МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НАУЧНО-ИССЛЕДОВАТЕЛЬСКИЙ ИНСТИТУТ «МОСКОВСКИЙ ЭНЕРГЕТИЧЕСКИЙ ИНСТИТУТ»

Кафедра МКМ

КУРСОВАЯ РАБОТА

по дисциплине: «Численные методы»

на тему:

«Анализ моделей регрессии для прогнозирования»

Выполнили:

студенты 3 курса группы А-05-22

Антонова Анна

Гусак Анастасия

Преподаватель:

доцент кафедры ММ

Амосова Ольга Алексеевна

Москва 2024

Содержание

Введение

Представим, что имеется множество объектов X. Необходимо каждому объекту сопоставить какое-то значение. Числа, которым сопоставляются объекты из нашего множества называют целевой переменной.

Например, узнать, стоит ли одобрять кредит клиенту банка 40 лет, имеющего 2 детей, который не платит налоги. В данном случае, множество объектов, на основании которых предсказывается значение – и есть множество объектов X. В свою очередь то, насколько клиент банка будет платежеспособен по кредиту – есть предсказываемое целевое значение. Это пример задачи классификации.

Представим другую ситуацию: есть данные о работе нефтяной скважины, по которым необходимо оценить перспективы дальнейшей добычи. В данном случае по набору данных модель будет, к примеру, оценивать потенциальный годовой объём добываемой нефти. Это пример задачи регрессии.

Таким образом, задачи классификации и регрессии можно сформулировать как поиск отображения из множества объектов X в множество возможных целевых значений.

Математически задачи можно описать так:

- классификация: $X \to \{0, 1, \ldots, K\}$ где 0, …, K – номера классов,
- регрессия: $X \to R$, R – действительное число

В данной курсовой работе исследуются методы регрессионного анализа и их реализация. Основное внимание уделяется анализу линейных, полиномиальных и логистических моделей регрессии, а также численным методам, используемым для оптимизации их параметров. В работе будет изучена применимость каждой модели в различных условиях, рассмотрены их ограничения и слабые стороны. В рамках исследования будет разработано и реализовано несколько моделей с использованием как встроенных библиотек, так и самостоятельных решений, что позволит на практике изучить регрессионные методы.

Градиентный спуск

Градиентный спуск — численный метод нахождения *локального* минимума или максимума функции посредством движения вдоль градиента, один из основных численных методов оптимизации.

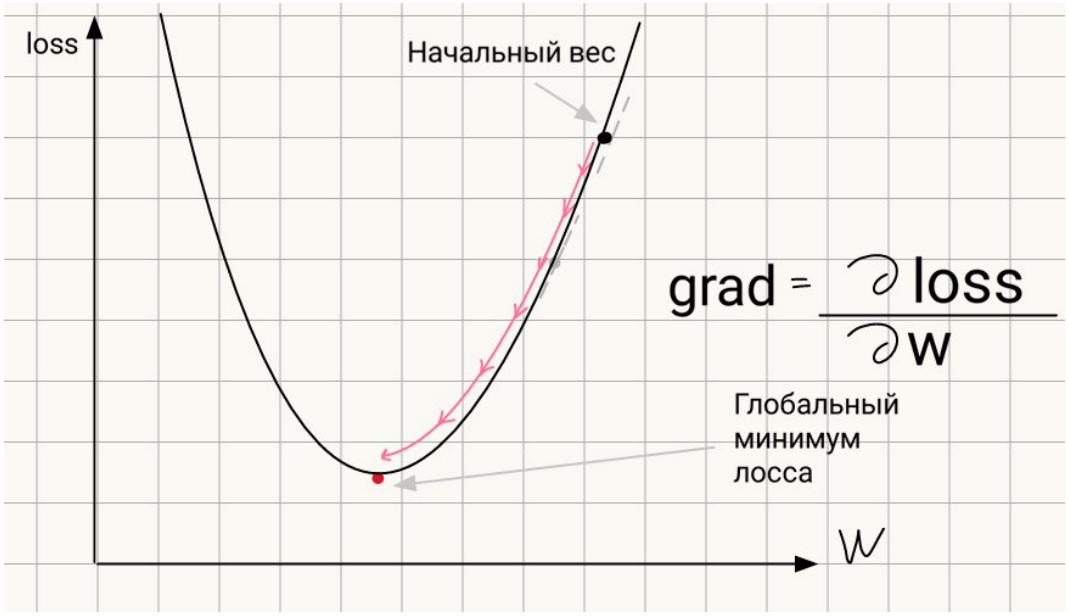
Градиент – это вектор, который указывает направление наискорейшего возрастания функции. Он вычисляется как частные производные функции потерь по каждому параметру:

$$grad\ loss = \nabla loss\,(x) = \frac{\partial loss}{\partial w_1}, \ldots, \frac{\partial loss}{\partial w_n}$$

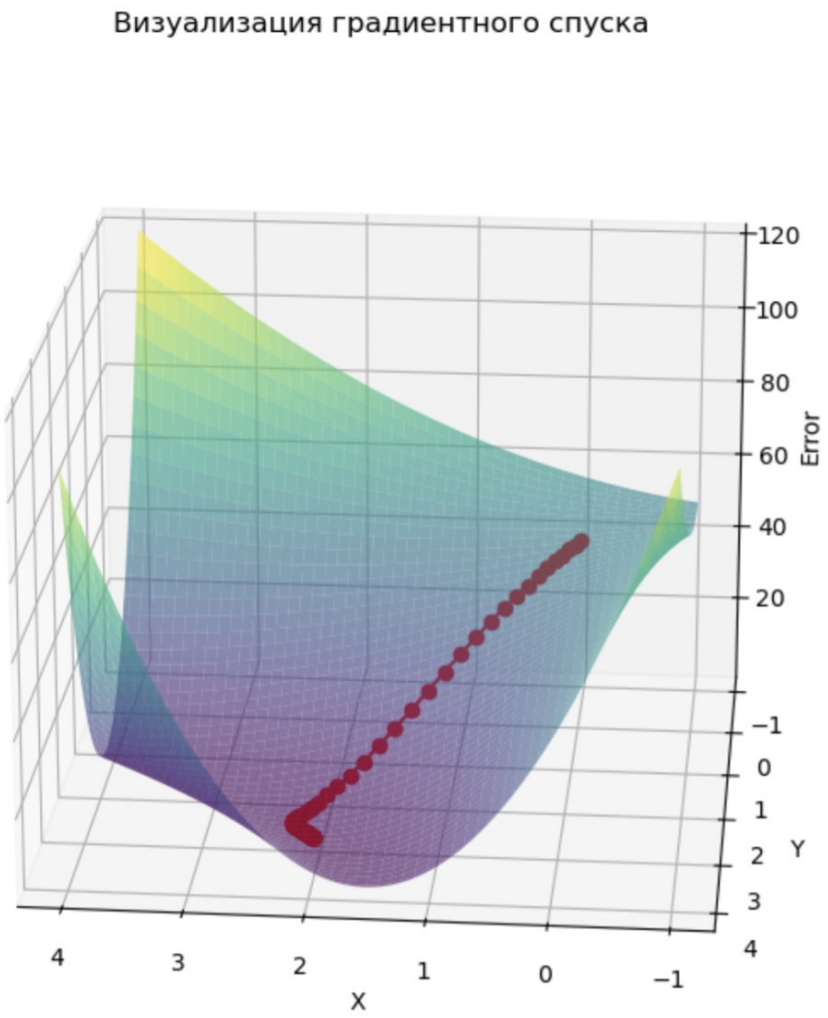
Градиентный спуск активно применяется в машинном обучении для минимизации функции ошибки.

$$loss\,(x) \rightarrow \min_{x}$$

По причине движения градиента в сторону возрастания, нетрудно заметить, что при минимизации функции ошибки необходимо двигаться в сторону антиградиента – направления, противоположного градиенту.



Визуализируем работу градиентного спуска с случае пространств с большей размерностью:



Алгоритм градиентного спуска:

Заранее зададим некоторое число α, которое будет влиять на то, какие большие шаги градиентного спуска совершаются. Оно называется learning rate.

Также необходимо задать значение tolerance, которая будет считать разницу между весами по модулю:

$tolerance = w_{i+1} - w_1$

1) Выбираем точку, с которой начинаем оптимизацию
2) На каждом шаге будем менять все переменные, от которых зависит функция:

$$x_1 = x_1 - \alpha \frac{\partial f}{\partial x_1}, \ldots, x_n = x_n - \alpha \frac{\partial f}{\partial x_n}$$

Или в векторной форме:

$$X = X - \alpha \nabla f$$

Минимизация продолжается до того момента, пока изменения между коэффициентами весов не станет достаточно малым или пройдет допустимое количество шагов.

Градиентный спуск для линейной регрессии:

$$\frac{\partial L}{\partial w_1} = \sum_{i=1}^{l} \frac{\partial \left(x_i^T w - y_i \right)^2}{\partial w_1} = \sum_{i=1}^{l} 2 \left(x_i^T w - y_i \right) \frac{\partial x_i^T w}{\partial w_1} = \sum_{i=1}^{l} 2 \left(x_i^T w - y_i \right) x_{i1}$$

Градиентный спуск для логистической регрессии:

$$\frac{\partial L}{\partial w_1} = \sum_{i=1}^{l} \frac{y_i}{\sigma \left(x_i \right)} \frac{\partial \sigma \left(\mathrm{x_i} \right)}{\partial \mathrm{w_i}} - \frac{1 - y_i}{1 - \sigma \left(x_i \right)} \frac{\partial \sigma \left(x_i \right)}{\partial w_i}$$

$$\frac{\partial \sigma \left(x_i \right)}{\partial w_1} = \sigma \left(x_i \right) \left(1 - \sigma \left(x_i \right) \right) x_{ij}$$

Тогда

$$\frac{\partial L}{w_1} = \sum_{i=1}^{l} x_{ij} \left[y_i \left(1 - \sigma \left(x_i \right) \right) - \left(1 - y_i \right) \sigma \left(x_i \right) \right]$$

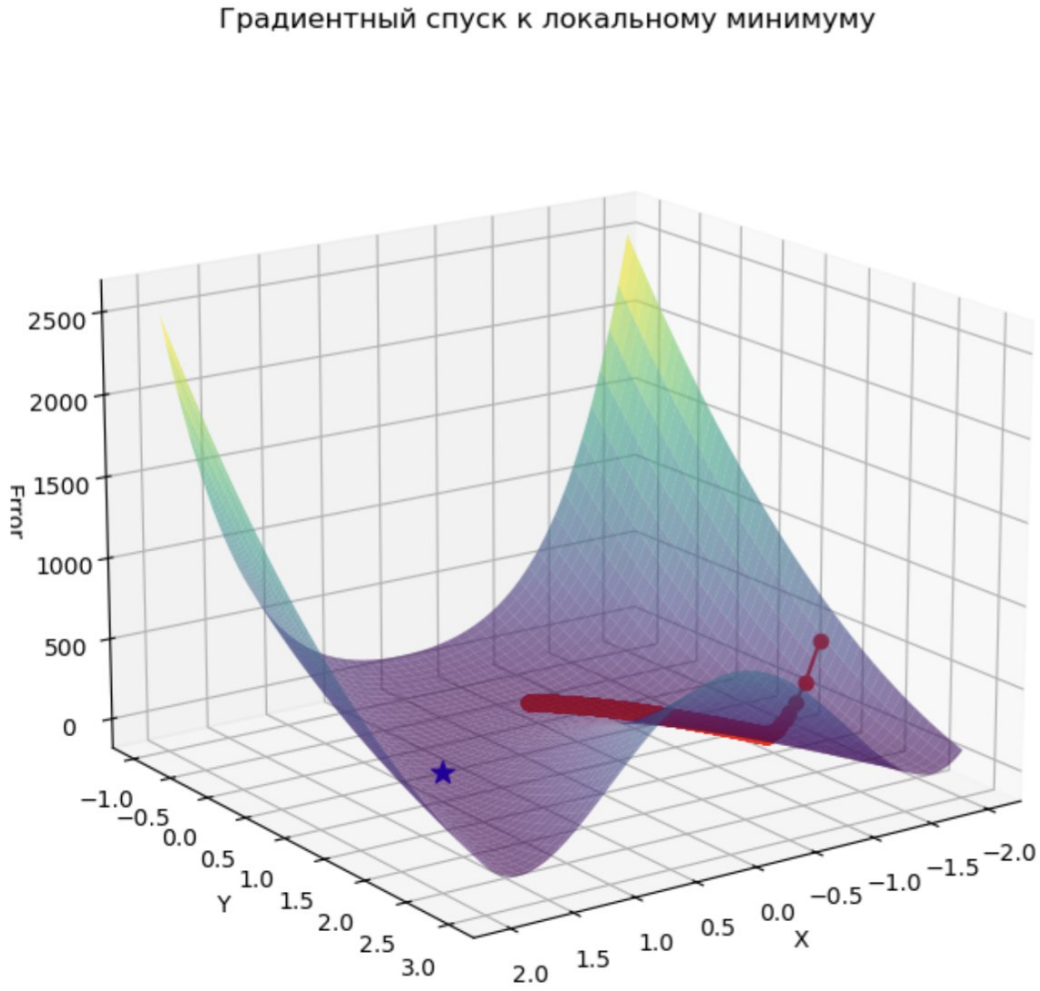
Замечание:

Важно учесть, что при минимизации с помощью градиентного спуска движение происходит в сторону локального минимума. Информативной иллюстрацией особенности работы градиентного спуска может стать функция, имеющая несколько локальных минимумов. В данном примере была использована функция Розенброка, которая часто используется для тестирования алгоритмов оптимизации.

```
def rosenbrock(x, y):
    return (1 - x)**2 + 100 * (y - x**2)**2

def gradient_rosenbrock(x, y):
    dx = -2 * (1 - x) - 400 * x * (y - x**2)
    dy = 200 * (y - x**2)
    return np.array([dx, dy])
```

```
# Создание сетки для визуализации функции Розенброка
x = np.linspace(-2, 2, 100)
y = np.linspace(-1, 3, 100)
X, Y = np.meshgrid(x, y)
Z = rosenbrock(X, Y)

# Создание 3D графика
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
```

```
# Построение поверхности функции Розенброка
ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.6)

# Построение траектории градиентного спуска
ax.plot(trajectory[:, 0], trajectory[:, 1], rosenbrock(trajectory[:, 0],
trajectory[:, 1]), color='r', marker='o')
global_min_x, global_min_y = 1, 1
ax.scatter(global_min_x, global_min_y, rosenbrock(global_min_x,
global_min_y), color='b', marker='*', s=100, label='Глобальный минимум')
# Настройка осей и легенды
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Error')
ax.set_title('Градиентный спуск к локальному минимуму')

# Изменение угла обзора
ax.view_init(elev=17, azim=55)

plt.show()
```



Известно, что глобальный минимум функции Розенброка находится в точке (1,1). Однако заданные начальные условия "пришли" к точке: (-0.34, 0.12).

Линейная регрессия

1. Одномерная линейная регрессия

1.1.Теоретическая справка

Одномерная линейная регрессия представляет из себя простейшую задачу регрессии.

Пусть x – признак, по которому предсказывается целевое значение, а y – искомая целевая переменная.

Для того, чтобы чётко сформулировать задачу, необходимо выразить на математическом языке желание «приблизить» $f_w(x)$ – научиться измерять качество модели и минимизировать её ошибку, меняя обучаемые параметры. В приведенном ниже примере обучаемые параметры — это веса w:

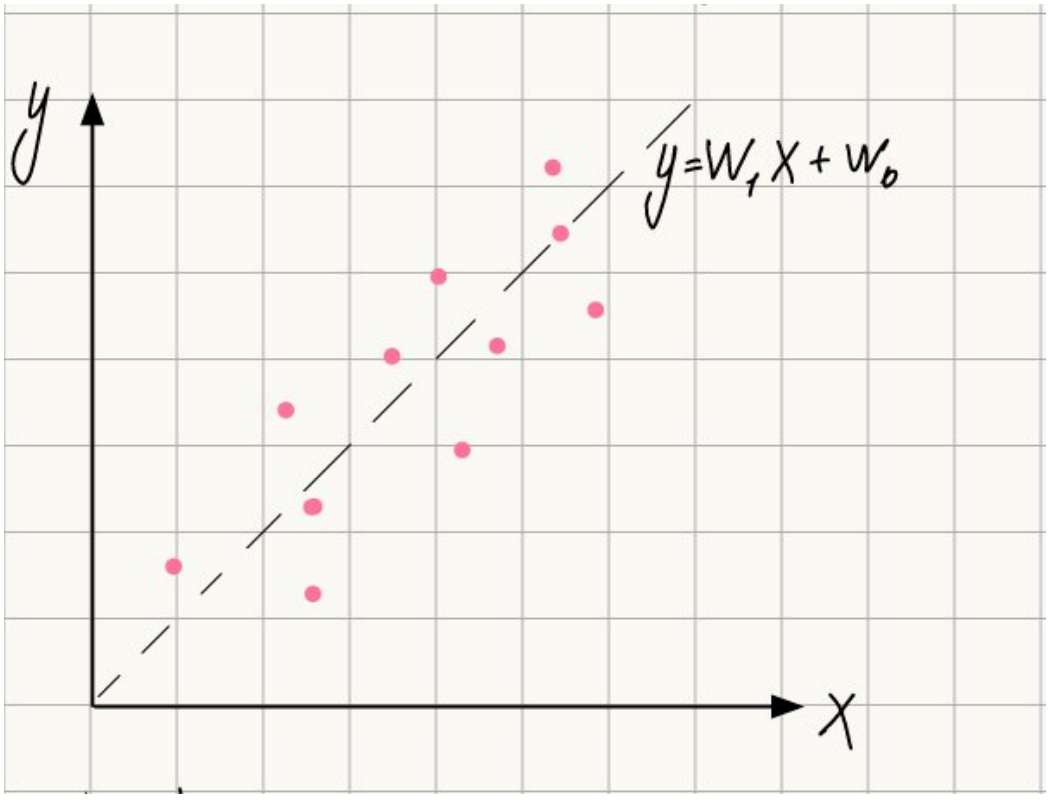
$$y = w_0 + w_1 \cdot x, \text{ где } w_0, w_1 \text{ – коэффициенты регрессии.}$$

Свободный член w_0 часто опускают, потому что такого же результата можно добиться, добавив ко всем x_i признак, тождественно равный единице. Тогда функция будет иметь вид:

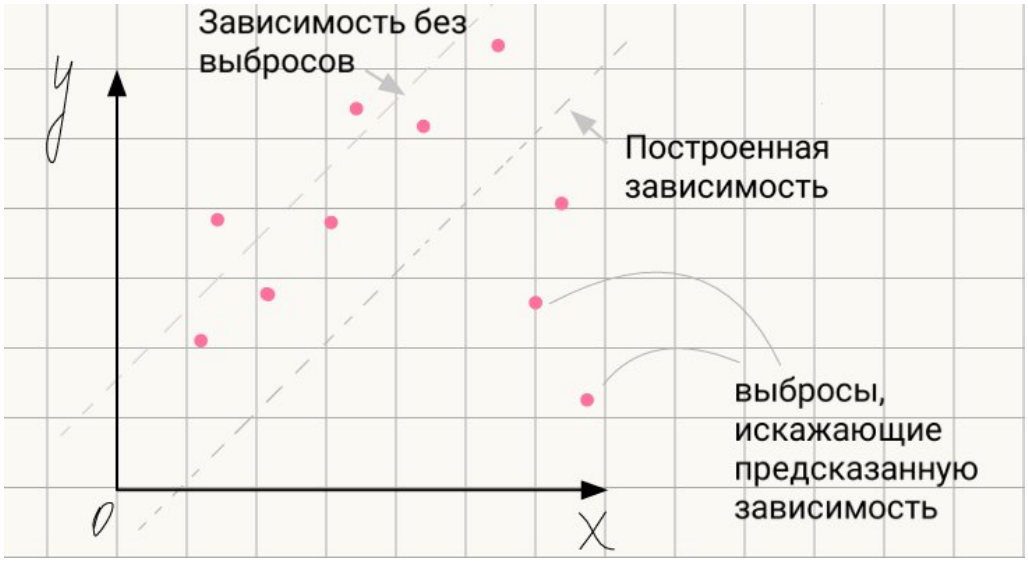
$$w = \langle w_0, w_1 \rangle, \; X = \langle 1, x \rangle$$

$$f_w(x) = \langle w, X \rangle \text{ – скалярное произведение}$$

Визуализация задачи одномерной линейной регрессии:



Нужно добиться, чтобы на парах (x, y) функция f_w как можно лучше приближала нашу зависимость:



Функция, оценивающая то, как часто модель ошибается, традиционно называется функцией потерь, функционалом качества или loss function.

Функции потерь бывают разными. От их выбора зависит то, насколько задачу в дальнейшем легко решать, и то, в каком смысле получится приблизить предсказание модели к целевым значениям. Интуитивно понятно, что для текущей задачи необходимо взять вектор y и вектор предсказаний модели и сравнить, насколько они похожи.

Например, функцией потерь в задаче линейной регрессии может являться $L2$ - норма разницы в квадрате – это квадрат евклидова расстояния $||y - f_w(x)||^2$ между вектором целевых значений и вектором ответов модели, то есть приближение в смысле самого простого и понятного «расстояния»:

$$L(f,X,y) = ||y - f(X)||^2 = \sum_{i=1}^{N} \left(y_i - \langle x_i, w \rangle\right)^2 = ||y - Xw||^2$$

Такой функционал ошибки не очень хорош для сравнения поведения моделей на выборках разного размера. Представим, что необходимо понять, насколько качество модели на тестовой выборке из 2000 объектов хуже, чем на обучающей из 5000 объектов. При измерении $L2$-нормы величина ошибки получается в одном случае – 200, а в другом - 500. Эти числа не очень интерпретируемы. Гораздо практичнее посмотреть на среднеквадратичное отклонение

$$L(f,X,y) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \langle x_i, w\rangle)^2$$

Данная функция ошибки называется Mean Squared Error (MSE) или среднеквадратичное отклонение.

Для каждой конкретной линейной функции, которую задают веса w_i, получается число, которое оценивает, насколько точно эта функция приближает значения к y. Чем меньше это число, тем точнее решение, значит для того, чтобы найти лучшую модель, функцию ошибки надо минимизировать по w:

$$||y - Xw||^2 \to \min_w$$

Минимизация функции потерь выполняется с помощью приближенного численного метода – МНК (метод наименьших квадратов):

$w_{i+1} = w_i - \alpha \cdot \nabla loss = w_i - \alpha \frac{\partial loss}{\partial w_i}$, где α – параметр алгоритма (темп обучение)

Подробнее о градиентном спуске написано в параграфе выше.

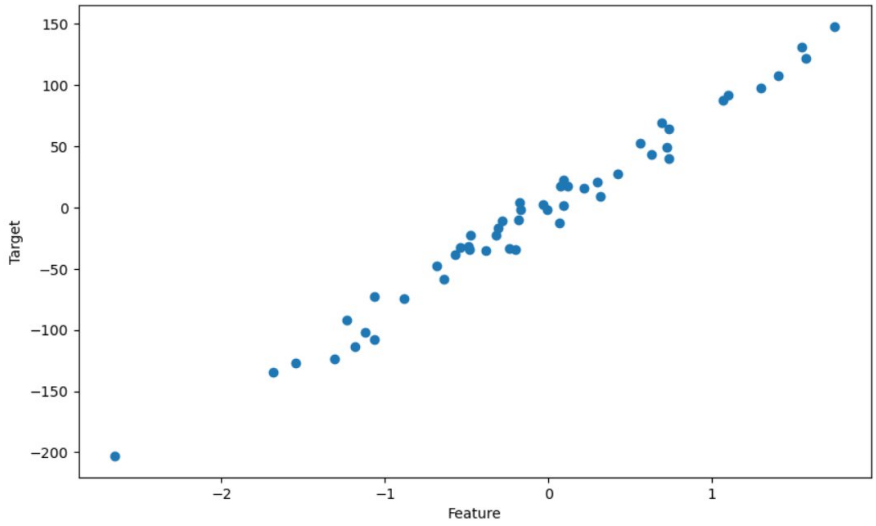
1.2.Реализация одномерной регрессии.

Для начала импортируем все необходимые библиотеки:

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

Создадим с помощью функции из библиотеки sklearn тренировочную выборку с шумом:

```
from sklearn.datasets import make_regression

X, y = make_regression (n_samples = 50, n_features= 1,  n_informative = 1,
noise = 10, random_state = 11)
```



Реализация с помощью библиотеки sklearn

Импортируем линейную регрессию:

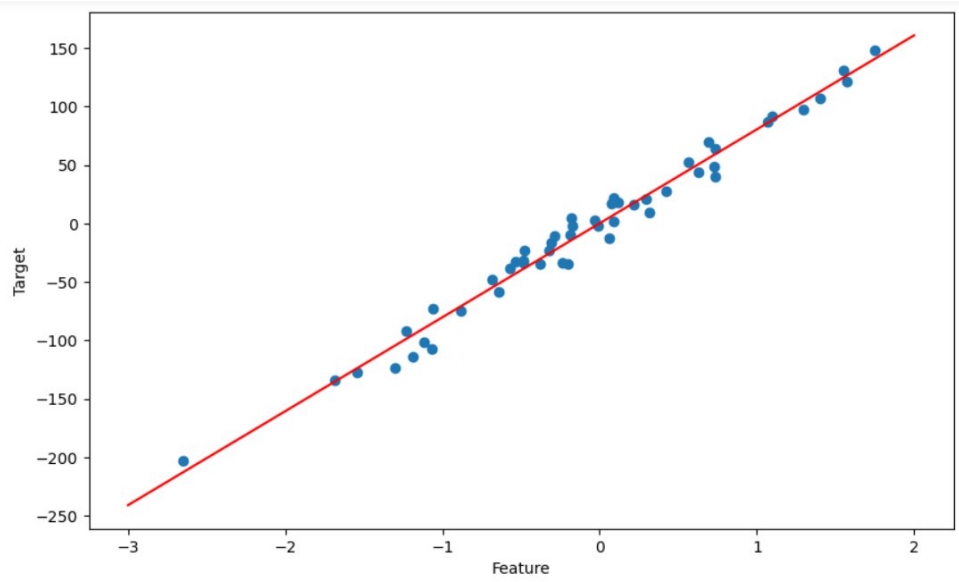
```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model
```

Обучим модель с помощью функции fit на созданных данных и получим коэффициенты w_0,w_1:

```
model.fit(X, y)

print(model.coef_ , model.intercept_)
sklearn_a =  model.coef_[0]
sklearn_b = model.intercept_
```

```
[80.41862354] 0.18171887542100773
```

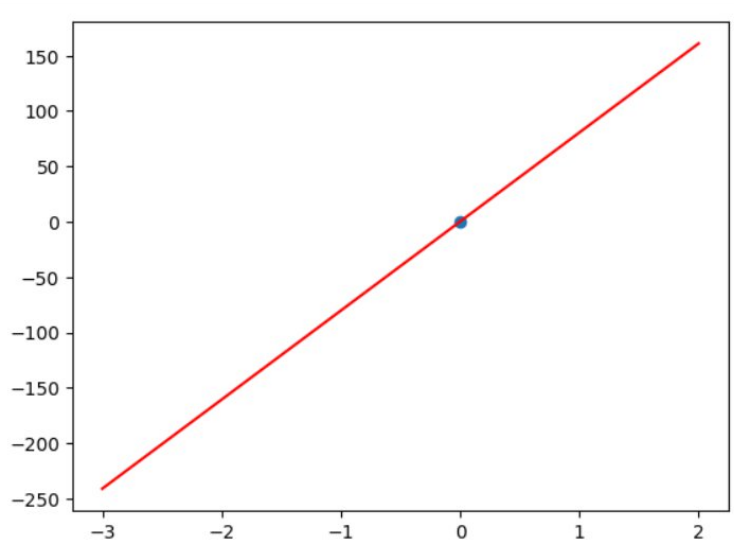
Визуализируем полученный результат:




```
Линейная модель вида: 80.4186235397987x + 0.18171887542100773
```

Пример предсказания:

```
x = 0
y_target = sklearn_a*x+ sklearn_b

plt.scatter(x, y_target)
plt.plot(x_feature, model_y_sklearn, color = 'r')
plt.show()
```



Авторская реализация

```
df = pd.DataFrame({'X': X[:, 0], 'y' : y, 'prediction' : sklearn_a*X[:,0]+
sklearn_b})
df.head()
```

	X	y	prediction
0	0.630080	43.654341	50.851871
1	-1.061634	-72.682350	-85.193462
2	0.296347	21.196446	24.013545
3	1.402771	107.587651	112.990641
4	0.689682	69.620632	55.645021

Запишем функции среднего квадрата ошибки, а также найдем значение градиента по весам w_1 и w_0:

```
def MSE(X, w1, w0, y):
    y_pred = w1*X[:, 0] + w0
    return np.sum( (y-y_pred)**2 / (len(y_pred)+1))

def MSE_grad_w0(X, w1, w0, y):
    y_pred = w1*X[:, 0] + w0
    return 2/(len(X)+1) * np.sum( (y - y_pred ))*(-1) # -1 - производная
по w0

def MSE_grad_w1(X, w1, w0, y):
    y_pred = w1*X[:, 0] + w0
    return 2/(len(X)+1) * np.sum( (y - y_pred)* np.array( [-X[:,0]])) # -X
- производная по w1
```

А теперь реализуем алгоритм минимизации среднеквадратичной ошибки с помощью градиентного спуска:

```
# начальная точка
w1 = 0
w0 = 0

learning_rate = 0.01
next_w1 = w1
```

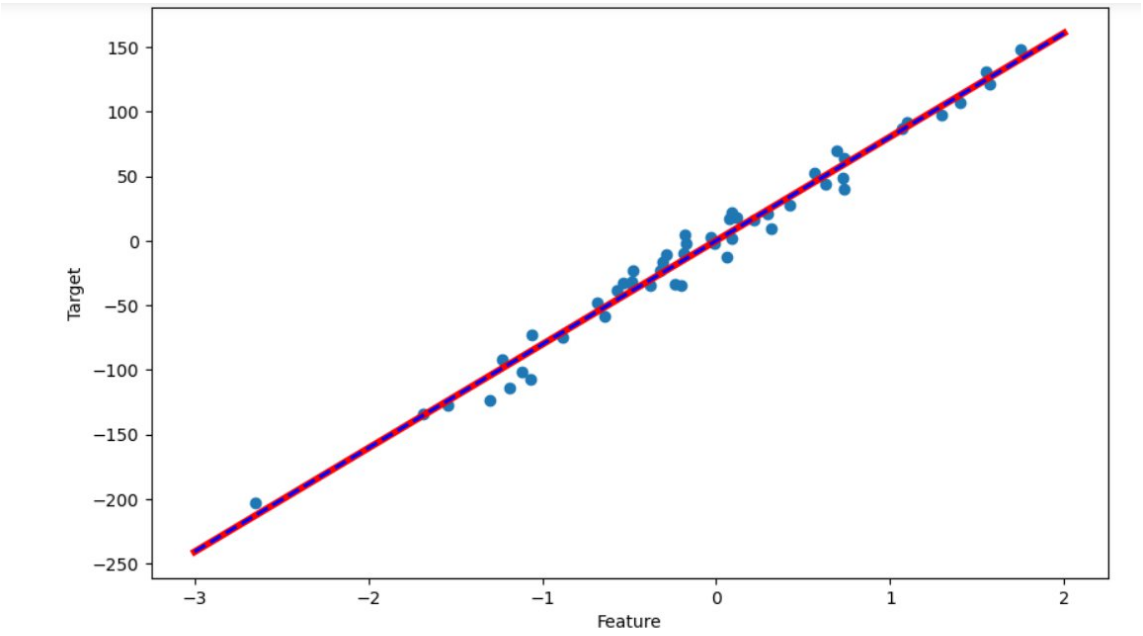
```
next_w0 = w0

max_iterations = 1000
eps = 10**(-6)
it = 1
while (it <= max_iterations):
    cur_w1 = next_w1
    cur_w0 = next_w0

    next_w1 =  cur_w1 - learning_rate * MSE_grad_w1(X, cur_w1, cur_w0, y)
    next_w0 =  cur_w0 - learning_rate * MSE_grad_w0(X, cur_w1, cur_w0, y)

    print(f'Итерация: {it}')
    print(f'Текущая точка: {cur_w1}x + {cur_w0}')
    print(f'Следующая точка: {next_w1}x + {next_w0}')
    print(f'Mean Squared Error: {round(MSE(X, cur_w1, cur_w0, y), 3)}')
    print('-' * 10)
    it+=1
    if abs(cur_w1 - next_w1) <= eps and abs(cur_w0 - next_w0) <= eps:
        break
```

Результат:

```
Итерация: 1
Текущая точка: 0x + 0
Следующая точка: 1.298539813557112x + -0.13648773069168443
Mean Squared Error: 5329.835
----------
Итерация: 2
Текущая точка: 1.298539813557112x + -0.13648773069168443
Следующая точка: 2.575868970426488x + -0.26803779500950453
Mean Squared Error: 5160.763
----------
...
Итерация: 893
Текущая точка: 80.41855826593822x + 0.18169205946114214
Следующая точка: 80.41855927348799x + 0.18169247158837717
Mean Squared Error: 109.736
----------
Итерация: 894
Текущая точка: 80.41855927348799x + 0.18169247158837717
Следующая точка: 80.41856026548236x + 0.18169287738935666
Mean Squared Error: 109.736
----------
```

Сравнение двух моделей

```
x_feature = np.arange(-3, 3)

model_y_our = next_w1*x_feature + next_w0

fig = plt.figure(figsize = (10, 6))
plt.plot(x_feature, model_y_sklearn, linewidth = 4, color = 'r')
plt.plot(x_feature, model_y_our, '--', linewidth = 2, color = 'b')
plt.plot()
plt.xlabel('Feature')
plt.ylabel('Target')
plt.scatter(X, y)
plt.show()
print(f'Линейная модель sklearn вида: {sklearn_a}x + {sklearn_b}')
print(f'Авторская модель вида: {next_w1}x + {next_w0}')
```




```
Линейная модель sklearn вида: 80.4186235397987x + 0.18171887542100773
Наша линейная модель вида: 80.35565499469715x + 0.15865053766558465
```

По графику видно, что построенная модель линейной регрессии почти идентична модели из библиотеки sklearn. Из данного результата можно сделать вывод, что был правильно реализован МНК и алгоритм градиентного спуска.

2. Многомерная линейная регрессия

2.1.Теоретическая справка

Пусть X – матрица признаков, по которому предсказывается целевое значение, а y – искомая целевая переменная. Тогда функция имеет вид:

$y = \langle x_i, w \rangle + w_0$, x_i – столбец всех значений одного признака

Аналогично случаю, рассмотренному в одномерной линейной регрессии, добавим ко всем x_i признак, тождественно равный единице:

$$(x_{i1} \dots x_{iD}) \cdot (w_1 \dots w_D)^T + w_0 = (1, x_{i1}, \dots x_{iD}) \cdot (w_0, w_1, \dots, w_D)^T$$

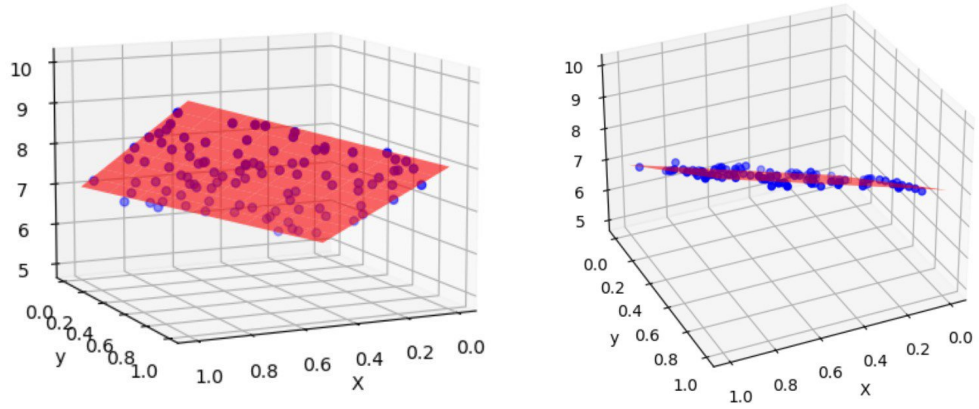
Тогда функция будет иметь вид:

$$f_w(x_i) = \langle w, x_i \rangle$$

Значит, в качестве решения будем искать такой вектор

$$(w_0, w_1, \dots, w_D) \in R^{D+1}$$

Визуализация задачи многомерной линейной регрессии:



Так как $x^{(1)}, \dots, x^{(D)}$ – столбцы матрицы X, то есть столбец признаков, то

$$Xw = w_1 x^{(1)} + \dots + w_D x^{(D)}$$

Теперь задача линейной регрессии формулируется следующим образом: необходимо найти такую линейную комбинацию столбцов $x^{(1)}, \dots, x^{(D)}$, которая наилучшим образом приближает столбец значений целевых переменных $x^{(1)}, \dots, x^{(D)}$ по евклидовой норме, или найти проекцию вектора y на подпространство, образованное векторами $x^{(1)}, \dots, x^{(D)}$.

Разложим $y = y_{\parallel} + y_{\perp}$, где $y_{\parallel} = Xw$ – та самая проекция, а $y_{\perp}$ - ортогональная составляющая, то есть $y_{\perp} = y - Xw \perp x^{(1)}, \dots, x^{(D)}$. Это можно выразить в матричном виде:

$$X^T(y - Xw) = 0$$

В самом деле, каждый элемент столбца $X^T(y - Xw)$ – это скалярное произведение строки X^T (одного из $x^{(i)}$) на $y - Xw$. Из уравнения $X^T(y - Xw) = 0$ можно выразить w:

$$w = \left(X^T X\right)^{-1} X^T y$$

Градиент функции потерь $L(f_w, X, y) = \frac{1}{N}\|Xw - y\|$ будет выглядеть

$$\nabla L = \frac{2}{N} X^T(Xw - y)$$

2.2. Авторская реализация многомерной линейной регрессии

Зададим данные

```
from sklearn.datasets import make_regression

X, y, coefficients = make_regression (n_samples = 50, n_features= 3,
n_informative = 3, noise = 10, coef = True,random_state = 11)
```

Обучим модель многомерной линейной регрессии с помощью библиотеки sklearn

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X, y)

print(model.coef_ , model.intercept_)
sklearn_a =  model.coef_
sklearn_b = model.intercept_
```

```
[16.03610262 26.27272897 95.12674829] -0.470328599082972
```

Запишем функции для вычисления среднеквадратичной функции потерь и её градиента

```
def MSE(X, w, y):
    y_pred = X@w
    return np.sum( (y-y_pred)**2 / (len(y_pred)+1))

def MSE_grad(X, w, y):
    y_pred = X@w
    return 2/(len(X)+1) *(y - y_pred )@(-X)
```

Реализуем алгоритм градиентного спуска для минимизации функции потерь

```
weights = np.zeros((count_weights+1))

learning_rate = 0.01

next_weights = weights

max_iterations = 10000
eps = 10**(-3)
it = 1
while (it <= max_iterations):
    cur_weights = next_weights
```

```
    next_weights =  cur_weights - learning_rate * MSE_grad(X, cur_weights,
y)

    print(f'Итерация: {it}')
    print(f'Mean Squared Error: {round(MSE(X, cur_weights, y), 3)}')
    it+=1
    if np.linalg.norm(cur_weights - next_weights, ord = 2) <= eps:
        break
```

```
Итерация: 1
Mean Squared Error: 6554.436
Итерация: 2
Mean Squared Error: 6379.119
Итерация: 3
Mean Squared Error: 6208.932
…
Итерация: 629
Mean Squared Error: 111.102
Итерация: 630
Mean Squared Error: 111.102
Итерация: 631
Mean Squared Error: 111.102
```

```
print(f'Веса при признаках: {next_weights[:]}')
print(f'Настоящие веса линейной регрессии: {coefficients}')
```

```
Веса при признаках: [16.04201752 26.21559011 95.06406426 -0.43342483]
Настоящие веса линейной регрессии: [17.59034817 29.10379758 96.13877718]
```

Полиномиальная регрессия

1. Теоретическая справка

Полиномиальная регрессия подразумевает нелинейную связь целевой переменной с признаками. Этот метод полезен, когда данные имеют более сложную зависимость. Например, пусть стоит задача прогнозирования скорости автомобиля y (км/ч) от времени разгона t (секунды). Экспериментальные данные покажут, что эта зависимость носит нелинейный характер. Например, при малых значениях t скорость увеличивается быстро, а затем рост замедляется.

Искомая переменная будет иметь вид:

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_n x^n, \text{ где } x \text{ – столбец признаков}$$

Для начала нужно определить степень полинома n. Подбор n зависит от сложности данных и необходимой точности аппроксимации. Чем выше степень полинома, тем лучше модель может подстраиваться под данные, однако возникает риск переобучения.

Затем, нужно преобразовать данные. Для полиномиальной регрессии исходная переменная x расширяется до набора $1, x, x^2, x^3, \ldots, x^n$, образуя полиномиальные признаки.

Тогда функция будет иметь вид:

$$f_w(X) = \mathrm{Xw}, \text{ где } X = \left(1, x, x^2, x^3, \ldots, x^n\right),\ w = \left(w_0, w_1, \ldots w_n\right)$$

Для нахождения коэффициентов будем использовать метод наименьших квадратов, который заключается в минимизации среднеквадратичного отклонения MSE:

$$L\left(y, X, w\right) = \frac{1}{N}\|y - \mathrm{f_w}(\mathrm{X})\|^2 = \frac{1}{N}\|y - \mathrm{Xw}\|^2, \text{ где N – размер данных}$$

Минимизацию будем реализовывать с помощью градиентного спуска, для этого нужно вычислить градиент функции ошибки, он будет иметь вид:

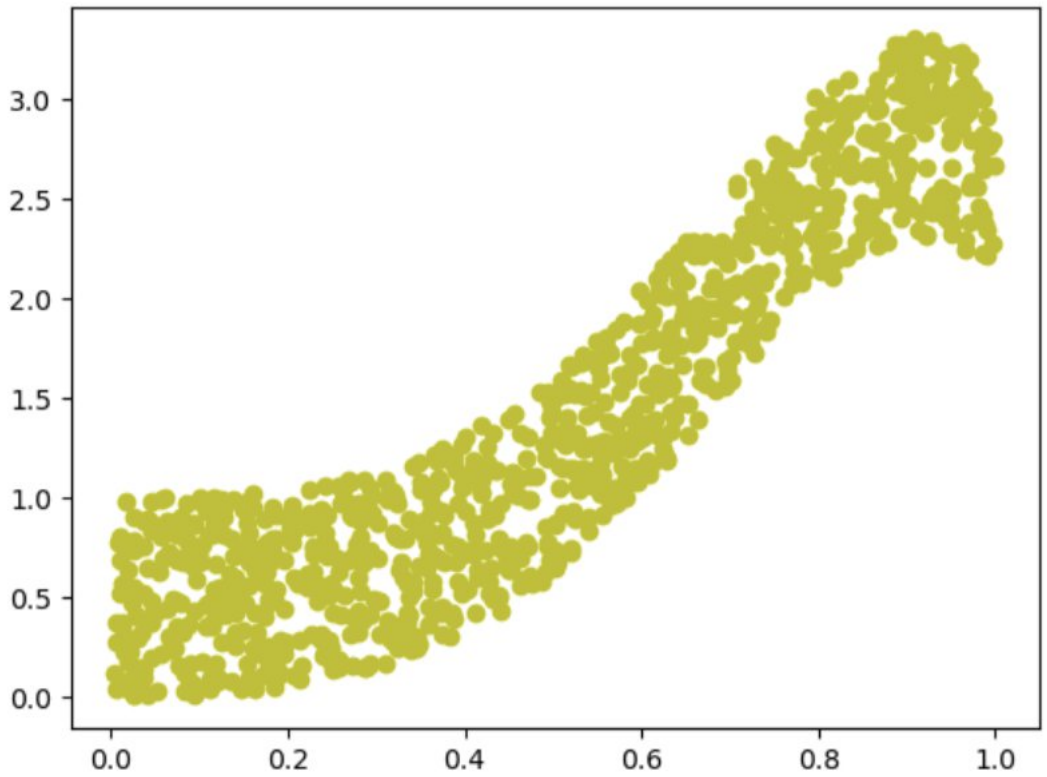
$$\nabla L\left(y, X, w\right) = \frac{-2}{N} \cdot X^T\left(y - Xw\right)$$

2. Авторская реализация и сравнение с моделью sklearn

Импортируем библиотеки и зададим множество точек.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
np.random.seed(42)
X = np.random.rand(1000,1)
y = 5*((X)**(3) * np.sin(np.exp(X))) + np.random.rand(1000,1)
```

```
plt.scatter(X,y, color='yellow')
plt.show()
```



Реализуем функции MSE, её градиента и трансформации признаков к виду полиномиальных признаков.

```
def transform_features(X, degree):
    new_features = []
    for i in range(len(X)):
        temp = []
        for j in range(degree + 1):
            temp.append(X[i][0]**j)
        new_features.append(temp)
    return np.array(new_features)

def MSE(X, w, y):
    y_pred = X @ w
    return np.sum((y - y_pred) ** 2) / len(X)

def MSE_grad_w(X, w, y):
    y_pred = X @ w
    return -2 / len(X) *  X.T @ (y - y_pred)
```

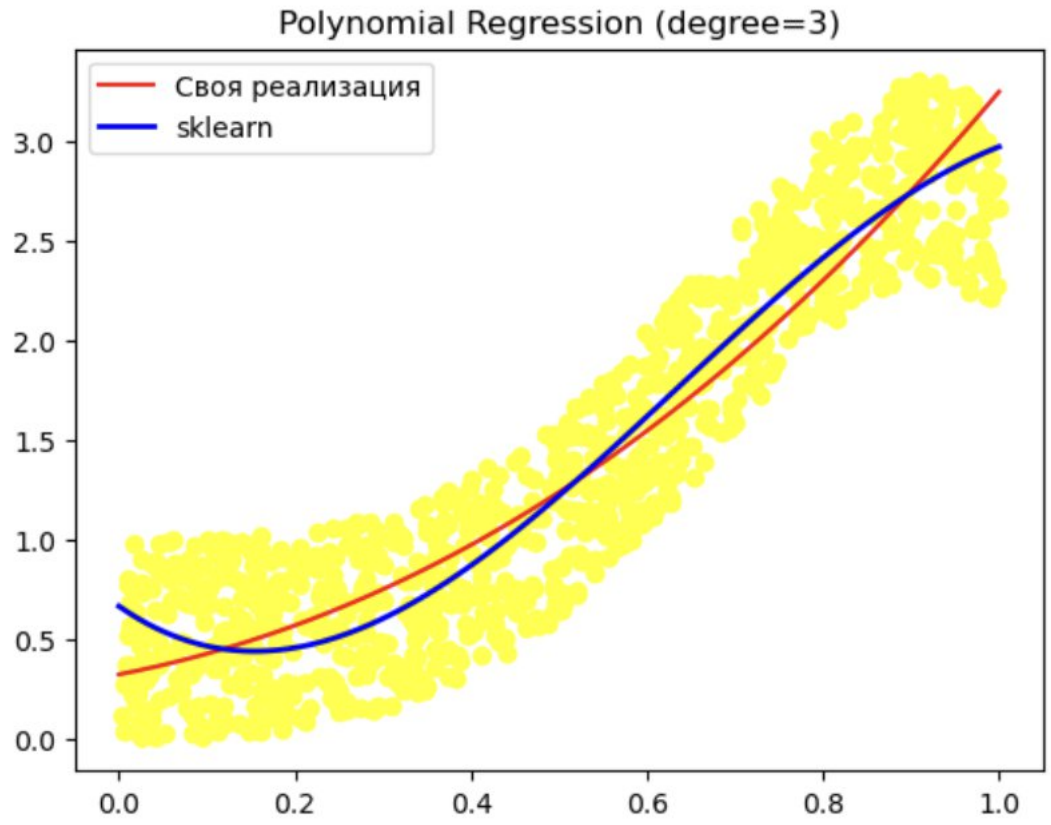
Реализуем функцию градиентного спуска и предсказания целевой переменной

```
def gradient_descent(X, y, degree, learning_rate=0.0001,
max_iterations=1000, eps=1e-6):
    X_poly = transform_features(X, degree)
    w = np.random.randn(X_poly.shape[1], 1)
    for iteration in range(max_iterations):
        gradient = MSE_grad_w(X_poly, w, y)
        new_w = w - learning_rate * gradient
        print(MSE(X_poly, w, y))
        if np.linalg.norm(new_w - w, ord=1) < eps:
            break
```

```
        w = new_w
    return w

def polynomial_predict(w, X, degree):
    X_poly = transform_features(X, degree)
    return X_poly @ w
```

Обучим авторскую модель и модель из sklearn и сравним полученные результаты

```
poly_features = PolynomialFeatures(degree = 3, include_bias=False)
X_poly = poly_features.fit_transform(X)
polynomial_model = LinearRegression()
polynomial_model.fit(X_poly, y)
polynomial_model.intercept_, polynomial_model.coef_

degree = 3

w = gradient_descent(X, y, degree, learning_rate=0.1, max_iterations=5000)

X_new = np.linspace(0, 1, 100).reshape(100, 1)
y_new = polynomial_predict(w, X_new, degree)

plt.scatter(X, y, color='yellow')
plt.plot(X_new, y_new, color='red')
plt.title(f'Polynomial Regression (degree={degree})')

X_new_poly = poly_features.transform(X_new)
y_new_sklearn = polynomial_model.predict(X_new_poly)
plt.plot(X_new, y_new_sklearn, "b-", linewidth=2)
plt.show()
```



Логистическая регрессия

1. Одномерная логистическая регрессия

1.1. Теоретическая справка

Рассмотрим задачу классификации как задачу предсказания вероятности. Для каждого объекта из множества X модель логистической регрессии предсказывает вероятность её принадлежности определённому классу. Рассмотрим пример, когда может пригодиться логистическая регрессия. Допустим, стоит задача кредитного скоринга и нужно по определенным данным о кредиторе понять, сможет ли он выплатить кредит, и на основе этих данных принять решение о выдаче кредита. В этом случае в наборе данных могут встретиться два человека с одинаковыми признаками (возраст, пол, зарплата, сумма кредитования и т.д.), но один из них выплатит кредит, а другой – нет. Такая задача может поставить в тупик модели, которые были рассмотрены ранее, так как на одинаковых наборах признаков, значения целевой переменной будут отличаться.

Возникает проблема, вероятность принимает значения от 0 до 1, а обучить линейную модель с соблюдением таких ограничений достаточно сложно. Но можно научить модель предсказывать какой-то объект, связанный с вероятностью, который будет принимать значения от $(-\infty; +\infty)$, а потом преобразовывать его к вероятности. Этим объектом будет отношения вероятности положительного события к отрицательному – $log\frac{p}{1-p}$

Ответом нашей модели будет $log\frac{p}{1-p}$. Выразим отсюда вероятность:

$$\langle w,x_i\rangle = log\frac{p}{1-p}$$

$$e^{\langle w,x_i\rangle} = \frac{p}{1-p}$$

$$p = \frac{1}{1+e^{-\langle w,x_i\rangle}}$$

Функция в правой части называется сигмоидой

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

Таким образом $p = \sigma\left(\langle w,x_i\rangle\right)$

Необходимо понять, какую функцию потерь нужно использовать, и что она должна минимизировать, потому что как таковая вероятность неизвестна, но зато можно найти другую величину – это правдоподобие. Если вероятность позволяет нам предсказывать неизвестные

результаты, основанные на известных параметрах, то правдоподобие позволяет оценивать неизвестные параметры, основанные на известных результатах. И это как раз то, что нужно. Будем использовать метод максимума правдоподобия для распределения Бернулли (распределение с двумя исходами).

Правдоподобие позволяет понять, насколько вероятно получить данные значения целевого значения y при данных X и весах w. Оно имеет вид:

$$p(y|X,w) = \prod_i p(y_i|x_i,w)$$

И для распределения Бернулли оно будет выглядеть так:

$$p(y|X,w) = \prod_i p_i^{y_i}(1-p_i)^{1-y_i}$$

где p_i – это вероятность, посчитанная из ответов модели. Для удобства перейдём к логарифмическому правдоподобию:

$$l(w,X,y) = \sum_i \left(y_i \log(p_i) + (1-y_i)\log(1-p_i)\right)$$

$$l(w,X,y) = \sum_i \left(y_i \log \sigma(\langle w,x_i\rangle) + (1-y_i)\log(1-\sigma(\langle w,x_i\rangle))\right)$$

Заметим, что

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

Тогда

$$l(w,X,y) = \sum_i \left(y_i \log \sigma(\langle w,x_i\rangle) + (1-y_i)\log(\sigma(-\langle w,x_i\rangle))\right)$$

Результатом будет w, для которого правдоподобие максимально. Чтобы можно было минимизировать функцию потерь, умножим на -1:

$$L(w,X,y) = -\sum_i \left(y_i \log \sigma(\langle w,x_i\rangle) + (1-y_i)\log(\sigma(-\langle w,x_i\rangle))\right)$$

Получили функцию потерь для логистической регрессии.

Для минимизации будем использовать градиентный спуск. Выведем формулу для градиентного спуска.

Воспользуемся свойством сигмоиды:

$$\frac{d\log\sigma(z)}{dz} = \left(\log\left(\frac{1}{1+e^{-z}}\right)\right)' = \frac{e^{-z}}{1+e^{-z}} = \sigma(-z)$$

$$\frac{d\log\sigma(-z)}{dz} = -\sigma(z)$$

Отсюда:

$$\nabla_w \log\sigma\left(\langle w,x_i\rangle\right) = \sigma\left(-\langle w,x_i\rangle\right)x_i$$

$$\nabla_w \log\sigma\left(-\langle w,x_i\rangle\right) = -\sigma\left(\langle w,x_i\rangle\right)x_i$$

Окончательно получим:

$$\nabla_w L(y,X,w) = -\sum_i \left(y_i x_i \sigma\left(-\langle w,x_i\rangle\right) - (1-y_i)\,x_i\sigma\left(\langle w,x_i\rangle\right)\right) =$$

$$= -\sum_i \left(y_i x_i (1 - \sigma\left(\langle w,x_i\rangle\right)) - (1-y_i)\,x_i\sigma\left(\langle w,x_i\rangle\right)\right) =$$

$$= -\sum_i \left(y_i x_i - y_i x_i \sigma\left(\langle w,x_i\rangle\right)) - x_i\sigma\left(\langle w,x_i\rangle\right) + y_i x_i \sigma\left(\langle w,x_i\rangle\right)\right) =$$

$$= -\sum_i \left(y_i x_i - x_i\sigma\left(\langle w,x_i\rangle\right)\right)$$

Предсказание модели будет вычисляться, как было указано выше, следующим образом:

$$p = \sigma\left(\langle w,x_i\rangle\right)$$

Это вероятность положительного класса, от этого значения нужно перейти к предсказанию самого класса. Для этого можно сказать, что, если предсказанная величина больше какого-то порогового значения, то элемент относится к этому классу. Для каждой задачи разумно это пороговое значение подбирать отдельно, минимизируя нужную метрику на отложенной тестовой выборке. Например, сделать так, чтобы доля положительных и отрицательных классов примерно совпадала с реальной.

1.2. Реализация с помощью библиотеки sklearn

Для начала импортируем все необходимые библиотеки:

```
import sklearn.metrics
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
```

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

Создадим с помощью функции из библиотеки sklearn тренировочную выборку с шумом:

```
X, y = make_classification(n_samples=25, n_features=1, n_informative=1,
                           n_redundant=0, random_state=11,
n_clusters_per_class=1, class_sep=0.4)
display(X, y)
```

Нормализуем данные

```
scaler = StandardScaler()
X = scaler.fit_transform(X)
X.mean(axis=0), X.std(axis=0)
```

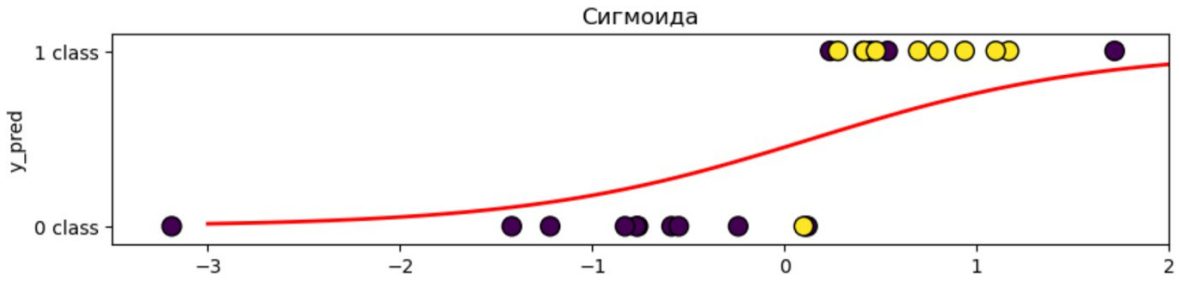
Обучим модель

```
model = LogisticRegression()
model.fit(X, y)

model_a = model.coef_[0][0]
model_b = model.intercept_[0]

print(f"Точность модели: {model.score(X, y)}")

y_pred = model.predict(X)
```

```
Точность модели: 0.8
```

Запишем функция сигмоиды и визуализируем результат

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



1.3. Авторская реализация

Добавим фиктивный признак

```
X = np.c_[np.ones_like(X), X]
```

Напишем реализацию логистической функции потерь, сравним её с реализацией из sklearn. А также напишем функцию, вычисляющую градиент логистической функции потерь.

```
def logloss(y_true, y_pred):
    y_pred = np.clip(y_pred, 1e-16, 1 - 1e-16)
    n = len(y_true)
    logloss_val = 0
    for i in range(n):
        logloss_val += y_true[i] * np.log(y_pred[i]) + (1 - y_true[i]) *
np.log(1 - y_pred[i])
```

```
    return -logloss_val / n
print('Проверка корректности функции logloss')
print(f"Моё значение: {logloss(y, y_pred)}")
print(f"Значение sklearn: {sklearn.metrics.log_loss(y, y_pred)}")
```

```
Проверка корректности функции logloss
Моё значение: 7.35572498739363
Значение sklearn: 7.20873067782343
```

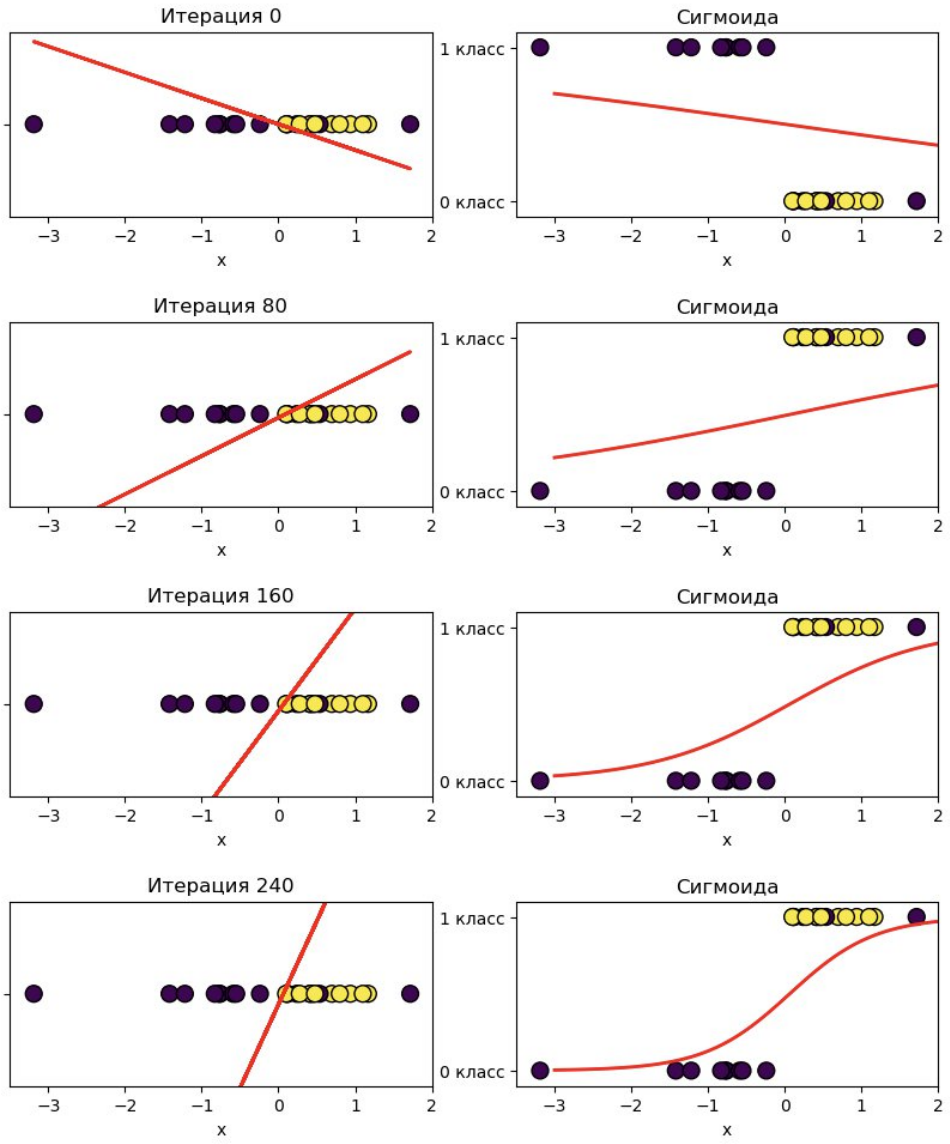
```
def gradient_logloss(y_true, x, w):
    y_pred = sigmoid(x @ w)
    return x.T @ (y_pred - y_true)
```

А теперь реализуем алгоритм минимизации логистической функции потерь с помощью градиентного спуска с выводом истории обучения:

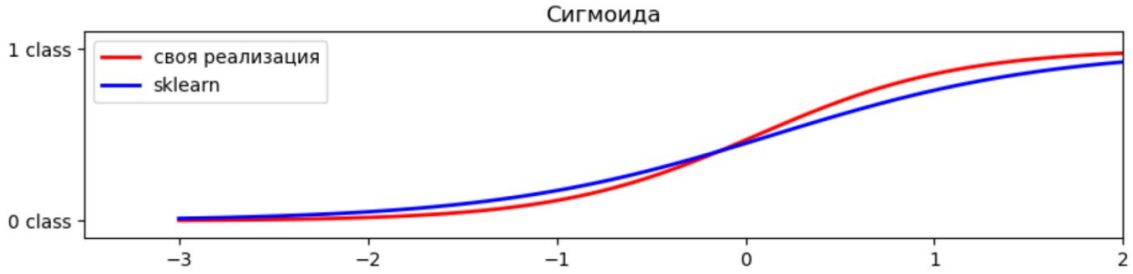
```
# установка минимального значения, на которое должны изменяться веса
eps = 0.0001

# первоначальное точка
np.random.seed(9)
w = np.random.randn(X.shape[1])

# размер шага (learning rate)
learning_rate = 0.001
next_w = w
# количество итерация
n = 250
for i in range(n):
    cur_w = next_w

    # движение в негативную сторону вычисляемого градиента
    next_w = cur_w - learning_rate * gradient_logloss(y, X, w)

    # остановка когда достигнута необходимая степень точности
    if np.linalg.norm(cur_w - next_w) <= eps:
        break

    if i % 80 == 0:
        print(f"Итерация: {i}")
        y_proba = sigmoid(X @ next_w)
        y_class = np.where(y_proba >= 0.5, 1, 0)
        accuracy = (y_class == y).sum() / len(y)
        print(f"Logloss {logloss(y, y_proba)}")
        print(f"Accuracy {accuracy}")
        print('--------------------')

        model_grad = next_w @ X.T
        plt.figure(figsize=(10, 2))

        plt.subplot(121)
        plt.plot(X[:, 1], model_grad, linewidth=2, c='r',
label='gradient')
        plt.scatter(X[:, 1], np.zeros(X.shape[0]), c=y, s=100,
edgecolors='black')
        plt.ylabel(' ')
        plt.xlabel('x')
        plt.yticks(np.arange(0, 1), [''])
        plt.title(f'Итерация {i}')
        plt.ylim(-1, 1)
        plt.xlim(-3.5, 2)

        plt.subplot(122)
```

```
        x_ones = np.c_[np.ones_like(x), x]
        model_pred_x = next_w @ x_ones.T
        plt.plot(x, sigmoid(model_pred_x), linewidth=2, c='r')
        plt.scatter(X[:, 1], sigmoid(model_grad) >= 0.5, c=y, s=100,
edgecolors='black')
        plt.ylabel('')
        plt.xlabel('x')
        plt.yticks(np.arange(0, 2), ['0 класс', '1 класс'])
        plt.ylim(-0.1, 1.1) ;plt.xlim(-3.5, 2)
        plt.title('Сигмоида')

plt.figure(figsize=(10, 2))
x_ones = np.c_[np.ones_like(x), x]
model_pred_x = next_w @ x_ones.T
plt.plot(x, sigmoid(model_pred_x), linewidth=2, c='r', label='своя
реализация')
plt.plot(x, sigmoid(model_y_sk), linewidth=2, c='b', label='sklearn')
plt.legend()
plt.yticks (np.arange(0, 2), ['0 class', '1 class'])
plt.ylim(-0.1, 1.1)
plt.xlim(-3.5, 2)
plt.title('Сигмоида')
```

История обучения:



Сравнение моделей:



2. Многоклассовая логистическая регрессия

2.1. Теоретическая справка

Пусть теперь в исходных данных будет больше признаков, а также данные будут относится к K классам. В логистической регрессии для двух классов была построена линейная модель:

$$b(x) = \langle w,x \rangle + w_0$$

А затем переводили её прогноз в вероятность с помощью сигмоидной функции

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

Теперь для многоклассовой регрессии построим K линейных моделей:

$$b_k = \langle w_k,x \rangle + w_{0k}$$

Каждая из которых даёт оценку принадлежности объекта одному из классов. Воспользуемся оператором $softmax(z_1,\ldots,z_K)$, который нормализует векторы:

$$softmax(z_1,\ldots,z_K) = \left(\frac{e^{z_1}}{\sum_{k=1}^{K} e^{z_k}}, \ldots, \frac{e^{z_K}}{\sum_{k=1}^{K} e^{z_k}}\right)$$

Вероятность k-го класса будет выражаться как:

$$P(y=k|x,w) = \frac{exp(\langle w_k,x \rangle + w_{0k})}{\sum_{j=1}^{K} exp(\langle w_j,x \rangle + w_{0j})}$$

Обучать веса будем также с помощью метода максимального правдоподобия:

$$\sum_{i=1}^{N} log\, P(y=y_i|x_i,w) \to max_{w_1,\ldots w_K}$$

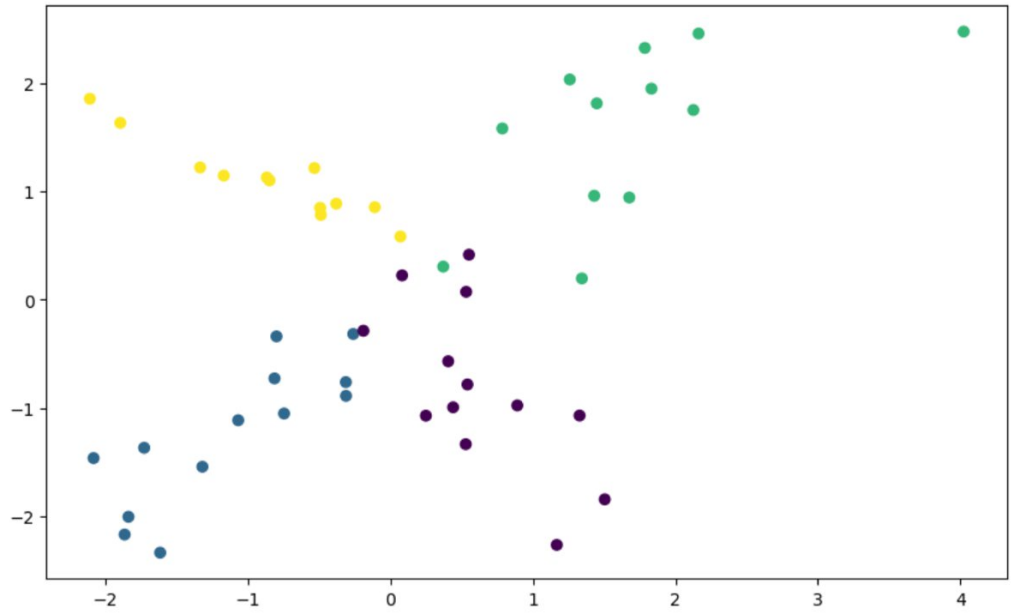
2.2. Авторская реализация и сравнение с моделью sklearn

Импортируем необходимые библиотеки

```
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
import numpy as np
from matplotlib import pyplot as plt
```

Зададим исходные данные

```
X, y = make_classification(n_samples=50, n_features=5, n_classes=4,
n_clusters_per_class=1, random_state=42)
```



Напишем класс многомерной логистической регрессии

```
class MulticlassLogisticRegression:
    def __init__(self, learning_rate=0.01, num_iter=1000):
        self.learning_rate = learning_rate
        self.num_iter = num_iter
        self.weights = None  # Списком весов для каждого класса
        self.classes = None  # Список уникальных классов

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

    def gradient_logloss(self, y_true, x, w):
        y_pred = self.sigmoid(x @ w)
        return x.T @ (y_pred - y_true)

    def _softmax(self, z):
        exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  #
Нормализация для стабильности
        return exp_z / np.sum(exp_z, axis=1, keepdims=True)

    def fit(self, X, y):
        self.classes = np.unique(y)  # Уникальные классы
        self.weights = []  # Инициализация списка весов

        # Добавляем столбец единиц
        X = np.c_[np.ones(len(X)), X]

        for c in self.classes:
            # Создаем метки для бинарной классификации (1 для класса c, 0
для остальных)
            y_binary = (y == c).astype(int)

            # Инициализация весов
            w = np.zeros(X.shape[1])

            # Градиентный спуск
            for _ in range(self.num_iter):
                grad_w = self.gradient_logloss(y_binary, X, w)
                w -= self.learning_rate * grad_w
```

```
            # Сохраняем веса для текущего класса
            self.weights.append(w)

    def predict(self, X):
        # Добавляем столбец единиц
        X = np.c_[np.ones(len(X)), X]
        logits = np.dot(X, np.transpose(self.weights))
        y_pred = self._softmax(logits)
        return np.argmax(y_pred, axis=1)
```

Обучим модель и сделаем предсказание

```
model = MulticlassLogisticRegression(learning_rate=0.1, num_iter=1000)
model.fit(X, y)
y_pred = model.predict(X)
accuracy = np.mean(y_pred == y)
print(f"Точность нашей модели: {accuracy}")
```

```
Точность нашей модели: 0.92
```

Обучим модель из sklearn

```
sklearn_model = LogisticRegression(multi_class='ovr', solver='lbfgs',
max_iter=1000, random_state=42)
sklearn_model.fit(X, y)
y_pred_sklearn = sklearn_model.predict(X)
accuracy_sklearn = accuracy_score(y, y_pred_sklearn)
print(f"Точность модели sklearn: {accuracy_sklearn}")
```

```
Точность модели sklearn: 0.92
```

KNN - регрессия

Теоретическая справка

Основой для построения данного метода стал алгоритм кластеризации KNN, который, в силу рассматриваемой темы курсовой работы, не будет изучаться.

Постановка задачи:

Основная идея метода построения регрессионной модели на основе классификатора заключается в нахождении среднего арифметического значения целевой переменной для каждого кластера, расстояние от объектов которого до рассматриваемого объекта является минимальным:

Алгоритм строится следующим образом:

- сначала вычисляется расстояние между тестовым и всеми обучающими образцами
- далее из них выбирается k-ближайших образцов (соседей), где число k задаётся заранее
- итоговым прогнозом среди выбранных k-ближайших образцов среднее арифметическое
- предыдущие шаги повторяются для всех тестовых образцов

Реализация с помощью библиотеки sklearn

Импорт необходимых библиотек:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
```

Генерация случайных данных для регрессии:

```
X, y = make_regression(n_samples=100, n_features=1, noise=10,
random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

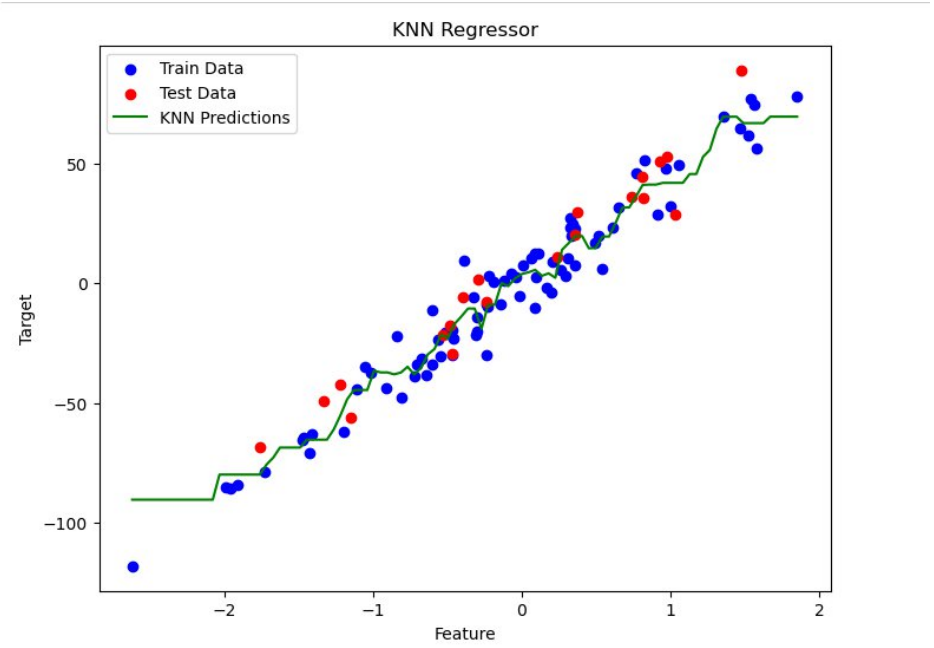
Создание модели KNN-регрессора:

```
model = KNeighborsRegressor(n_neighbors=5)
model.fit(X_train, y_train)
```

Предсказание значений с помощью модели и построение графика:

```
y_pred = model.predict(X_test)

xx = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
yy = model.predict(xx)
```

```
plt.figure(figsize=(8, 6))
plt.scatter(X_train, y_train, color='blue', label='Train Data')
plt.scatter(X_test, y_test, color='red', label='Test Data')
plt.plot(xx, yy, color='green', label='KNN Predictions')
plt.xlabel('Feature')
plt.ylabel('Target')
plt.title('KNN Regressor')
plt.legend()
plt.show()
```



Авторская реализация

Импорт необходимых библиотек:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
```

Функция для вычисления Евклидова расстояния между объектами

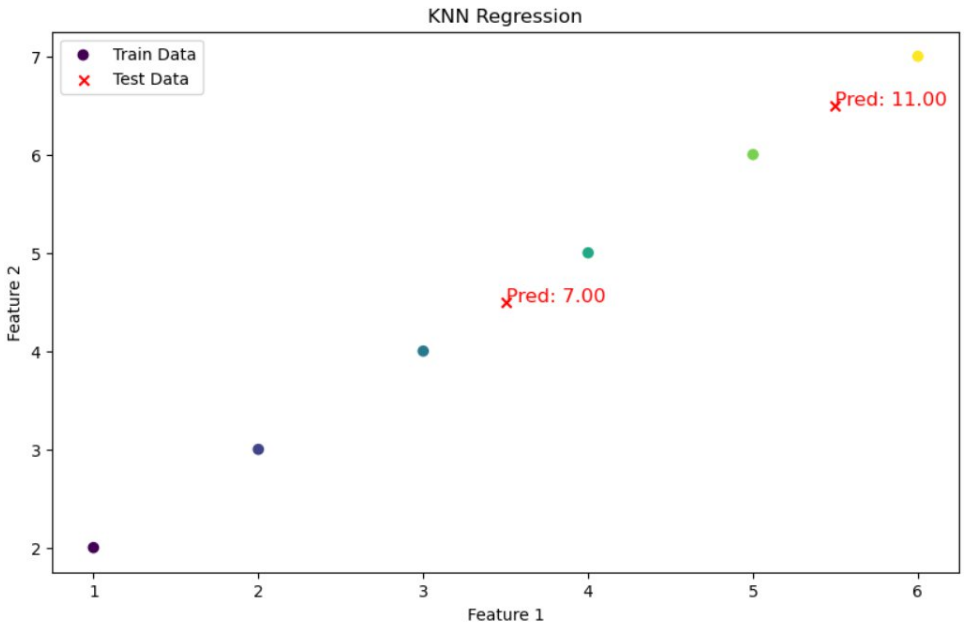
```
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))
```

Функция KNN модели:

```
def knn_predict(X_train, y_train, X_test, k=3):
    predictions = []
    for x in X_test:
        # Вычисляем расстояния до всех точек в тренировочном наборе
        distances = [euclidean_distance(x, x_train) for x_train in
X_train]
        # Получаем индексы k ближайших соседей
        k_indices = np.argsort(distances)[:k]
        # Получаем значения целевой переменной для k ближайших соседей
        k_nearest_labels = [y_train[i] for i in k_indices]
        # Возвращаем среднее значение целевой переменной для k ближайших
соседей
        predictions.append(np.mean(k_nearest_labels))
    return np.array(predictions)
```

Предсказание модели:

```
# Пример данных
X_train = np.array([[1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7]])
y_train = np.array([3, 5, 7, 9, 11, 13])

X_test = np.array([[3.5, 4.5], [5.5, 6.5]])

# Делаем предсказания
predictions = knn_predict(X_train, y_train, X_test, k=3)
print(predictions)

# Построение графиков
plt.figure(figsize=(10, 6))

# График тренировочных данных
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap='viridis',
label='Train Data')

# График тестовых данных
plt.scatter(X_test[:, 0], X_test[:, 1], c='red', marker='x', label='Test
Data')

# График предсказаний
for i, pred in enumerate(predictions):
    plt.text(X_test[i, 0], X_test[i, 1], f'Pred: {pred:.2f}', color='red',
fontsize=12)

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('KNN Regression')
plt.legend()
plt.show()
```



Заключение

Сравнение авторских реализаций с библиотечными помогло лучше понять сильные и слабые стороны каждого подхода.

Например, линейная регрессия, особенно в многомерном варианте, имеет несколько значительных слабостей, которые могут ограничивать её применение и точность предсказаний:

- Предположение о линейности. Линейная регрессия предполагает, что зависимость между независимыми и зависимой переменной является линейной. Это ограничивает её использование в случаях, когда истинные зависимости являются нелинейными. Например, попытка аппроксимировать параболическую зависимость с помощью линейной модели приведет к значительным ошибкам.
- Мультиколлинеарность. Наличие линейной зависимости между независимыми переменными (мультиколлинеарность) может привести к неустойчивости оценок коэффициентов модели. Это затрудняет интерпретацию результатов и может вызвать большие колебания в значениях коэффициентов при изменении выборки. В крайних случаях матрица признаков может стать вырожденной, что делает невозможным вычисление коэффициентов методом наименьших квадратов и градиентным спуском.

Эти слабости делают линейную регрессию менее подходящей для сложных задач анализа данных и предсказания по сравнению с более гибкими методами машинного обучения.

Основные слабости полиномиальной регрессии:

- Переобучение. Полиномиальная регрессия может легко переобучаться, особенно при использовании полиномов высокой степени. Это происходит, когда модель слишком точно подстраивается под обучающие данные, что приводит к плохой обобщающей способности на новых данных. Чем выше степень полинома, тем больше риск переобучения.
- Выбор степени полинома. Определение подходящей степени полинома является критически важным, но часто сложным процессом. Неправильный выбор может привести либо к недообучению (слишком низкая степень), либо к переобучению (слишком высокая степень). Это требует дополнительных методов оценки, таких как кросс-валидация.

- Чувствительность к выбросам. Полиномиальная регрессия очень чувствительна к выбросам в данных. Наличие аномальных значений может значительно исказить результаты модели и привести к неправильным выводам.
- Сложность интерпретации. Интерпретировать коэффициенты полиномиальной регрессии может быть сложнее, чем в линейной модели. Например, влияние независимой переменной на зависимую переменную становится нелинейным и зависит от степени полинома, что затрудняет понимание реальных взаимосвязей.
- Мультиколлинеарность. При использовании полиномов высокой степени могут возникать проблемы мультиколлинеарности между признаками, что может усложнить оценку коэффициентов и снизить их надежность.
- Проблемы с экстраполяцией. Полиномиальная регрессия может давать ненадежные прогнозы за пределами диапазона обучающих данных. Экстраполяция с помощью высоких степеней полиномов может привести к неожиданным и неадекватным результатам.

Эти слабости необходимо учитывать при выборе полиномиальной регрессии как метода анализа данных или предсказания, особенно в контексте сложных и многомерных наборов данных.

Основные слабости логистической регрессии:

- Линейность. Логистическая регрессия предполагает, что классы могут быть разделены линейной границей. Если данные не линейно разделимы, модель может показывать плохие результаты и низкую точность. Это ограничивает её применение в сложных задачах с нелинейными зависимостями.
- Ограниченные возможности расширения. Хотя логистическая регрессия может быть расширена на многоклассовые задачи, это делает модель более сложной и может потребовать дополнительных усилий для настройки и оценки.

Эти слабости делают логистическую регрессию менее подходящей для некоторых типов задач классификации, особенно когда данные имеют сложные структуры или нелинейные зависимости.

Основные слабости KNN регрессора

- Чувствительность к шуму: KNN регрессор подвержен влиянию шумов в данных. Наличие выбросов или аномальных значений может значительно исказить результаты, так как алгоритм основывает свои предсказания на ближайших соседях, которые могут быть ненадежными.

- Проблема выбора параметра k: Оптимальное значение k (число ближайших соседей) критически важно для производительности модели. Слишком малое значение k может привести к переобучению, тогда как слишком большое значение может сгладить важные локальные особенности данных, что снизит точность предсказаний.
- Высокая вычислительная сложность: KNN является методом, который требует вычисления расстояний между тестовой точкой и всеми обучающими примерами. Это делает его медленным при работе с большими объемами данных, особенно если требуется много предсказаний.
- Неэффективность при высоких размерностях.

Эти слабости делают KNN регрессор менее подходящим для некоторых задач, особенно когда данные имеют сложные структуры или высокую размерность.

Таким образом, в данной курсовой работе были исследованы методы регрессионного анализа и их реализация. В работе была изучена применимость каждой модели в различных условиях, а также слабые стороны. В рамках исследования было реализовано несколько моделей с использованием как встроенных библиотек, так и авторских решений, что позволило на практике изучить регрессионные методы.

Список литературы

1. Яндекс Образование. Руководство по машинному обучению [Электронный ресурс]. – Режим доступа: https://education.yandex.ru/handbook/ml, свободный.
2. Stepik. Основы машинного обучения [Электронный ресурс]. – Режим доступа: https://stepik.org/course/68343/syllabus, свободный.
3. Stepik. Введение в машинное обучение [Электронный ресурс]. – Режим доступа: https://stepik.org/course/4852/syllabus?search=6144492006, свободный.
4. Stepik. Основы Python для анализа данных [Электронный ресурс]. – Режим доступа: https://stepik.org/course/3356/syllabus, свободный.
5. Stepik. Машинное обучение: базовый курс [Электронный ресурс]. – Режим доступа: https://stepik.org/course/8057/syllabus?search=6144523137, свободный.
6. VK Образование. Программа по аналитике данных [Электронный ресурс]. – Режим доступа: https://education.vk.company/program/data-analytics, свободный.