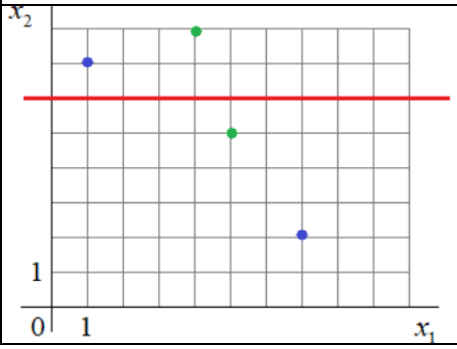
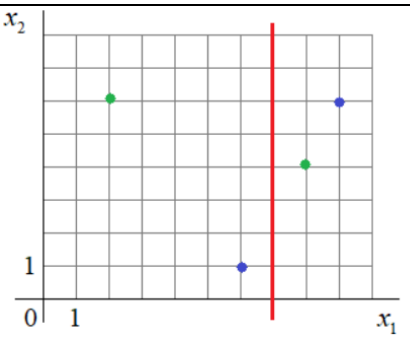
$$M_i = y_i \cdot \langle \omega, x_i \rangle, \quad i = 1, 2, 3, 4,$$

где $y_i \in \{-1; +1\}$ – метка класса объекта x_i; $\langle \omega, x_i \rangle$ – скалярное произведение векторов ω и x_i. Координаты вектора ω должны удовлетворять условию: отступ для точек, расположенных дальше от разделяющей линии, должен быть положительным, а для точек, расположенных близко к разделяющей линии, – отрицательным.

Табл. 10. Графики разделяющих линий

Вариант	Графики	Вариант	Графики
1		6	

Окончание табл. 10

Вариант	Графики	Вариант	Графики
2		7	
3		8	
4		9	
5		10	

Содержание отчета

1. Титульный лист с названием лабораторной работы, номером своего варианта, фамилией студента и номером группы.
2. Расчеты для весов разделяющей линии.
3. Расчеты для отступов.
4. Выводы по полученным результатам.

Ниже приведены обучающие выборки, необходимые для выполнения заданий 3–6 (по вариантам).

Вариант 1

```
data_x = [(5.8, 2.7), (6.7, 3.1), (5.7, 2.9), (5.5, 2.4), (4.8, 3.4), (5.4,
3.4), (4.8, 3.0), (5.5, 2.5), (5.3, 3.7), (7.0, 3.2), (5.6, 2.9), (4.9, 3.1),
(4.8,3.0), (5.0, 2.3), (5.2, 3.4), (5.1, 3.8), (5.0, 3.0), (5.0, 3.3), (4.6,
3.1), (5.5,2.6), (5.0, 3.5), (6.7, 3.0), (6.0, 2.2), (4.8, 3.1), (6.4, 2.9),
(5.6, 3.0), (4.4,3.0), (4.9, 2.4), (5.6, 3.0), (5.0, 3.6), (5.1, 3.3), (5.8,
4.0), (5.5, 2.4), (5.2,2.7), (5.1, 3.8), (5.1, 3.5), (5.5, 4.2), (4.9, 3.1),
(5.9, 3.2), (5.7, 2.6), (4.7,3.2), (5.4, 3.9), (5.8, 2.6), (5.1, 3.4), (6.4,
3.2), (5.8, 2.7), (5.6, 2.7), (5.7,2.8), (5.4, 3.0), (5.0, 3.2), (4.6, 3.4),
(6.0, 2.7), (6.6, 3.0), (4.9, 3.0), (4.9,3.6), (4.4, 3.2), (5.4, 3.4), (6.0,
3.4), (5.9, 3.0), (6.1, 2.8), (5.1, 3.7), (5.5,3.5), (6.1, 3.0), (6.2, 2.2),
(5.7, 3.0), (5.2, 3.5), (5.4, 3.7), (4.6, 3.2), (5.2,4.1), (5.0, 2.0), (6.8,
2.8), (5.0, 3.5), (6.7, 3.1), (6.3, 3.3), (6.0, 2.9), (4.7,3.2), (6.6, 2.9),
(5.6, 2.5), (4.4, 2.9), (6.2, 2.9), (6.1, 2.9), (4.3, 3.0), (6.9,3.1), (5.7,
3.8), (5.4, 3.9), (6.1, 2.8), (4.6, 3.6), (5.5, 2.3), (4.8, 3.4), (6.5,2.8),
(6.3, 2.5), (5.1, 3.8), (5.7, 4.4), (5.0, 3.4), (4.5, 2.3), (5.7, 2.8),
(5.1,2.5), (5.1, 3.5), (6.3, 2.3), (5.0, 3.4)]

data_y = [1, 1, 1, 1, -1, -1, -1, 1, -1, 1, 1, -1, -1, 1, -1, -1, -1, -1, -1, 1,
-1, 1, 1, -1, 1, 1, -1, 1, 1, -1, -1, -1, 1, 1, -1, -1, -1, -1, 1, 1, -1, -1, 1,
-1, 1, 1, 1, 1, 1, -1, -1, 1, 1, -1, -1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1,
-1, -1, -1, 1, 1, -1, 1, 1, 1, -1, 1, 1, -1, 1, 1, -1, 1, -1, -1, 1, -1, 1, -1,
1, 1, -1, -1, -1, -1, 1, 1, -1, 1, -1]
```

Вариант 2

```
data_x = [(4.9, 3.3), (5.6, 4.5), (6.4, 4.3), (6.7, 5.7), (6.3, 5.0), (5.2,
3.9), (5.5, 3.7), (5.6, 3.6), (5.5, 3.8), (6.1, 4.7), (7.4, 6.1), (6.0, 5.1),
(5.5,4.4), (5.9, 5.1), (6.5, 5.8), (6.5, 4.6), (6.7, 4.4), (6.3, 5.6), (5.9,
4.8), (6.0,4.5), (5.6, 4.1), (5.6, 4.9), (4.9, 4.5), (6.2, 4.5), (6.1, 4.7),
(6.1, 4.9), (6.2,5.4), (5.7, 4.2), (6.1, 5.6), (5.8, 4.0), (6.6, 4.6), (5.6,
4.2), (7.2, 6.1), (7.7,6.7), (5.6, 3.9), (7.7, 6.9), (6.0, 4.0), (6.1, 4.0),
(7.6, 6.6), (5.1, 3.0), (6.3,6.0), (6.7, 5.7), (6.8, 5.9), (6.4, 5.5), (7.0,
4.7), (5.8, 5.1), (5.8, 5.1), (6.4,5.3), (6.3, 4.9), (6.4, 5.3), (5.7, 3.5),
(7.2, 5.8), (6.4, 5.6), (5.7, 4.5), (6.0,4.5), (7.7, 6.1), (6.2, 4.3), (7.1,
5.9), (7.3, 6.3), (5.0, 3.3), (6.3, 5.1), (5.8,3.9), (6.4, 4.5), (6.3, 5.6),
(6.8, 5.5), (6.9, 5.4), (5.5, 4.0), (5.7, 4.1), (6.5,5.5), (6.3, 4.7), (5.0,
3.5), (6.7, 5.8), (6.9, 4.9), (7.7, 6.7), (5.8, 4.1), (6.4,5.6), (6.7, 5.2),
(6.7, 4.7), (5.4, 4.5), (6.8, 4.8), (5.7, 4.2), (5.5, 4.0), (6.3,4.9), (6.5,
5.2), (5.8, 5.1), (6.0, 4.8), (6.2, 4.8), (6.5, 5.1), (7.9, 6.4), (6.7,5.0),
(6.7, 5.6), (6.0, 5.0), (6.1, 4.6), (5.7, 5.0), (7.2, 6.0), (6.3, 4.4),
(5.9,4.2), (6.9, 5.1), (6.6, 4.4), (6.9, 5.7)]

data_y = [-1, -1, -1, 1, 1, -1, -1, -1, -1, -1, 1, -1, -1, 1, 1, -1, -1, 1, -1,
-1, -1, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1, -1, -1, 1, 1, -1, 1, -1, -1, 1, -1, 1, 1,
1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, -1, 1, -1, -1, 1, 1, 1,
-1, -1, 1, -1, -1, 1, -1, 1, -1, 1, 1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1,
-1, 1, 1, -1, 1, 1, -1, -1, 1, -1, 1]
```

Вариант 3

```
data_x = [(7.2, 2.5), (6.4, 2.2), (6.3, 1.5), (7.7, 2.2), (6.2, 1.8), (5.7,
1.3), (7.1, 2.1), (5.8, 2.4), (5.2, 1.4), (5.9, 1.5), (7.0, 1.4), (6.8, 2.1),
```

```
(7.2, 1.6), (6.7, 2.4), (6.0, 1.5), (5.1, 1.1), (6.6, 1.3), (6.1, 1.4), (6.7,
2.1), (6.4, 1.8), (5.6, 1.3), (6.9, 2.3), (6.4, 1.9), (6.9, 2.3), (6.5, 2.2),
(6.0,1.5), (5.6, 1.1), (5.6, 1.5), (6.0, 1.0), (6.0, 1.8), (6.7, 2.5), (7.7,
2.3), (5.5,1.1), (5.8, 1.0), (6.9, 2.1), (6.6, 1.4), (6.3, 1.6), (6.1, 1.4),
(5.0, 1.0), (7.7,2.0), (4.9, 1.7), (7.2, 1.8), (6.8, 1.4), (6.1, 1.2), (5.8,
1.9), (6.3, 2.5), (5.7,2.0), (6.5, 1.8), (7.6, 2.1), (6.3, 1.5), (6.7, 1.4),
(6.4, 2.3), (6.2, 2.3), (6.3,1.9), (5.5, 1.3), (7.9, 2.0), (6.7, 1.8), (6.4,
1.3), (6.5, 2.0), (6.5, 1.5), (6.9,1.5), (5.6, 1.3), (5.8, 1.2), (6.7, 2.3),
(6.0, 1.6), (5.7, 1.2), (5.7, 1.0), (5.5,1.0), (6.1, 1.4), (6.3, 1.8), (5.7,
1.3), (6.1, 1.3), (5.5, 1.3), (6.3, 1.3), (5.9,1.8), (7.7, 2.3), (6.5, 2.0),
(5.6, 2.0), (6.7, 1.7), (5.7, 1.3), (5.5, 1.2), (5.0,1.0), (5.8, 1.9), (6.2,
1.3), (6.2, 1.5), (6.3, 2.4), (6.4, 1.5), (7.4, 1.9), (6.8,2.3), (5.6, 1.3),
(5.8, 1.2), (7.3, 1.8), (6.7, 1.5), (6.3, 1.8), (6.0, 1.6), (6.4,2.1), (6.1,
1.8), (5.9, 1.8), (5.4, 1.5), (4.9, 1.0)]

data_y = [1, 1, 1, 1, 1, -1, 1, 1, -1, -1, -1, 1, 1, 1, -1, -1, -1, 1, 1, 1, -1,
1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, -1, -1, 1, -1, -1, -1, -1, 1, 1, 1, -1, -1,
1, 1, 1, 1, 1, -1, -1, 1, 1, 1, -1, 1, 1, -1, 1, -1, -1, -1, -1, 1, -1, -1, -1,
-1, -1, 1, -1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, -1, -1, 1, -1, 1, 1,
-1, -1, 1, -1, 1, -1, 1, 1, -1, -1, -1]
```

Вариант 4

```
data_x = [(5.3, 2.3), (5.7, 2.5), (4.0, 1.0), (5.6, 2.4), (4.5, 1.5), (5.4, 2.3),
(4.8, 1.8), (4.5, 1.5), (5.1, 1.5), (6.1, 2.3), (5.1, 1.9), (4.0, 1.2),
(5.2,2.0), (3.9, 1.4), (4.2, 1.2), (4.7, 1.5), (4.8, 1.8), (3.6, 1.3), (4.6, 1.4),
(4.5,1.7),(3.0, 1.1), (4.3, 1.3), (4.5, 1.3), (5.5, 2.1), (3.5, 1.0), (5.6, 2.2),
(4.2, 1.5), (5.8, 1.8), (5.5, 1.8), (5.7, 2.3), (6.4, 2.0), (5.0, 1.7), (6.7,2.0),
(4.0, 1.3), (4.4, 1.4), (4.5, 1.5), (5.6, 2.4), (5.8, 1.6), (4.6, 1.3),
(4.1,1.3), (5.1, 2.3), (5.2, 2.3), (5.6, 1.4), (5.1, 1.8), (4.9, 1.5), (6.7,
2.2), (4.4,1.3), (3.9, 1.1), (6.3, 1.8), (6.0, 1.8), (4.5, 1.6), (6.6, 2.1),
(4.1, 1.3), (4.5,1.5), (6.1, 2.5), (4.1, 1.0), (4.4, 1.2), (5.4, 2.1),
(5.0, 1.5), (5.0, 2.0), (4.9,1.5), (5.9, 2.1), (4.3, 1.3), (4.0, 1.3), (4.9, 2.0),
(4.9, 1.8), (4.0, 1.3), (5.5,1.8), (3.7, 1.0), (6.9, 2.3), (5.7, 2.1), (5.3,
1.9), (4.4, 1.4), (5.6, 1.8), (3.3,1.0), (4.8, 1.8), (6.0, 2.5), (5.9, 2.3),
(4.9, 1.8), (3.3, 1.0), (3.9, 1.2), (5.6,2.1), (5.8, 2.2), (3.8, 1.1), (3.5, 1.0),
(4.5, 1.5), (5.1, 1.9), (4.7, 1.4), (5.1,1.6), (5.1, 2.0), (4.8, 1.4),
(5.0, 1.9), (5.1, 2.4), (4.6, 1.5), (6.1, 1.9), (4.7,1.6),(4.7, 1.4), (4.7,
1.2), (4.2, 1.3), (4.2, 1.3)]

data_y = [1, 1, -1, 1, -1, 1, 1, -1, 1, 1, 1, -1, 1, -1, -1, -1, -1, -1, -1, 1,
-1, -1, -1, 1, -1, 1, -1, 1, 1, 1, 1, -1, 1, -1, -1, -1, 1, 1, -1, -1, 1, 1, 1,
1, -1, 1, -1, -1, 1, 1, -1, 1, -1, -1, 1, -1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1,
-1, 1, -1, 1, 1, 1, -1, 1, -1, 1, 1, 1, 1, -1, -1, 1, 1, -1, -1, -1, 1, -1, -1,
1, -1, 1, 1, -1, 1, -1, -1, -1, -1, -1]
```

Вариант 5

```
data_x = [(5.8, 1.2), (5.6, 1.5), (6.5, 1.5), (6.1, 1.3), (6.4, 1.3), (7.7,
2.0), (6.0, 1.8), (5.6, 1.3), (6.0, 1.6), (5.8, 1.9), (5.7, 2.0), (6.3, 1.5),
(6.2,1.8), (7.7, 2.3), (5.8, 1.2), (6.3, 1.8), (6.0, 1.0), (6.2, 1.3), (5.7,
1.3), (6.3,1.9), (6.7, 2.5), (5.5, 1.2), (4.9, 1.0), (6.1, 1.4), (6.0, 1.6),
(7.2, 2.5), (7.3,1.8), (6.6, 1.4), (5.6, 2.0), (5.5, 1.0), (6.4, 2.2), (5.6,
1.3), (6.6, 1.3), (6.9,2.1), (6.8, 2.1), (5.7, 1.3), (7.0, 1.4), (6.1, 1.4),
(6.1, 1.8), (6.7, 1.7), (6.0,1.5), (6.5, 1.8), (6.4, 1.5), (6.9, 1.5), (5.6,
1.3), (6.7, 1.4), (5.8, 1.9), (6.3,1.3), (6.7, 2.1), (6.2, 2.3), (6.3, 2.4),
(6.7, 1.8), (6.4, 2.3), (6.2, 1.5), (6.1,1.4), (7.1, 2.1), (5.7, 1.0), (6.8,
1.4), (6.8, 2.3), (5.1, 1.1), (4.9, 1.7), (5.9,1.8), (7.4, 1.9), (6.5, 2.0),
(6.7, 1.5), (6.5, 2.0), (5.8, 1.0), (6.4, 2.1), (7.6,2.1), (5.8, 2.4), (7.7,
2.2), (6.3, 1.5), (5.0, 1.0), (6.3, 1.6), (7.7, 2.3), (6.4,1.9), (6.5, 2.2),
```

```
(5.7, 1.2), (6.9, 2.3), (5.7, 1.3), (6.1, 1.2), (5.4, 1.5), (5.2,1.4), (6.7,
2.3), (7.9, 2.0), (5.6, 1.1), (7.2, 1.8), (5.5, 1.3), (7.2, 1.6), (6.3,2.5),
(6.3, 1.8), (6.7, 2.4), (5.0, 1.0), (6.4, 1.8), (6.9, 2.3), (5.5, 1.3),
(5.5,1.1), (5.9, 1.5), (6.0, 1.5), (5.9, 1.8)]

data_y = [-1, -1, -1, -1, -1, 1, 1, -1, -1, 1, 1, -1, 1, 1, -1, 1, -1, -1, -1,
1, 1, -1, -1, -1, -1, 1, 1, -1, 1, -1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, 1, 1,
-1, -1, -1, -1, 1, -1, 1, 1, 1, 1, 1, -1, -1, 1, -1, -1, 1, -1, 1, -1, 1, 1, -1,
1, -1, 1, 1, 1, 1, 1, -1, -1, 1, 1, 1, -1, 1, -1, -1, -1, -1, 1, 1, -1, 1, -1,
1, 1, 1, 1, -1, 1, 1, -1, -1, -1, -1, 1]
```

Вариант 6

```
data_x = [(2.6, 1.0), (3.0, 2.3), (3.4, 1.6), (3.0, 1.5), (2.7, 1.0), (3.8,
2.0), (3.0, 1.8), (2.8, 2.1), (2.9, 1.3), (3.0, 1.8), (3.2, 1.5), (2.7, 1.8),
(3.0,1.4), (3.3, 2.5), (2.7, 1.9), (2.6, 1.2), (3.1, 1.4), (2.7, 1.9), (3.1,
2.4), (3.0,1.5), (2.8, 1.4), (3.1, 1.5), (2.5, 1.8), (2.8, 1.3), (2.8, 1.8),
(2.2, 1.5), (3.3,2.5), (3.2, 1.8), (3.6, 2.5), (3.0, 1.7), (2.3, 1.3), (2.8,
1.3), (2.5, 1.5), (3.0,2.1), (2.0, 1.0), (2.8, 2.2), (3.0, 1.6), (3.1, 2.1),
(3.2, 2.3), (2.5, 1.7), (2.5,1.1), (3.2, 2.3), (2.8, 1.2), (2.9, 1.5), (2.6,
1.4), (2.2, 1.0), (3.3, 2.1), (2.4,1.1), (3.4, 2.4), (3.0, 1.2), (2.8, 1.5),
(3.2, 1.4), (3.8, 2.2), (2.8, 2.4), (3.0,1.8), (3.0, 1.5), (3.0, 1.4), (3.0,
2.1), (2.4, 1.0), (3.4, 2.3), (2.7, 1.4), (2.3,1.0), (2.9, 1.3), (3.3, 1.6),
(2.7, 1.3), (2.6, 1.2), (2.4, 1.0), (3.2, 2.0), (3.1,1.5), (3.0, 1.8), (2.8,
1.9), (2.7, 1.9), (2.8, 1.5), (3.2, 1.8), (3.1, 1.8), (2.5,1.1), (2.2, 1.5),
(2.9, 1.4), (3.0, 2.1), (2.9, 1.3), (2.8, 2.0), (2.5, 2.0), (3.0,2.2), (3.2,
2.3), (3.1, 2.3), (2.8, 1.3), (2.5, 1.9), (3.0, 2.3), (2.9, 1.3), (2.9,1.8),
(3.0, 2.0), (2.9, 1.8), (2.7, 1.2), (2.5, 1.3), (3.0, 1.3), (2.6, 2.3),
(2.8,2.0), (2.9, 1.3), (2.7, 1.6), (2.3, 1.3)]

data_y = [-1, 1, -1, -1, -1, 1, 1, 1, -1, 1, -1, 1, -1, 1, 1, -1, -1, 1, 1, -1,
-1, -1, 1, -1, 1, -1, 1, -1, 1, -1, -1, -1, -1, 1, -1, 1, 1, 1, 1, 1, -1, 1, -1,
-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, 1, 1, -1, -1, 1, -1, 1, -1, -1, -1, -1, -1,
-1, -1, 1, -1, 1, 1, 1, -1, 1, 1, -1, 1, -1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1,
1, 1, 1, -1, -1, -1, 1, 1, -1, -1, -1]
```

Вариант 7

```
data_x = [(3.0, 1.3), (3.4, 1.6), (3.4, 0.4), (3.7, 0.2), (3.5, 0.2), (3.4,
0.2), (3.4, 0.4), (3.9, 0.4), (3.4, 0.3), (3.2, 0.2), (2.8, 1.3), (3.5, 0.3),
(2.4,1.0), (3.0, 0.1), (3.6, 0.2), (3.2, 0.2), (2.9, 0.2), (2.9, 1.3), (2.3,
1.3), (3.8,0.2), (3.2, 1.5), (2.3, 1.0), (3.0, 1.7), (3.3, 0.2), (3.4, 0.2),
(3.8, 0.3), (2.0,1.0), (3.1, 0.2), (2.5, 1.3), (2.4, 1.1), (3.2, 0.2), (2.2,
1.0), (3.1, 1.4), (3.0,0.2), (3.0, 0.2), (3.4, 0.2), (3.7, 0.2), (2.8, 1.2),
(2.9, 1.4), (4.0, 0.2), (3.2,1.4), (3.2, 0.2), (2.9, 1.3), (2.9, 1.3), (3.5,
0.2), (3.3, 1.6), (2.9, 1.3), (2.7,1.0), (2.9, 1.3), (3.4, 0.2), (3.2, 0.2),
(4.1, 0.1), (3.5, 0.6), (2.7, 1.4), (2.3,0.3), (2.9, 1.5), (3.1, 1.5), (3.5,
0.2), (2.7, 1.6), (3.3, 0.5), (3.0, 1.4), (3.6,0.2), (3.0, 1.2), (2.8, 1.3),
(2.5, 1.1), (3.0, 1.5), (3.1, 0.2), (2.6, 1.0), (2.7,1.2), (2.2, 1.5), (3.7,
0.4), (3.4, 0.2), (3.5, 0.3), (3.6, 0.1), (2.5, 1.5), (2.6,1.2), (2.8, 1.3),
(3.1, 0.1), (2.4, 1.0), (3.1, 1.5), (2.3, 1.3), (2.8, 1.5), (3.0,0.3), (3.0,
0.2), (2.5, 1.1), (3.0, 1.5), (3.2, 1.8), (3.9, 0.4), (2.8, 1.4), (4.2,0.2),
(3.4, 0.2), (2.7, 1.3), (3.8, 0.3), (3.0, 1.4), (2.6, 1.2), (4.4, 0.4),
(3.8,0.4), (3.1, 0.2), (3.0, 0.1), (3.0, 1.5)]

data_y = [1, 1, -1, -1, -1, -1, -1, -1, -1, -1, 1, -1, 1, -1, -1, -1, -1, 1, 1,
-1, 1, 1, 1, -1, -1, -1, 1, -1, 1, 1, -1, 1, 1, -1, -1, -1, -1, 1, 1, -1, 1, -1,
1, 1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, -1, 1, 1, -1, 1, -1, 1, -1, 1, 1, 1, 1,
-1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, -1, 1, 1, 1, 1, -1, -1, 1, 1, 1, -1, 1,
-1, -1, 1, -1, 1, 1, -1, -1, -1, -1, 1]
```

Вариант 8

```
data_x = [(2.9, 6.0), (3.8, 5.1), (3.0, 4.9), (3.5, 5.0), (2.6, 5.5), (3.4,
4.6), (3.8, 5.1), (3.5, 5.5), (2.3, 5.0), (3.6, 4.9), (3.5, 5.1), (2.8, 5.7),
(3.0,5.4), (2.9, 6.4), (3.0, 4.3), (3.0, 4.8), (3.5, 5.1), (3.2, 4.7), (2.8,
5.7), (4.2,5.5), (2.5, 6.3), (2.4, 4.9), (3.1, 4.8), (3.7, 5.4), (3.0, 5.6),
(2.7, 5.6), (3.1,6.9), (2.7, 6.0), (3.4, 4.8), (2.4, 5.5), (3.3, 5.1), (2.5,
5.6), (2.9, 6.2), (3.0,5.9), (2.8, 6.1), (3.0, 4.4), (2.7, 5.2), (2.9, 5.7),
(3.3, 5.0), (3.2, 6.4), (3.4,5.2), (3.4, 5.0), (3.1, 4.9), (4.4, 5.7), (2.8,
6.1), (3.4, 5.0), (3.1, 6.7), (3.7,5.1), (3.1, 4.9), (4.0, 5.8), (2.3, 4.5),
(3.1, 6.7), (3.2, 5.0), (2.4, 5.5), (3.6,5.0), (3.9, 5.4), (3.5, 5.0), (2.6,
5.7), (2.8, 6.8), (3.9, 5.4), (2.2, 6.0), (3.2,4.4), (3.8, 5.7), (3.2, 4.7),
(2.9, 6.6), (3.0, 4.8), (2.6, 5.8), (3.0, 5.0), (3.4,5.1), (3.8, 5.1), (2.3,
6.3), (3.6, 4.6), (2.7, 5.8), (2.9, 4.4), (3.2, 4.6), (3.5,5.2), (3.1, 4.6),
(2.5, 5.5), (2.2, 6.2), (3.2, 7.0), (3.3, 6.3), (3.0, 6.1), (3.4,4.8), (3.4,
5.4), (2.3, 5.5), (2.5, 5.1), (3.4, 6.0), (2.0, 5.0), (2.9, 5.6), (2.7,5.8),
(2.8, 6.5), (3.4, 5.4), (3.7, 5.3), (4.1, 5.2), (3.0, 5.6), (3.0, 6.6),
(2.9,6.1), (3.0, 6.7), (3.0, 5.7), (3.2, 5.9)]

data_y = [1, -1, -1, -1, 1, -1, -1, -1, 1, -1, -1, 1, 1, 1, -1, -1, -1, -1, 1,
-1, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1, -1, 1, 1, 1, 1, -1, 1, 1, -1, 1, -1, -1,
-1, -1, 1, -1, 1, -1, -1, -1, -1, 1, -1, 1, -1, -1, -1, 1, 1, -1, 1, -1, -1, -1,
1, -1, 1, -1, -1, -1, 1, -1, 1, -1, -1, -1, -1, 1, 1, 1, 1, 1, -1, -1, 1, 1, 1,
1, 1, 1, 1, -1, -1, -1, 1, 1, 1, 1, 1, 1]
```

Вариант 9

```
data_x = [(1.3, 5.5), (1.5, 4.9), (4.9, 6.3), (1.5, 5.2), (3.5, 5.7), (1.4,
4.6), (4.8, 5.9), (4.5, 5.7), (3.7, 5.5), (1.5, 5.3), (4.6, 6.1), (1.6, 4.8),
(1.5,5.0), (4.0, 5.5), (1.3, 4.7), (1.4, 5.0), (1.7, 5.1), (1.5, 5.2), (3.9,
5.2), (1.5,4.6), (4.1, 5.8), (1.9, 5.1), (4.0, 5.5), (4.6, 6.6), (4.5, 6.4),
(4.5, 6.0), (4.7,6.1), (1.3, 4.5), (5.1, 6.0), (4.4, 6.6), (4.0, 6.1), (4.5,
6.2), (3.8, 5.5), (1.5,5.4), (4.9, 6.9), (3.0, 5.1), (4.5, 5.6), (1.4, 4.9),
(4.0, 5.8), (5.0, 6.7), (4.4,5.5), (3.9, 5.6), (1.4, 4.6), (3.3, 4.9), (3.9,
5.8), (4.2, 5.7), (4.4, 6.3), (1.4,5.1), (1.6, 5.0), (1.5, 5.1), (4.7, 6.3),
(3.6, 5.6), (4.4, 6.7), (1.7, 5.4), (1.3,4.4), (4.1, 5.6), (1.0, 4.6), (4.3,
6.2), (1.4, 4.4), (4.5, 6.0), (4.7, 6.7), (3.3,5.0), (1.5, 4.9), (3.5, 5.0),
(1.6, 4.7), (1.4, 4.9), (1.4, 4.8), (1.3, 5.0), (4.6,6.5), (4.0, 6.0), (4.7,
6.1), (1.6, 5.0), (1.4, 5.2), (4.7, 7.0), (1.1, 4.3), (1.6,5.1), (4.3, 6.4),
(1.2, 5.8), (1.9, 4.8), (1.4, 4.8), (1.5, 5.1), (4.8, 6.8), (4.1,5.7), (1.7,
5.7), (1.6, 5.0), (4.2, 5.7), (1.6, 4.8), (1.2, 5.0), (1.3, 4.4), (1.7,5.4),
(4.5, 5.4), (4.2, 5.6), (1.5, 5.4), (1.4, 5.5), (1.4, 5.1), (1.5, 5.1),
(4.2,5.9), (1.5, 5.7), (1.4, 5.0), (1.3, 5.4)]

data_y = [-1, -1, 1, -1, 1, -1, 1, 1, 1, -1, 1, -1, -1, 1, -1, -1, -1, -1, 1,
-1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1,
1, 1, 1, -1, -1, -1, 1, 1, 1, -1, -1, 1, -1, 1, -1, 1, 1, 1, -1, 1, -1, -1, -1,
-1, 1, 1, 1, -1, -1, 1, -1, -1, 1, -1, -1, -1, -1, 1, 1, -1, -1, 1, -1, -1, -1,
-1, 1, 1, -1, -1, -1, -1, 1, -1, -1, -1]
```

Вариант 10

```
data_x = [(3.0, 4.9), (2.7, 3.9), (3.0, 5.5), (2.6, 4.0), (2.9, 4.3), (3.1,
5.1), (2.2, 4.5), (2.3, 3.3), (2.7, 5.1), (3.3, 5.7), (2.8, 5.1), (2.8, 4.9),
(2.5,4.5), (2.8, 4.7), (3.2, 4.7), (3.2, 5.7), (2.8, 6.1), (3.6, 6.1), (2.8,
4.8), (2.9,4.5), (3.1, 4.9), (2.3, 4.4), (3.3, 6.0), (2.6, 5.6), (3.0, 4.4),
(2.9, 4.7), (2.8,4.0), (2.5, 5.8), (2.4, 3.3), (2.8, 6.7), (3.0, 5.1), (2.3,
4.0), (3.1, 5.5), (2.8,4.8), (2.7, 5.1), (2.5, 4.0), (3.1, 4.4), (3.8, 6.7),
(3.1, 5.6), (3.1, 4.7), (3.0,5.8), (3.0, 5.2), (3.0, 4.5), (2.7, 4.9), (3.0,
```

```
6.6), (2.9, 4.6), (3.0, 4.6), (2.6,3.5), (2.7, 5.1), (2.5, 5.0), (2.0, 3.5),
(3.2, 5.9), (2.5, 5.0), (3.4, 5.6), (3.4,4.5), (3.2, 5.3), (2.2, 4.0), (2.2,
5.0), (3.3, 4.7), (2.7, 4.1), (2.4, 3.7), (3.0,4.2), (3.2, 6.0), (3.0, 4.2),
(3.0, 4.5), (2.7, 4.2), (2.5, 3.0), (2.8, 4.6), (2.9,4.2), (3.1, 5.4), (2.5,
4.9), (3.2, 5.1), (2.8, 4.5), (2.8, 5.6), (3.4, 5.4), (2.7,3.9), (3.0, 6.1),
(3.0, 5.8), (3.0, 4.1), (2.5, 3.9), (2.4, 3.8), (2.6, 4.4), (2.9,3.6), (3.3,
5.7), (2.9, 5.6), (3.0, 5.2), (3.0, 4.8), (2.7, 5.3), (2.8, 4.1), (2.8,5.6),
(3.2, 4.5), (3.0, 5.9), (2.9, 4.3), (2.6, 6.9), (2.8, 5.1), (2.9, 6.3),
(3.2,4.8), (3.0, 5.5), (3.0, 5.0), (3.8, 6.4)]
data_y = [1, -1, 1, -1, -1, 1, -1, -1, -1, 1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1,
-1, -1, 1, 1, -1, -1, -1, 1, -1, 1, 1, -1, 1, 1, 1, -1, -1, 1, 1, -1, 1, 1, -1,
1, 1, -1, -1, -1, 1, 1, -1, 1, 1, 1, -1, 1, -1, 1, -1, -1, -1, -1, 1, -1, -1,
-1, -1, -1, -1, 1, -1, 1, -1, 1, 1, -1, 1, 1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1,
-1, 1, -1, 1, -1, 1, 11, 1, -1, 1, -1, 1]
```

Задание 3. Задача бинарной классификации (метод градиентного спуска).

Даны обучающие выборки и функции потерь (по вариантам) для обучения линейного алгоритма бинарной классификации образов.

Модель линейного алгоритма, называемая линейным классификатором, должна иметь вид

$$a(x) = \text{sign}\left(\langle \omega, x \rangle\right),$$

где $\omega = [\omega_0, \omega_1, \omega_2]^T$ – вектор весовых коэффициентов модели (определяют положение разделяющей линии); $x = [1, x_1, x_2]^T$ – вектор, составленный из значений факторов объекта и единицы;

$$\text{sign}(v) = \begin{cases} -1, v < 0, \\ +1, v > 0 \end{cases} \quad - \text{ знаковая функция.}$$

Метки классов принимают значения $Y \in \{-1, 1\}$. Необходимо обучить модель $a(x)$, т. е. найти значения весовых коэффициентов $\omega = [\omega_0, \omega_1, \omega_2]^T$ с помощью алгоритма градиентного спуска (программы, написанной на языке Python), который должен минимизировать эмпирический риск:

$$Q(X^l) = \sum_{i=1}^{l} \left[y_i \neq a(x_i) \right] \to \min_{\omega},$$

где [•] – нотация Айверсона. Если условие в скобках истинно, то нотация Айверсона возвращает 1, если условие в скобках ложно, то 0. То есть эмпирический риск $Q(X^l)$ показывает число неверных классификаций.

Так как градиентный алгоритм может минимизировать только гладкие, дифференцируемые функции, то величину $Q(X^l)$ следует сверху ограничить именно таким функционалом:

$$Q(X^l) \le \overline{Q}\left(X^l\right) = \sum_{i=1}^{l} L(a(x_i), y_i) \to \min_{\omega},$$

где $L(a(x_i), y_i) = L(M_i)$ – выбранная функция потерь (здесь $M_i = y_i \cdot \langle \omega, x_i \rangle$ – отступ).

Функция потерь (как и набор обучающих данных) определяется вариантом (табл. 11).

Табл. 11. Функции потерь по вариантам

Вариант	Функция потерь для реализации градиентного алгоритма	Производная функции потерь
1	$L(M) = \log_2(1 + e^{-M})$ – логарифмическая	$\frac{\partial L(M)}{\partial \omega} = -\frac{e^{-M} \cdot x^T \cdot y}{\left(1 + e^{-M}\right) \cdot \ln 2}$
2	$Q(M) = (1 - M)^2$ – квадратичная	$\frac{\partial Q(M)}{\partial \omega} = -2 \cdot (1 - \omega^T \cdot x \cdot y) \cdot x^T \cdot y$
3	$S(M) = 2 \cdot (1 + e^M)^{-1}$ – сигмоидная	$\frac{\partial S(M)}{\partial \omega} = -\frac{2e^M \cdot x^T \cdot y}{\left(1 + e^M\right)^2}$
4	$E(M) = e^{-M}$ – экспоненциальная	$\frac{\partial E(M)}{\partial \omega} = -e^{-M} \cdot x^T \cdot y$
5	$L(M) = \log_2(1 + e^{-M})$ – логарифмическая	$\frac{\partial L(M)}{\partial \omega} = -\frac{e^{-M} \cdot x^T \cdot y}{\left(1 + e^{-M}\right) \cdot \ln 2}$
6	$Q(M) = (1 - M)^2$ – квадратичная	$\frac{\partial Q(M)}{\partial \omega} = -2 \cdot (1 - \omega^T \cdot x \cdot y) \cdot x^T \cdot y$
7	$S(M) = 2 \cdot (1 + e^M)^{-1}$ – сигмоидная	$\frac{\partial S(M)}{\partial \omega} = -\frac{2e^M \cdot x^T \cdot y}{\left(1 + e^M\right)^2}$
8	$E(M) = e^{-M}$ – экспоненциальная	$\frac{\partial E(M)}{\partial \omega} = -e^{-M} \cdot x^T \cdot y$
9	$L(M) = \log_2(1 + e^{-M})$ – логарифмическая	$\frac{\partial L(M)}{\partial \omega} = -\frac{e^{-M} \cdot x^T \cdot y}{\left(1 + e^{-M}\right) \cdot \ln 2}$
10	$Q(M) = (1 - M)^2$ – квадратичная	$\frac{\partial Q(M)}{\partial \omega} = -2 \cdot (1 - \omega^T \cdot x \cdot y) \cdot x^T \cdot y$

В качестве начальных значений весовых коэффициентов можно взять следующие:

$$\omega_0 = 0,\ \omega_1 = 0,\ \omega_2 = 1.$$

Шаг в градиентном алгоритме для коэффициента ω_0 целесообразно выбрать побольше, а для коэффициентов ω_1, ω_2 поменьше.

Содержание отчета

1. Математические расчеты, необходимые для реализации алгоритма обучения.
2. Текст программы обучения линейной модели с использованием градиентного алгоритма на языке Python.
3. Результаты работы программы в виде графика множества точек обучающей выборки (каждый класс точек должен быть представлен разными маркерами и цветами) и полученной разделяющей линии.
4. Выводы по полученным результатам.

Задание 4. Задача классификации (метод опорных векторов).

Реализуйте на языке Python (с применением пакета scikit-learn) линейный алгоритм метода опорных векторов для данных обучающей выборки своего варианта. Вычислите количество и долю неверных классификаций. Отобразите на плоскости объекты обучающей выборки и разделяющую линию, полученную в результате обучения (точки, изображающие объекты разных классов, должны иметь разные маркеры и цвет).

Содержание отчета

1. Программа, реализующая метод опорных векторов.
2. Графики и результаты работы программы.
3. Выводы по полученным результатам.

Задание 5. Задача классификации (наивный байесовский классификатор).

Реализуйте на языке Python наивный байесовский классификатор на основе данных обучающей выборки своего варианта. Будем считать, что признаки независимы и распределены по гауссовскому закону (нормальной плотности распределения вероятностей). Посчитайте количество и долю неверных классификаций для вашей выборки. Отобразите на плоскости объекты обучающей выборки (точки, изображающие объекты разных классов, должны быть разных цветов).

Содержание отчета

1. Математические расчеты, связанные с построением наивного байесовского классификатора.
2. Программа, реализующая наивный байесовский классификатор.
3. Графики и результаты работы программы.
4. Выводы по полученным результатам.

Задание 6. Исследование работы L2-регуляризатора в задачах регрессии.

Дана функция $y(x)$ (по вариантам).

1. Аппроксимируйте функцию $y(x)$ с помощью линейной модели

$$a(x) = \omega_0 + \sum_{i=1}^{13} \omega_i x^i,$$

т. е. полиномом 13-й степени. Весовые коэффициенты ω_i (i = 0, 1, 2, …, 13) найдите с помощью градиентного алгоритма по обучающему набору данных своего варианта.

2. Обучающую выборку составьте из всех четных индексов сгенерированных значений функции:

$$X^l : \left\{ \left(x_{2i}, y = f\left(x_{2i}\right) \right)_i^{\frac{l}{2}} \right\}.$$

То есть сначала формируется первое значение x_0 с целевым значением $y_0 = f(x_0)$, затем второе $(x_2, y_2 = f(x_2))$ и так до конца диапазона.

3. Вычислите значения весовых коэффициентов ω_i (i = 0, 1, 2, …, 13) для квадратичной функции потерь (в задачах регрессии обычно используют именно такую функцию потерь), минимизирующие эмпирический риск:

$$Q(X^l) = \frac{1}{2} \sum_{i=1}^{l} \left(y_i - a\left(x_i\right) \right)^2 \to \min_{\omega}.$$

Весовые коэффициенты вычисляются по формуле

$$\omega_* = \left(X^T \cdot X \right)^{-1} \cdot X^T \cdot Y,$$

где X – входные векторы обучающей выборки; Y – вектор (или матрица) целевых значений обучающей выборки:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_l \end{bmatrix} = \begin{bmatrix} x_{11}\ x_{12} \dots x_{1n} \\ x_{21}\ x_{22} \dots x_{2n} \\ \dots \quad \dots \quad \dots \dots \\ x_{l1}\ x_{l2} \dots x_{ln} \end{bmatrix}, \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_l \end{bmatrix}.$$

4. Вычислите прогнозы функции с помощью полученной модели $a(x)$ для всего диапазона значений. В отсчетах, не участвующих в выборке, значения модели должны сильно расходиться с целевыми.

5. Вычислите коэффициенты вектора ω с L2-регуляризатором по формуле

$$\omega_* = \left(X^T \cdot X + \lambda \cdot I \right)^{-1} \cdot X^T \cdot Y ,$$

где $\lambda > 0$ – коэффициент регуляризации; $I_{n \times n}$ – единичная матрица.

6. Для новой модели $a(x)$ повторите вычисление прогнозов функции для всего диапазона значений.

Все программы реализуйте на языке Python с использованием пакетов NumPy и Matplotlib. Функция $y(x)$ представлена в табл. 12.

Табл. 12. Функции для исследования L1- и L2-регуляризаторов

Вариант	Функция для исследования L1- и L2-регуляризаторов
1	$y(x) = \frac{1}{10 + x^3}, \quad x \in [0; 10; 0{,}1]$
2	$y(x) = -x^4 + 100x^2 + x, \quad x \in [0; 10; 0{,}1]$
3	$y(x) = x^3 - 10x^2 + x, \quad x \in [0; 10; 0{,}1]$
4	$y(x) = 0{,}1x^5 - 100x^3 + 700x^2, \quad x \in [0; 10; 0{,}1]$
5	$y(x) = -0{,}1x^5 + 5x^3 - 700x^2, \quad x \in [0; 10; 0{,}1]$
6	$y(x) = \frac{1}{10 + x^2}, \quad x \in [0; 10; 0{,}1]$
7	$y(x) = x^4 - 10x^3 + 20x^2 - 100x, \quad x \in [0; 10; 0{,}1]$
8	$y(x) = -0{,}01x^6 - 2x^5 + 200x^3, \quad x \in [0; 10; 0{,}1]$
9	$y(x) = -0{,}01x^6 + 4x^4 - 200x^2, \quad x \in [0; 10; 0{,}1]$
10	$y(x) = x^3 - 5x^2 - 100x + 200, \quad x \in [0; 10; 0{,}1]$

Содержание отчета

1. Математические выкладки для реализации алгоритмов.
2. Тексты программ с результатами их работы.
3. Выводы по полученным результатам.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Джулли, А. Библиотека Keras – инструмент глубокого обучения. Реализация нейронных сетей с помощью библиотек Theano и TensorFlow / А. Джулли, С. Пал. – М. : ДМК-Пресс, 2018. – 294 с.
2. Горелик, А. Л. Методы распознавания : учеб. пособие / А. Л. Горелик, В. А. Скрипкин. – 4-е изд., испр. – М. : Высш. шк., 2004. – 261 с.
3. Лутц, М. Программирование на Python / М. Лутц ; пер. с англ. – 4-е изд. – СПб. : Символ, 2013. – 992 с.
4. Вьюгин, В. Математические основы машинного обучения и прогнозирования / В. Вьюгин. – М. : МЦНМО, 2014. – 218 с.
5. Флах, П. Машинное обучение. Наука и искусство построения алгоритмов, которые извлекают знания из данных / П. Флах. – М. : ДМК-Пресс, 2012. – 412 с.
6. Силен, Д. Основы Data Science и Big Data. Python и наука о данных / Д. Силен, А. Мейсман, М. Али. – СПб. : Питер, 2017. – 336 с.
7. Гудфеллоу, Я. Глубокое обучение / Я. Гудфеллоу, И. Бенджио, А. Курвилль. – 2-е изд., испр. – М. : ДМК-Пресс, 2018. – 652 с.
8. Флах, П. Машинное обучение / П. Флах ; пер. с англ. – М. : ДМК-Пресс, 2015. – 400 с.
9. Вентцель, Е. С. Исследование операций: задачи, принципы, методология : учеб. пособие / Е. С. Вентцель. – М. : Высш. шк., 2007. – 208 с.
10. Паттерсон, Дж. Глубокое обучение с точки зрения практика / Дж. Паттерсон, А. Гибсон. – М. : ДМК-Пресс, 2018. – 418 с.
11. PRIP 2011. Распознавание образов и обработка информации = Pattern Recognition and Information Processing : материалы XI Междунар. конф., Минск, 18–20 мая 2011 г. – Минск : БГУИР, 2011. – 472 с.

Св. план 2024, поз. 32

Учебное издание

Князюк Наталья Владимировна
Рыкова Ольга Васильевна

МЕТОДЫ МАШИННОГО ОБУЧЕНИЯ

ПОСОБИЕ

Редактор *А. Ю. Шурко*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *В. А. Долгая*

Подписано в печать 28.11.2025. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 5,93. Уч.-изд. л. 6,4. Тираж 100 экз. Заказ 4.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий № 1/238 от 24.03.2014,
№ 2/113 от 07.04.2014, № 3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск