

10. Методы регрессионного анализа

Цель занятия

В результате обучения на этой неделе:

- вы узнаете, в каких задачах машинного обучения используются линейные модели;
- сможете формально поставить задачу линейной регрессии;
- познакомитесь с теоремой Гаусса-Маркова;
- научитесь использовать L1 и L2 регуляризации для решения задач машинного обучения;
- разберете построение линейной регрессии с помощью sklearn.

План занятия

1. [Интерполяция данных](#)
2. [Простая линейная регрессия](#)
3. [Линейная регрессия с помощью sklearn](#)
4. [Линейные модели машинного обучения](#)
5. [Линейная регрессия](#)
6. [Теорема Гаусса-Маркова](#)
7. [L1 и L2 регуляризация](#)
8. [Решение линейной регрессии и анализ устойчивости решения](#)

Конспект занятия

1. Интерполяция данных

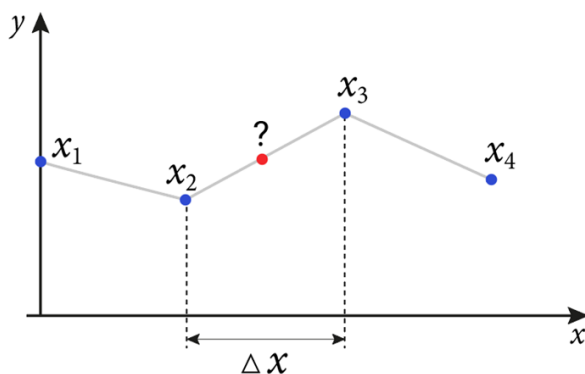
Поговорим о трех разных понятиях, позволяющих исследовать зависимости между дискретными наборами данных, а именно:

- интерполяция;
- аппроксимация;
- регрессия.

Интерполяция

Нередко исходные данные при решении математических задач представлены рядом точек произвольной зависимости, которая может быть как известной, так и неизвестной. Для вычисления промежуточных значений функции используется аппарат **интерполяции**.

Например, наблюдения за изменением температуры воздуха в помещении производились в конце дня. Чтобы оценить значения в утреннее время, потребуется выполнить интерполяцию:



Экстраполяция

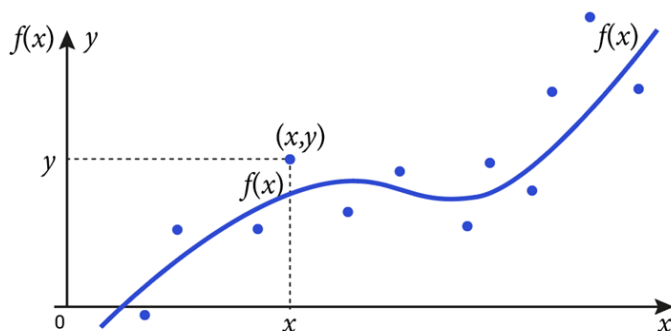
Экстраполяция — предсказание поведения функции вне интервала. Например, предсказание курса доллара на основании предыдущих колебаний.



Аппроксимация

Аппроксимация — метод, состоящий из замены «сложной» функции на более «простую». Аппроксимацию используют, когда нужно получить функцию, похожую на данную, но более удобную для вычислений и манипуляций (дифференцирования, интегрирования и т. п.). Основная цель аппроксимации — сгладить ломаные или другие более сложные кривые.

Кроме того, мы часто имеем дело с очень шумными или извилистыми данными. Для решения этой задачи исследования достаточно знать общую тенденцию. В таких случаях используется аппроксимация.

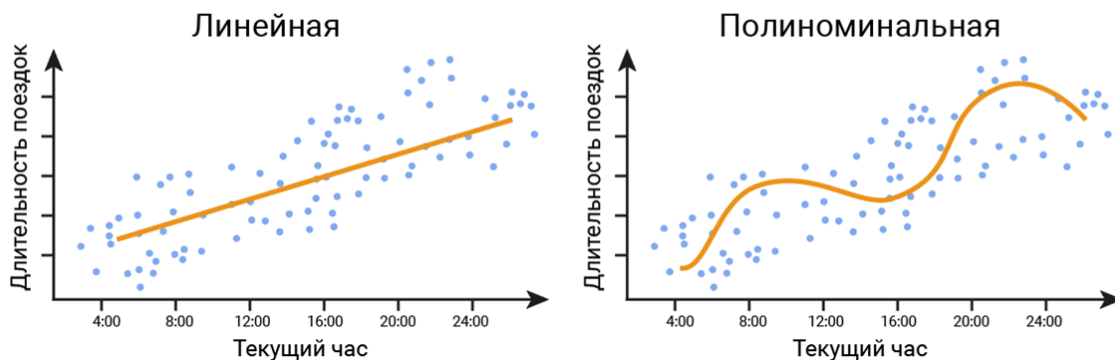


Регрессия

Регрессия представляет собой метод аппроксимации зависимостей между входными и выходными переменными на основе линейной или полиномиальной модели. Как правило, используется в предсказательных целях. Например, предсказание

автомобильной пробки в зависимости от времени суток:

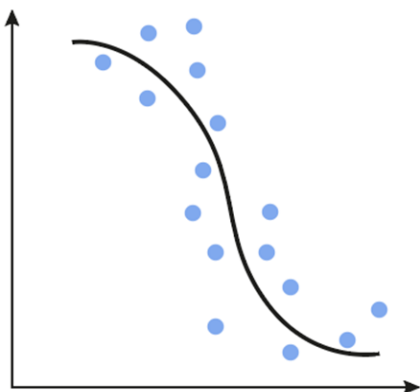
Предсказываем пробки



2. Простая линейная регрессия

Регрессия — это способ аппроксимации взаимосвязи двух дискретных величин. Ее задача — объяснить одним из способов зависимость между некоторыми переменными:

Связана ли переменная X с переменной Y ? Если да, то в чем эта связь заключается? Можем ли мы ее использовать, чтобы предсказать Y ?



Для чего используется регрессия в анализе данных:

- прогноз стоимости ценных бумаг;
- анализ спроса, объема продаж;

- медицинские диагнозы;
- любые зависимости числа от времени.

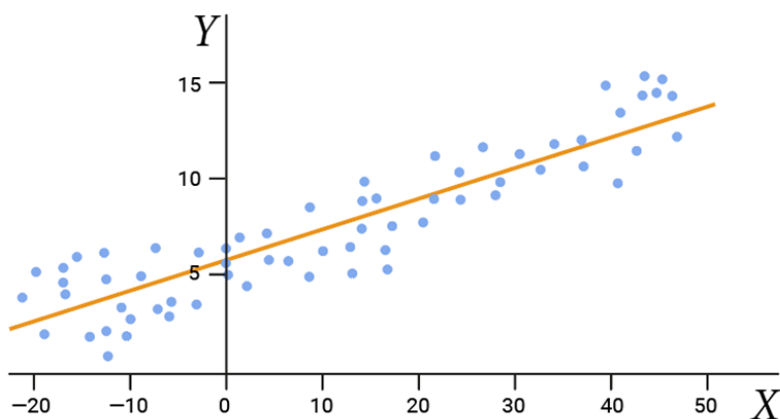
Популярные алгоритмы для регрессионных моделей:

- линейная регрессия;
- нелинейная регрессия.

Выбор алгоритма зависит от задачи и целей исследования.

Простая линейная регрессия

Простая линейная регрессия (парная линейная регрессия) моделирует связь между величиной одной переменной и величиной второй, например, по мере увеличения X увеличивается и Y . Или же по мере уменьшения X уменьшается и Y .



Уравнение линейной регрессии:

$$y = ax + b,$$

где a и b — параметры модели.

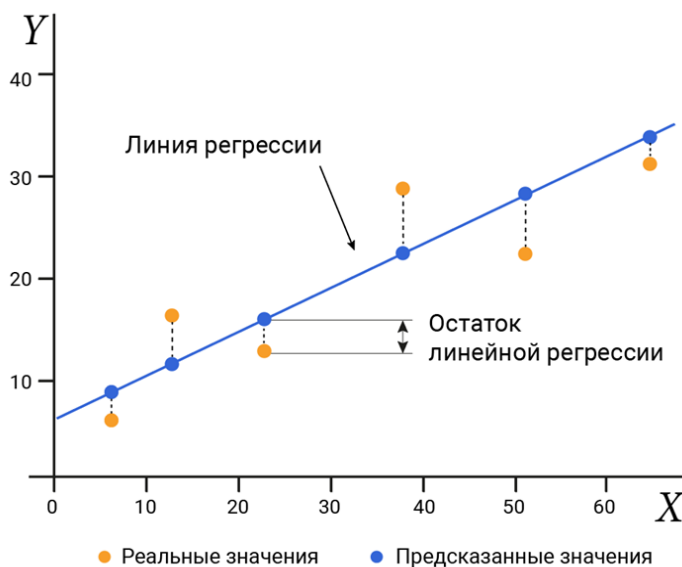
Иными словами, задача линейной регрессии сводится к задаче оптимизации параметров a и b , которые определяют зависимость наилучшим образом, описывающую исследуемые переменные. Как это сделать?

Подгонка параметров

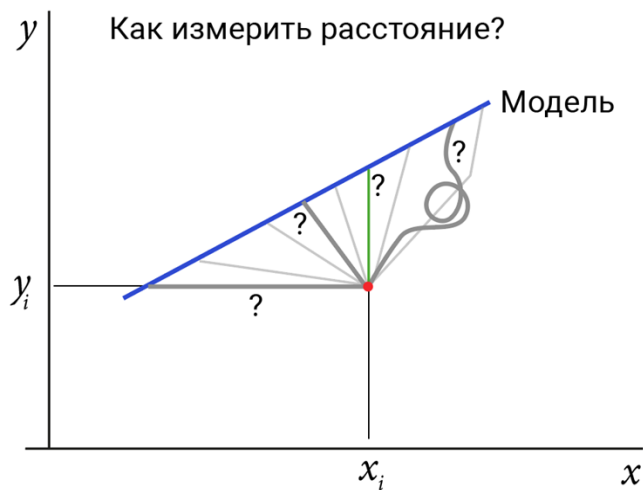
Визуально можно отметить, что чем ближе прямая расположена к наблюдаемым данным, тем лучше она их описывает. Математически мы можем рассчитать

отклонение значений модели от наблюдаемых и выбрать модель с самыми минимальными отклонениями от фактических значений.

Подгонка параметров осуществляется путем минимизации отклонений значения модели от фактических.



Однако возникает вопрос: каким образом рассчитывать эти отклонения? Расстояние от точки до прямой можно считать множеством различных способов:



Абсолютное расстояние

Простейший вариант — сумма модулей отклонений или **Least Absolute Distance (LAD)**:

$$LAD = \sum_i^N |y_i - \hat{y}_i|$$

Эта формула основана на определении абсолютного расстояния между смоделированными и наблюдаемыми значениями.

Другой подход также основан на вычислении абсолютного расстояния. **Mean Absolute Error (MAE)** — среднее модулей отклонений:

$$MAE = \frac{1}{N} \sum_i^N |y_i - \hat{y}_i|$$

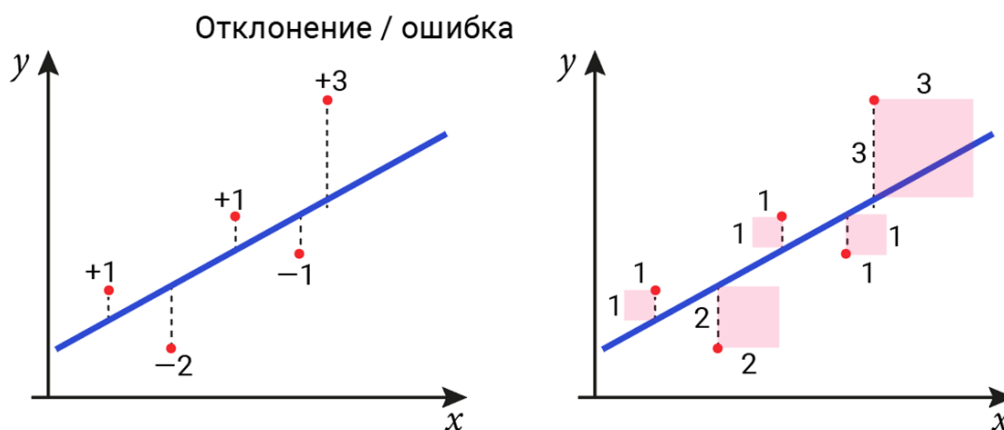
Эти функции называются **функциями потерь**. Чтобы подобрать параметры регрессионной модели, достаточно просто минимизировать одну из этих функций.

Метод наименьших квадратов

Метод наименьших квадратов — наиболее удобный с точки зрения вычислений.

Функция потерь в этом методе **Sum of Squared Errors** (SSE, RSS, SSR) — сумма квадратов отклонений регрессанта от модели:

$$SSE = \sum_i^N (y_i - \hat{y}_i)^2$$



Преимущество такого подхода в том, что производная квадратичной функции является линейной функцией.

Аналогично абсолютному отклонению существует еще одна квадратичная функция потерь. **Mean Squared Error** (MSE, MSD) — среднеквадратичная ошибка:

$$MSE = \frac{1}{N} \sum_i^N (y_i - \widehat{y}_i)^2$$

Оценка качества модели

Для оценки, насколько хорошо выбранная модель соответствует наблюдаемому набору данных, вполне разумно использовать среднеквадратичную оценку. Однако стандартной принятой мерой была выбрана **Root Mean Square Error** (RMSE, RMSD) — квадратный корень из среднеквадратичной ошибки:

$$RMSE = \sqrt{\frac{1}{N} \sum_i^N (y_i - \widehat{y}_i)^2}$$

При оценке того, насколько хорошо модель соответствует набору данных, чаще используется RMSE, потому что она измеряется в тех же единицах, что и целевая переменная.

Еще одна важная оценка качества модели регрессии — **коэффициент детерминации** (coefficient of determination R^2) — доля дисперсии, объясняемая моделью, со значениями в интервале от 0 до 1, измеряет долю вариации данных:

$$R^2 = 1 - \frac{SSE}{SST},$$

$$\text{где } SSE = \sum_i^N (y_i - \widehat{y}_i)^2,$$

$$SST = \sum_i^N (y_i - \overline{y})^2$$

Коэффициент детерминации полезен в интерпретации результатов регрессии, где определяется, насколько модель подогнана к данным.

Стоит обратить внимание, что эта оценка обратно пропорциональна дисперсии наблюдаемых целевых значений.

3. Линейная регрессия с помощью sklearn

Разберем пример осуществления линейной регрессии с помощью библиотеки sklearn:

```
import pandas as pd
import numpy as np
```



```
import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error, r2_score #Загрузим две
метрики оценки качества
```

Получим датасет. Загрузим набор данных [Diabetes](#), в котором содержится информация о 442 пациентах с диабетом: возраст, пол, индекс массы тела, среднее артериальное давление и шесть измерений сыворотки крови, а также количественный показатель прогрессирования заболевания через год после исходного уровня.

Набор данных можно также получить, используя функцию [load_diabetes](#) из scikit-learn:

```
# Load the diabetes dataset

from sklearn import datasets

diabetes = datasets.load_diabetes(as_frame=True)

diabetes.frame.head()

>>>
```

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6	target
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019907	-0.017646	151.0
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068332	-0.092204	75.0
2	0.085299	0.050680	0.044451	-0.005670	-0.045599	-0.034194	-0.032356	-0.002592	0.002861	-0.025930	141.0
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022688	-0.009362	206.0
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031988	-0.046641	135.0

Последняя колонка target — наша целевая колонка, которую мы будем предсказывать. В ней лежит количественный показатель прогрессирования заболевания через год после исходного уровня.

```
diabetes.data
```

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019907	-0.017646
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068332	-0.092204
2	0.085299	0.050680	0.044451	-0.005670	-0.045599	-0.034194	-0.032356	-0.002592	0.002861	-0.025930
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022688	-0.009362
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031988	-0.046641
...
437	0.041708	0.050680	0.019662	0.059744	-0.005697	-0.002566	-0.028674	-0.002592	0.031193	0.007207
438	-0.005515	0.050680	-0.015906	-0.067642	0.049341	0.079165	-0.028674	0.034309	-0.018114	0.044485
439	0.041708	0.050680	-0.015906	0.017293	-0.037344	-0.013840	-0.024993	-0.011080	-0.046883	0.015491
440	-0.045472	-0.044642	0.039062	0.001215	0.016318	0.015283	-0.028674	0.026560	0.044529	-0.025930
441	-0.045472	-0.044642	-0.073030	-0.081413	0.083740	0.027809	0.173816	-0.039493	-0.004222	0.003064

442 rows x 10 columns

Атрибут `data` хранит только первые 10 колонок, которые описывают признаки.

```
diabetes.target
```

```
>>>
```

```
0      151.0
1       75.0
2      141.0
3      206.0
4      135.0
...
437    178.0
438    104.0
439    132.0
440    220.0
441     57.0
```

```
Name: target, Length: 442, dtype: float64
```

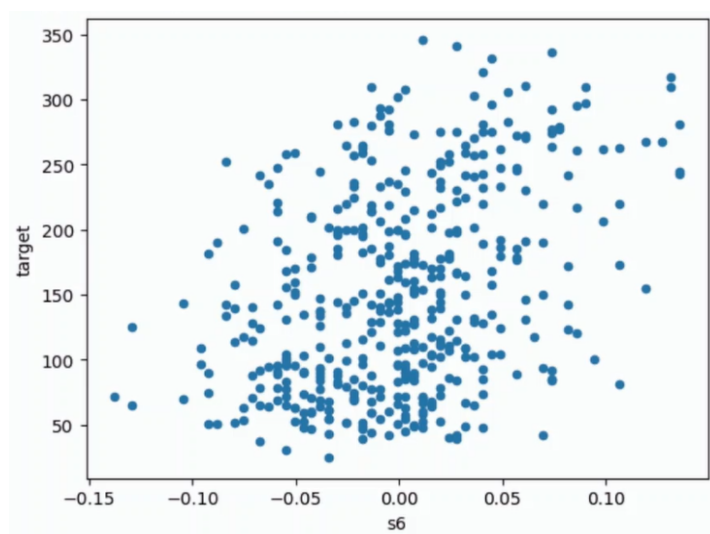
Простая линейная регрессия

В качестве первого примера можно использовать признак s6:

```
feature_name = 's6'
```

Построим диаграмму рассеяния между признаком s6 и целевой переменной:

```
diabetes.frame.plot.scatter(feature_name, 'target')
```



Осуществить линейную регрессию можно, используя специальный класс [LinearRegression](#) из scikit-learn:

```
# Create linear regression object
regr = LinearRegression()

# Train the model using the training sets
regr.fit(diabetes.data[[feature_name]], diabetes.target)

# Make predictions using the testing set
```

```
diabetes_y_pred = regr.predict(diabetes.data[[feature_name]])

print(f"Coefficients: \n{regr.coef_}")

print(f"Intercept: {regr.intercept_}")

print(f"Mean squared error: {mean_squared_error(diabetes.target,
diabetes_y_pred):.2f}")

print(f"Coefficient of determination: {r2_score(diabetes.target,
diabetes_y_pred):.2f}")

# print(f"Coefficient of determination:
{regr.score(diabetes.data[[feature_name]], diabetes.target):.2f}")

>>> Coefficients:

[619.22282068]

Intercept: 152.13348416289694

Mean squared error: 5062.38

Coefficient of determination: 0.15
```

Расчет метрик можно произвести не только с помощью отдельных функций из библиотеки `sklearn`, но также с помощью методов, встроенных в класс `linear regression`:

```
# Create linear regression object

regr = LinearRegression()

# Train the model using the training sets

regr.fit(diabetes.data[[feature_name]], diabetes.target)

# Make predictions using the testing set

diabetes_y_pred = regr.predict(diabetes.data[[feature_name]])
```

```
print(f"Coefficients: \n{regr.coef_}")

print(f"Intercept: {regr.intercept_}")

print(f"Mean squared error: {mean_squared_error(diabetes.target,
diabetes_y_pred):.2f}")

# print(f"Coefficient of determination: {r2_score(diabetes.target,
diabetes_y_pred):.2f}")

print(f"Coefficient of determination:
{regr.score(diabetes.data[[feature_name]], diabetes.target):.2f}")
```

```
>>> Coefficients:

[619.22282068]

Intercept: 152.13348416289694

Mean squared error: 5062.38

Coefficient of determination: 0.15
```

Результат тот же самый.

Визуализируем результаты нашей линейной регрессии:

```
plt.scatter(diabetes.data[feature_name], diabetes.target)

plt.plot(

    diabetes.data[feature_name],

    list(map(

        lambda x: regr.coef_[0]*x+regr.intercept_,

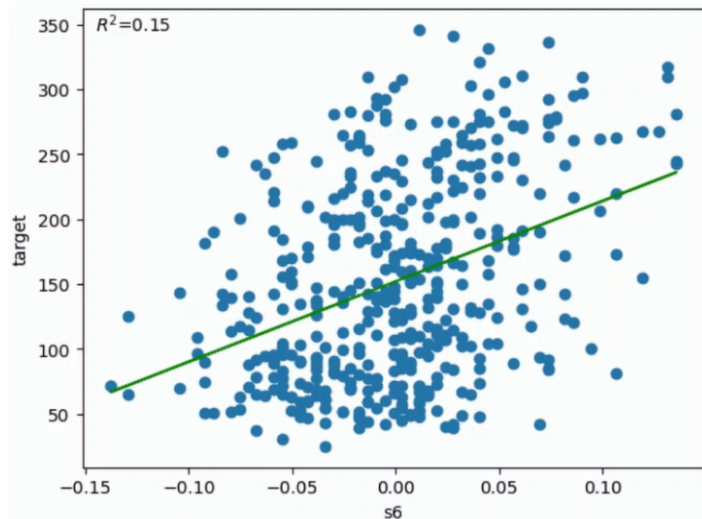
        diabetes.data[feature_name]

    )),

    color='g'

)
```

```
plt.text(-.145, 345, f'$R^2$={r2_score(diabetes.target,  
diabetes_y_pred):.2f}')  
  
plt.xlabel(feature_name)  
  
plt.ylabel('target')  
  
plt.show()
```



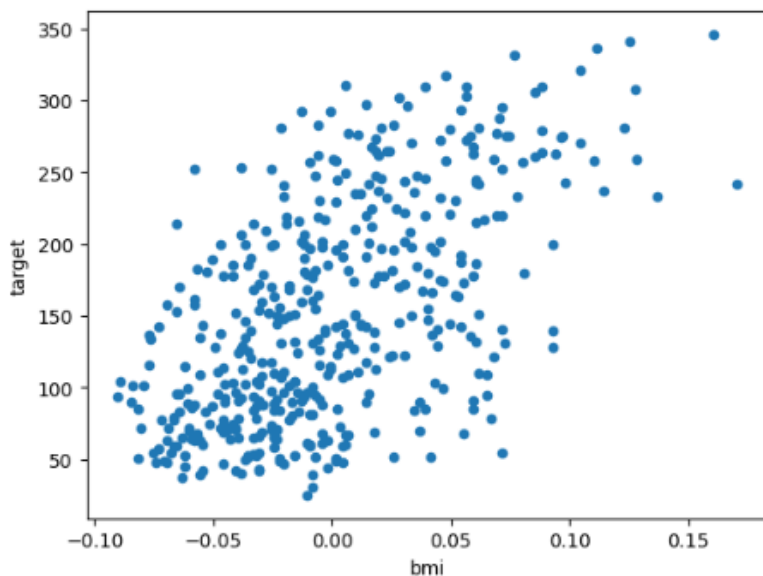
Видим, что коэффициент детерминации близок к нулю. 0,15 — очень маленькое значение. Эта модель регрессии очень слабо описывает наши данные.

Возьмем другой признак:

```
feature_name = 'bmi'
```

И повторим все то же самое для него:

```
diabetes.frame.plot.scatter(feature_name, 'target')
```



Уже видим, что данные разбросаны чуть меньше:

```
# Create linear regression object
regr = LinearRegression()

# Train the model using the training sets
regr.fit(diabetes.data[[feature_name]], diabetes.target)

# Make predictions using the testing set
diabetes_y_pred = regr.predict(diabetes.data[[feature_name]])

print(f"Coefficients: \n{regr.coef_}")

print(f"Intercept: {regr.intercept_}")

print(f"Mean squared error: {mean_squared_error(diabetes.target,
diabetes_y_pred):.2f}")
```

```
# print(f"Coefficient of determination: {r2_score(diabetes.target,  
diabetes_y_pred):.2f}")
```

```
print(f"Coefficient of determination:  
{regr.score(diabetes.data[[feature_name]], diabetes.target):.2f}")
```

```
>>> Coefficients:
```

```
[949.43526038]
```

```
Intercept: 152.13348416289617
```

```
Mean squared error: 3890.46
```

```
Coefficient of determination: 0.34
```

```
plt.scatter(diabetes.data[feature_name], diabetes.target)
```

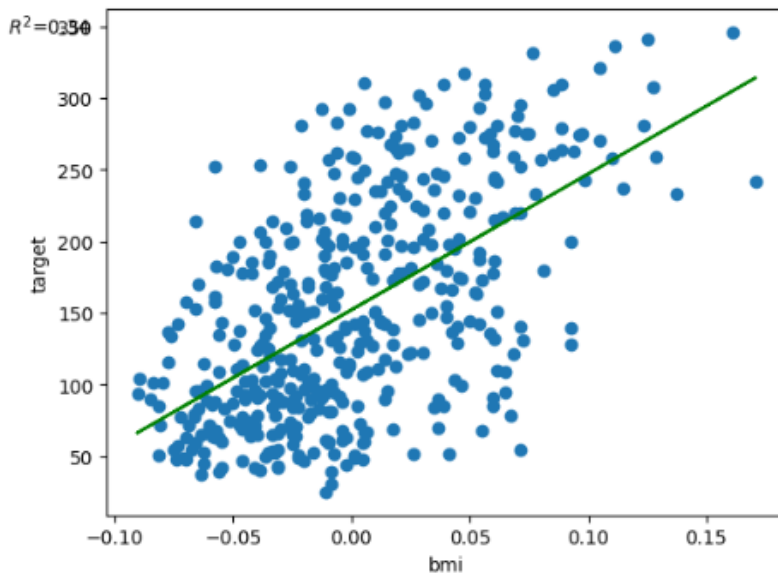
```
plt.plot(  
    diabetes.data[feature_name],  
    list(map(  
        lambda x: regr.coef_[0]*x+regr.intercept_,  
        diabetes.data[feature_name]  
    )),  
    color='g'  
)
```

```
plt.text(-.145, 345, f'$R^2$={r2_score(diabetes.target,  
diabetes_y_pred):.2f}')
```

```
plt.xlabel(feature_name)
```

```
plt.ylabel('target')
```

```
plt.show()
```

Здесь коэффициент детерминации чуть больше. Мы можем найти более подходящую модель, которая будет лучше описывать наши данные.

Множественная линейная регрессия

Множественной линейной регрессией называется такая модель, которая зависит от нескольких переменных. В данном случае мы будем использовать все признаки, которые имеются в нашем датасете.

```
# Create linear regression object
regr = LinearRegression()

# Train the model using the training sets
regr.fit(diabetes.data, diabetes.target)

# Make predictions using the testing set
diabetes_y_pred = regr.predict(diabetes.data)
```

```
print(f"Coefficients: \n{regr.coef_}")

print(f"Intercept: {regr.intercept_}")

print(f"Mean squared error: {mean_squared_error(diabetes.target,
diabetes_y_pred):.2f}")

print(f"Coefficient of determination: {r2_score(diabetes.target,
diabetes_y_pred):.2f}")

>>> Coefficients:

[ -10.0098663  -239.81564367  519.84592005  324.3846455  -792.17563855
  476.73902101  101.04326794  177.06323767  751.27369956   67.62669218]

Intercept: 152.13348416289597

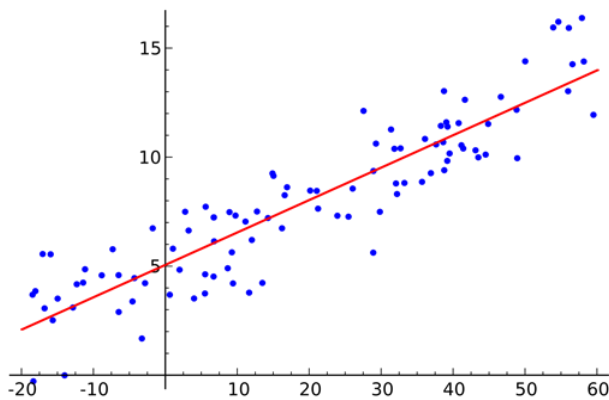
Mean squared error: 2859.70

Coefficient of determination: 0.52
```

Значение коэффициента детерминации значительно выше по сравнению с тем, когда мы использовали только один признак. То есть количество признаков может повлиять на качество результата модели. Но в нашем случае все равно качество оставляет желать лучшего.

4. Линейные модели машинного обучения

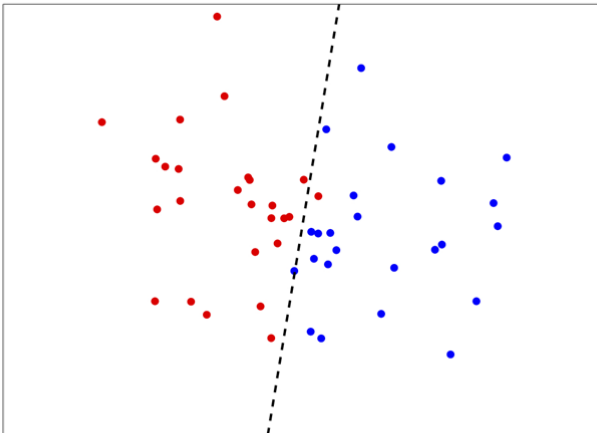
Линейные модели — одни из самых популярных и эффективных моделей в мире машинного обучения. В задаче линейной регрессии задается линейная зависимость между целевой переменной и исходными признаками:



Наша целевая переменная принимает значения как некоторая линейная комбинация, то есть взвешенная сумма исходных значений признаков:

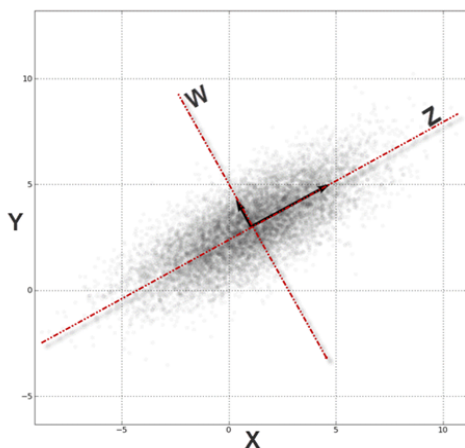
$$\hat{Y}_i = b_0 + b_1 X_i$$

В линейной задаче классификации появляется линейная разделяющая гиперплоскость:



При этом в задаче классификации сама модель уже может быть нелинейной.

Линейные модели могут применяться и в случае снижения размерности:



Эта картинка из метода главных компонент (PCA). Мы пытаемся найти линейное подпространство в исходном признаковом пространстве, в котором будет максимально точно описана исходная выборка с точки зрения потерь дисперсии.

Все линейные модели обладают тем свойством — они зависят только от линейных комбинаций признаков модели. Из этого следует их замечательное свойство — они быстрые в отличие от того же kNN.

Линейные модели встречаются в огромном числе задач:

- регрессии;
- классификации;
- в методах снижения размерности;
- в задачах обучения без учителя.

Линейные модели — те самые строительные блоки, из которых строятся другие более сложные модели. Линейные модели можно ассамблировать, они могут частично присутствовать в деревьях. Вообще, решающее дерево — это линейная модель над другим признаковым пространством.

Линейные модели — строительные блоки нейронных сетей. Нейронные сети — это линейные преобразования и нелинейные функции активации.

Пока мы пытаемся напрямую отобразить значение признакового пространства в пространство целевой переменной. Потом научимся отображать через

промежуточные представления для получения более качественного результата.

5. Линейная регрессия

Поставим формально задачу линейной регрессии:

$$L = \{x_i, y_i\}_{i=1}^N, x_i \in R^n, y_i \in R$$

Это задача обучения с учителем. У нас есть выборка L и значения целевой переменной y_i .

Наша оценка целевой переменной есть линейная комбинация исходных признаков с некоторыми весами с некоторым свободным членом:

$$\hat{y} = \omega_0 + \sum_{k=1}^p x_k \cdot \omega_k = \{x = [1, x_1, x_2, \dots, x_p]\} = x^T w$$

$w = (\omega_0, \omega_1, \dots, \omega_n)$ – вектор весов.

Если у нас не будет свободного члена, то все линейные зависимости будут проходить через 0. Но далеко не всегда наши данные проходят через 0. Свободный член отвечает за сдвиг относительно нуля.

Мы сделали упрощение формулы и добавим к вектору x единицу:

$$x = [1, x_1, x_2, \dots, x_p]$$

Никакого физического смысла это не несет.

В таком предположении мы можем сказать, что оптимальное решение для задачи линейной регрессии то, которое доставляет минимум некоторой функции ошибки:

$$\hat{w} = \arg \min_w \|Y - \hat{Y}\|_2^2 = \arg \min_w \|Y - Xw\|_2^2$$

Для простоты возьмем среднюю квадратичную ошибку. Значит, нужно минимизировать среднее значение квадрата отклонений.

Эта задача имеет аналитическое решение. Распишем нашу функцию потерь:

$$Q(w) = (Y - Xw)^T(Y - Xw) = \|Y - Xw\|_2^2,$$

$$X = [x_1, \dots, x_n], x_i \in R^p, Y = [y_1, \dots, y_n], y_i \in R$$

Возьмем производную:

$$\nabla_w Q(w) = \nabla_w [Y^T Y - Y^T X w - w^T X^T Y + w^T X^T X w] =$$

$$= 0 - X^T Y - X^T Y + (X^T X + X^T X) w = 0.$$

Получаем оптимальное значение параметров:

$$\hat{w} = (X^T X)^{-1} X^T Y$$

Что, если матрица $X^T X$ необратима? Матрица X — это число объектов \times число признаков. В общем случае это неквадратная матрица. Обратную матрицу у неквадратной матрицы взять нельзя.

$X^T X$ — эта матрица квадратная. В каком случае она может оказаться необратимой? Когда признаки линейно зависимы. В этом случае матрица окажется вырожденной. Обратную матрицу у вырожденной посчитать невозможно. Значит, решение найти нельзя. Что делать?

Если при малом изменении обучающих данных модель сильно меняется, то она неустойчива.

Пример. Допустим, представлено истинное значение w . Второй и третий признак зависимы друг от друга и отличаются на какой-то случайный шум.

```
w_true
array([ 2.68647887, -0.52184084, -1.12776533])

w_star = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(Y)
w_star
array([ 2.68027723, -186.0552577, 184.41701118])
```

Нашли оценку вектора весов. Сумма во втором и третьем векторе практически одинакова. Была информация только о линейной комбинации признаков. В случае когда признаки созависимы, скоррелированы, наше решение неустойчиво.

Если исходная задача неразрешима, и мы постулируем, что в исходной задаче не единственное решение, то нужно вводить дополнительное ограничение на наше решение:

$$\hat{w} = (X^T X + \lambda I)^{-1} X^T Y \quad (1)$$

$$I = \text{diag}[1, \dots, 1_p]$$

Введем понятие регуляризации. **Регуляризация** — по сути, ограничение. Мы к матрице $X^T X$ добавляем диагональную, и тогда она перестанет быть вырожденной. Мы сможем получить единственное решение.

Аналитическое решение (1) — задача минимизации величины:

$$Q(w) = \|Y - Xw\|_2^2 + \lambda^2 \|w\|_2^2$$

У нас все еще сохраняется квадратичное отклонение, и появляется слагаемое с квадратом нормы вектора весов.

Каждый признак при решении задачи оптимизации обладает весом, наименьшим из возможных. При дальнейшем уменьшении начнет расти функция ошибки.

6. Теорема Гаусса-Маркова

Теорема Гаусса-Маркова очень простая, но при этом крайне важная. Многие результаты машинного обучения были достигнуты сначала эвристически, из каких-то интуитивных, математических, но не строго доказанных предположений, и показали свою эффективность на практике. Их используют и сейчас во всем мире, хотя многие решения до сих пор не имеют четкого обоснования.

Рассмотрим задачу регрессии. Наша модель линейна:

$$Y = Xw + \varepsilon, \varepsilon = [\varepsilon_1, \dots, \varepsilon_N]$$

То есть предсказание целевой переменной зависит от признакового описания линейным образом.

$\varepsilon = [\varepsilon_1, \dots, \varepsilon_N]$ — внешний шум. Есть во всех измерениях и исследованиях, не подлежит описанию.

Теорема Гаусса-Маркова. Если целевая переменная зависит от признаков линейным образом, и:

- ошибка является несмещенной:

$$E(\varepsilon_i) = 0 \quad \forall i - \text{мат. ожидание ошибки равно нулю};$$

- дисперсия ошибки существует и конечна:

$$\text{Var}(\varepsilon_i) = \sigma^2 < \infty \quad \forall i;$$

- ковариация любых двух ошибок не зависима:

$$\text{Cov}(\varepsilon_i, \varepsilon_j) = 0 \quad \forall i \neq j.$$

В таком случае решение, получаемое методом наименьших квадратов:

$$\hat{\mathbf{w}} = (X^T X)^{-1} X^T Y$$

Минимизация квадратичной ошибки — оптимальная оценка параметров среди несмещенных (Best Linear Unbiased Estimator, BLUE). Иными словами, если ошибка обладает перечисленными выше свойствами, то наша линейная модель будет оптимальной с точки зрения минимизации дисперсии, если будем минимизировать среднюю квадратичную ошибку.

Для чего нужен этот результат? В этом случае есть гарантия, что мы можем получить оптимальное аналитическое решение.

Стоит заметить, что теорема Гаусса-Маркова, несмотря на свою полезность, часто не работает. Как мы обсуждали ранее, матрица $X^T X$ может быть необратимой. В этом случае найти решение мы не сможем. «Починить» это можно, добавив регуляризацию, и тогда получим устойчивое решение.

Когда мы используем регуляризацию, мы получаем смещенное решение относительно исходной задачи. Получается, что теорема Гаусса-Маркова не работает, если мы принимаем любую регуляризацию.

Не всегда используют среднеквадратичную ошибку, можно применять и среднюю абсолютную ошибку:

- Для L2 регуляризации норма $\|w\|_2^2$, и $MSE = \frac{1}{n} \|x^T w - y\|_2^2$ — среднеквадратичная ошибка.
- Для L1 регуляризации норма $\|w\|_1$, и $MAE = \frac{1}{n} \|x^T w - y\|_1$ — средняя абсолютная ошибка.

Важно! Следует отличать норму, которую мы используем для подсчета отклонения, и норму, которую используем для регуляризации.

7. L1 и L2 регуляризация

Рассмотрим различные нормы в задаче регрессии.

Мы можем минимизировать как среднеквадратичную ошибку, так и среднюю абсолютную. Можно брать ошибки и с более высокими четными степенями.

Различные нормы доставляют нам различные функции ошибки. Первая норма — MAE, вторая норма — MSE. Мы можем использовать различные техники регуляризации: L1 и L2.

Все эти ошибки различны. Когда мы говорим про задачи машинного обучения, то имеем в виду решение оптимизационной задачи. Мы хотим минимизировать эту ошибку на данной выборке.

Если меняем функцию ошибки, то меняем оптимизационную задачу. Тогда поменяется и решение, и результат, и некоторые свойства. Поэтому выбирать функцию ошибки нужно аккуратно.

Разные нормы в различной регуляризации тоже доставляют различные результаты. Рассмотрим основное различие L1 и L2 регуляризаций.

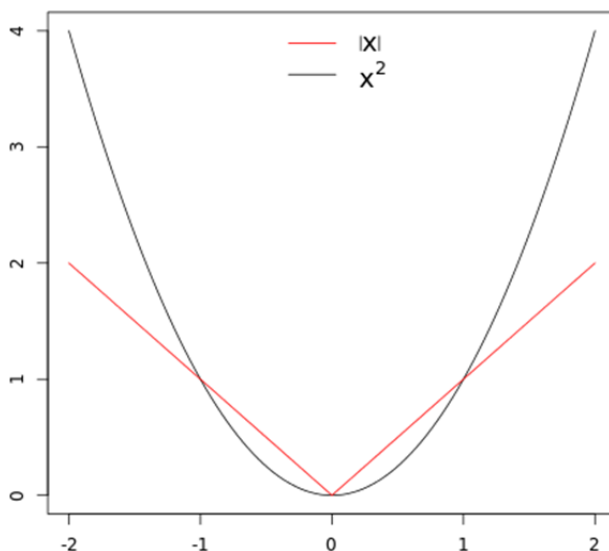
MSE-ошибка:

- Основание L2 нормы. По сути, это евклидово расстояние. Мы смотрим на квадрат отклонения.
- Это удобная ошибка с точки зрения того, что она доставляет оптимальное усреднение, согласно теореме Гаусса-Маркова.
- Она дифференцируема.

- Есть недостаток — она чувствительна к шумам. Если присутствуют шумы, то она увеличивает в квадрат раз.

MAE-ошибка:

- Не дифференцируема, в нуле есть разрыв производной. Но мы можем ее использовать в градиентных методах, поскольку там не требуется ее гладкость. Достаточно того, что мы знаем производную в каждой точке. В нуле можем доопределить нулем.
- MAE линейно увеличивает ошибку. Она гораздо более толерантна к шумам.
- Решение аналитически выразить нельзя.



L1 и L2 нормы можно использовать для регуляризации. Во всех регуляризациях мы накладываем дополнительное ограничение на наше решение.

L2-регуляризация:

- Позволяет добавить диагональный член в аналитическое решение.
- Ограничение на решение — добавляем вторую норму вектора весов. Оптимизируем за счет минимизации весов по второй норме. Норма вектора — сумма квадратов координат.

L1 регуляризация:

- Позволяет бороться с различными проблемами.

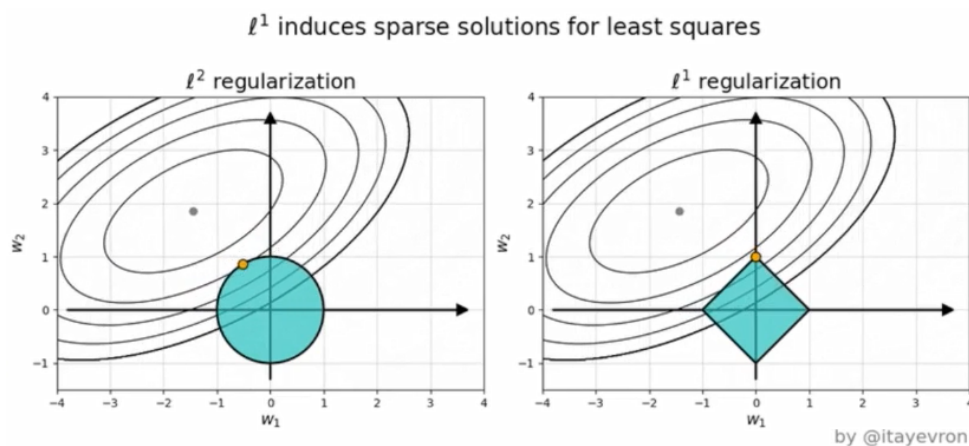
- Решение аналитически выражаться не будет.
- Ограничение на решение — добавляем первую норму вектора весов. Оптимизируем за счет минимизации весов по первой норме. Норма вектора — сумма модулей координат.

Физический смысл подходов L1 и L2 регуляризаций

L2 норма ограничивает максимальную норму нашего вектора и старается выровнять веса всех признаков.

L1 норма по факту отбирает признаки. Отбор признаков — зануление некоторых из них.

Иллюстрацию различия L1 и L2 регуляризаций можно посмотреть [в статье Introduction to Machine Learning](#).



На рисунке эллипсы — линии уровня функции потерь. Каждая точка — решение в двумерном пространстве параметров. В данном случае всего два параметра: w_1 и w_2 .

Каждая точка соответствует какому-то значению функции потерь. В центре линий эллипсов достигается минимум. У нас есть второй член — регуляризация. Слева на рисунке круг в L2 норме, справа — «круг» в L1 норме. Мы одновременно должны минимизировать и функцию потерь, и минимизировать норму вектора весов.

В случае L1 и L2 нормы желтая точка ведет себя по-разному. Желтая точка — оптимальное решение в зависимости от функции потерь, то есть от наших данных. Для L2 нормы желтая точка передвигается вдоль окружности. В случае L1 нормы

точка все время «сваливается» в какую-то вершину. То есть один из весов зануляется, и либо первый, либо второй признак не вносит вклад в итоговое решение.

Допустим, мы решаем оптимизационную задачу с помощью градиентных методов. Градиент у L2 нормы $2x$. Антиградиент, соответственно, $-2x$. У L1 нормы градиент и антиградиент $+1$ и -1 — всегда константа независимо от значения величины признака. При L2 регуляризации при минимизации функции ошибки градиент падает, когда мы приближаемся к оптимуму. В случае L1 регуляризации норма вектора весов ненулевая, ошибка всегда будет, градиент всегда будет ненулевым. В этом случае L1 регуляризация требует зануления каких-то признаков, иначе оптимум не будет достигнут.

Существует множество функций потерь:

- **R2 score** — коэффициент детерминации. Это оценка, насколько модель лучше или хуже среднего константного предсказания. Коэффициент детерминации не работает в случае деления на нулевое предсказание.
- **MAPE** — средняя абсолютная ошибка, нормированная на значение целевой переменной. Не зависит от значения целевой переменной, измеряется в процентах. Не устойчива к нулевым или близким к нулю значениям целевой переменной.
- **SMAPE** — в этом случае происходит деление не на истинное значение таргета, а на усредненное значение таргета и предсказания и т. д.

8. Решение линейной регрессии и анализ устойчивости решения

Рассмотрим линейную регрессию, ее аналитическое и градиентное решение. Повторим градиентные методы оптимизации и обсудим анализ устойчивости.

Решение линейной регрессии

Можно работать локально с ноутбука или работать облачно:

```
...  
  
If you are using Google Colab, uncomment the next line to download  
`utils_02.py`.  
  
...
```

```
# !wget  
https://raw.githubusercontent.com/girafe-ai/ml-course/22f_basic/week0_02_linear_reg/utils_02.py
```

Импортируем все необходимое:

```
import numpy as np  
import matplotlib  
import matplotlib.pyplot as plt  
import pandas as pd  
import warnings  
warnings.filterwarnings("ignore")  
  
random_seed = 45  
  
matplotlib.rcParams.update({'font.size': 16})
```

Создадим небольшую случайную выборку:

```
n_features = 2  
n_objects = 300  
batch_size = 10  
num_steps = 43  
np.random.seed(random_seed+1)  
  
# Let it be the *true* weights vector  
w_true = np.random.normal(size=(n_features, ))
```

```
X = np.random.uniform(-5, 5, (n_objects, n_features))

# For different scales of features. In case of 3 features the code is equal
# to the commented line below
# X *= np.arange([1, 3, 5])[None, :]
X *= (np.arange(n_features) * 2 + 1)[np.newaxis, :]

# Here comes the *true* target vector
Y = X.dot(w_true) + np.random.normal(0, 1, n_objects)
```

У нас есть линейная модель:

$$\hat{Y} = Xw$$

Средняя квадратичная функция ошибки:

$$Q(Y, X, w) = MSE(Y, Xw) = \|Y - Xw\|_2^2 = \sum_i (y_i - x_i^T w)^2$$

Аналитическое решение имеет вид:

$$w = (X^T X)^{-1} X^T Y$$

Посмотрим на оценку решения:

```
w_star = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(Y)
w_star
```

```
>>> array([0.59054644, 1.23044047])
```

```
w_true
```

```
>>> array([0.58487584, 1.23119574])
```

Несоответствие между w_{star} и w_{true} может быть из-за наличия шумов.

Посмотрим, что произойдет в случае коррелированных признаков:

```
n_features = 3
n_objects = 300
batch_size = 10
num_steps = 43
eps = 1e-3

# Let it be the *true* weights vector
w_true = np.random.normal(size=(n_features, ))

X = np.random.uniform(-5, 5, (n_objects, n_features))

# Now we duplicate the second feature with some small noise, so features 2
and 3 are collinear
X[:, -1] = X[:, -2] + np.random.uniform(-eps, eps, X[:, -2].shape)

# Here comes the *true* target vector
Y = X.dot(w_true) + np.random.normal(0, 1, (n_objects))
```

Второй и третий признак теперь совпадают вплоть до небольшого шума:

```
w_star = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(Y)
w_star
```

```
>>> array([-0.40979459, 35.18332853, -36.65086199])
```

```
w_true
```

```
>>> array([1.05992911, 0.30253587, 0.18578283])
```

Очень сильное отличие.

Посмотрим на оценку суммы второго и третьего признака весов:

```
w_star[1:].sum()
```

```
>>> 0.5181098596909948
```

```
w_star[1:].sum()
```

```
>>> 0.48831870290055673
```

От запуска к запуску они меняются, но остаются похожими. Наше решение — любая пара чисел. Таких чисел бесконечное множество.

Введем L2 регуляризацию:

$$Q_{reg}(Y, X, w) = MSE(Y, Xw) + \lambda ||w||_2^2 = ||Y - Xw||_2^2 + \lambda ||w||_2^2 = \sum_i (y_i - x_i^T w)^2 + \sum_p \omega_p^2$$

Получаем аналитическое решение:

$$w_{reg}^* = (X^T X + \lambda I_p)^{-1} X^T Y.$$

Матрица $X^T X + \lambda I_p$ обратима всегда.

Посмотрим на наше решение:

```
w_star_reg = np.linalg.inv(X.T.dot(X) +  
0.05*np.eye(n_features)).dot(X.T).dot(Y)  
w_star_reg
```



```
>>> array([ 0.81856465, -0.56904305, -0.29321512])
```

```
w_true[1:].sum()
```

```
>>> -0.8563631037988879
```

```
w_star_reg[1:].sum()
```

```
>>> -0.8622581626339355
```

Теперь у нас единственное решение.

Градиентный спуск

Градиентные решения гораздо более популярны на практике. В случае

аналитического решения $w_{reg}^* = \left(X^T X + \lambda I_p\right)^{-1} X^T Y$ размерность матрицы $X^T X$ есть p^2 .

Чтобы обратить матрицу, придется посчитать количество операций $p^3 + p^2 N$, где N — количество объектов.

Поэтому часто используют градиентный спуск:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \nabla Q(\mathbf{w}^{(t)})$$

Градиент для матрицы весов:

$$\nabla Q(\mathbf{w}) = -2X^T Y + 2X^T X \mathbf{w} = 2X^T (X \mathbf{w} - Y)$$

Рассмотрим пример, мы его уже видели ранее. Стохастический градиентный спуск:

```
n_features = 2  
n_objects = 300
```

```
batch_size = 10
num_steps = 43
np.random.seed(random_seed)

# Let it be the *true* weights vector
w_true = np.random.normal(size=(n_features, ))

X = np.random.uniform(-5, 5, (n_objects, n_features))

# For different scales of features. In case of 3 features the code is equal to the commented line below
# X *= np.arange([1, 3, 5])[None, :]
X *= (np.arange(n_features) * 2 + 1)[np.newaxis, :]

# Here comes the *true* target vector
Y = X.dot(w_true) + np.random.normal(0, 1, n_objects)
```

```
np.random.seed(random_seed)
w_0 = np.random.uniform(-2, 2, n_features)-0.5
w = w_0.copy()
w_list = [w.copy()]
step_size = 1e-2

for i in range(num_steps):
    w -= step_size * 2. * np.dot(X.T, (X.dot(w) - Y)) / Y.size # YOUR CODE HERE
    w_list.append(w.copy())
w_list = np.array(w_list)
```

```
# compute level set

A, B = np.meshgrid(np.linspace(-2, 2, 100), np.linspace(-2, 2, 100))

levels = np.empty_like(A)
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        w_tmp = np.array([A[i, j], B[i, j]])
        levels[i, j] = np.mean(np.power(np.dot(X, w_tmp) - Y, 2))

plt.figure(figsize=(13, 9))
plt.title('GD trajectory')
plt.xlabel('$w_1$')
plt.ylabel('$w_2$')
plt.xlim(w_list[:, 0].min() - 0.1, w_list[:, 0].max() + 0.1)
plt.ylim(w_list[:, 1].min() - 0.1, w_list[:, 1].max() + 0.1)
plt.gca().set_aspect('equal')

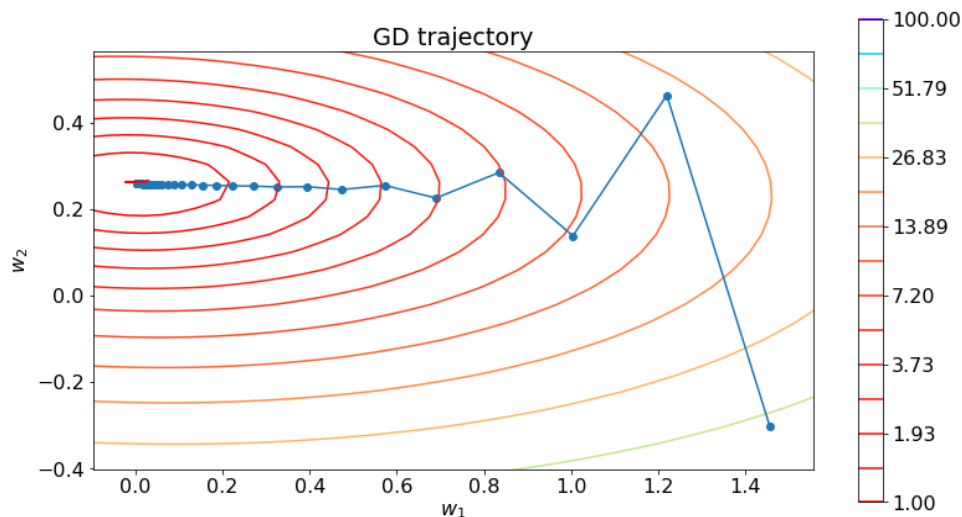
# visualize the level set

CS = plt.contour(A, B, levels, levels=np.logspace(0, 2, num=15),
cmap=plt.cm.rainbow_r)
CB = plt.colorbar(CS, shrink=0.8, extend='both')

# visualize trajectory

plt.scatter(w_true[0], w_true[1], c='r')
plt.scatter(w_list[:, 0], w_list[:, 1])
plt.plot(w_list[:, 0], w_list[:, 1])
```

```
plt.show()
```



Проведем анализ неустойчивости нашего решения. Под **неустойчивостью** будем понимать возможность получить различные значения матрицы весов при малом изменении обучающей выборки.

Число обусловленности:

$\kappa(a) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$, где $\sigma_{\max}(A)$ и $\sigma_{\min}(A)$ — максимальное и минимальное сингулярные числа матрицы A соответственно. Показывают, насколько «растянуты» в одну и другую стороны наши данные. Чем выше $\kappa(a)$, тем более неустойчиво наше решение.

Посмотрим на стохастический градиентный спуск:

```
def get_w_by_grad(X, Y, num_steps, w_0, lr):
    w = w_0.copy()

    for i in range(num_steps):
        w -= 2 * lr * np.dot(X.T, np.dot(X, w) - Y) / Y.shape[0]

    return w
```

```
def get_w_by_stoch_grad(X, Y, num_steps, w_0, lr_0, n_objects):  
    w = w_0.copy()  
    lr_0 = 0.45  
  
    for i in range(num_steps):  
        lr = lr_0 / ((i+1)**0.51)  
        sample = np.random.randint(n_objects, size=batch_size)  
        w -= 2 * lr * np.dot(X[sample].T, np.dot(X[sample], w) -  
            Y[sample]) / Y.shape[0]  
    return w  
  
def rmse(y_true, y_pred):  
    return np.linalg.norm(y_true-y_pred)
```

Минимизация средней квадратичной ошибки — выпуклая задача. Тогда решение, которое находится аналитическим методом, и решение, которое находится градиентным спуском, эквивалентны друг другу. Но если матрица A вырожденная или близка к таковой, то появляются вычислительные ошибки.

Посмотрим на графики:

```
lr = 1e-3  
sgd_lr = 0.1  
num_steps = 250  
noise_eps_seq = np.logspace(-2, -6, 20)  
  
w_0 = np.random.uniform(-2, 2, (n_features))
```

```
condition_numbers = []
vector_norms_list = []
rmse_list = []
results_list = []
for eps in noise_eps_seq:
    local_condition_numbers = []
    local_vector_norms_list = []
    local_rmse_list = []
    for i in range(50):
        X[:, -1] = 2 * (X[:, -2] + np.random.uniform(-eps, eps, X[:,
-2].shape))

    a = np.linalg.eigvals(X.T.dot(X))
    local_condition_numbers.append(a.max() / a.min())

    w_star = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(Y)
    w_star_grad = get_w_by_grad(X, Y, num_steps, w_0, lr)
    w_star_sgd = get_w_by_stoch_grad(X, Y, num_steps, w_0, sgd_lr,
n_objects)

    local_vector_norms_list.append([
        np.linalg.norm(w_star),
        np.linalg.norm(w_star_grad),
        np.linalg.norm(w_star_sgd),
    ])

    analytical_predict = X.dot(w_star)
    grad_predict = X.dot(w_star_grad)
```

```
sgd_predict = X.dot(w_star_sgd)

local_rmse_list.append([
    rmse(Y, analytical_predict),
    rmse(Y, grad_predict),
    rmse(Y, sgd_predict),
])

results_list.append([w_star, w_star_grad, w_star_sgd])

condition_numbers.append([np.mean(local_condition_numbers),
np.std(local_condition_numbers)])

vector_norms_list.append([
    np.mean(np.array(local_vector_norms_list), axis=0),
    np.std(np.array(local_vector_norms_list), axis=0),
])

rmse_list.append(np.mean(np.array(local_rmse_list), axis=0))

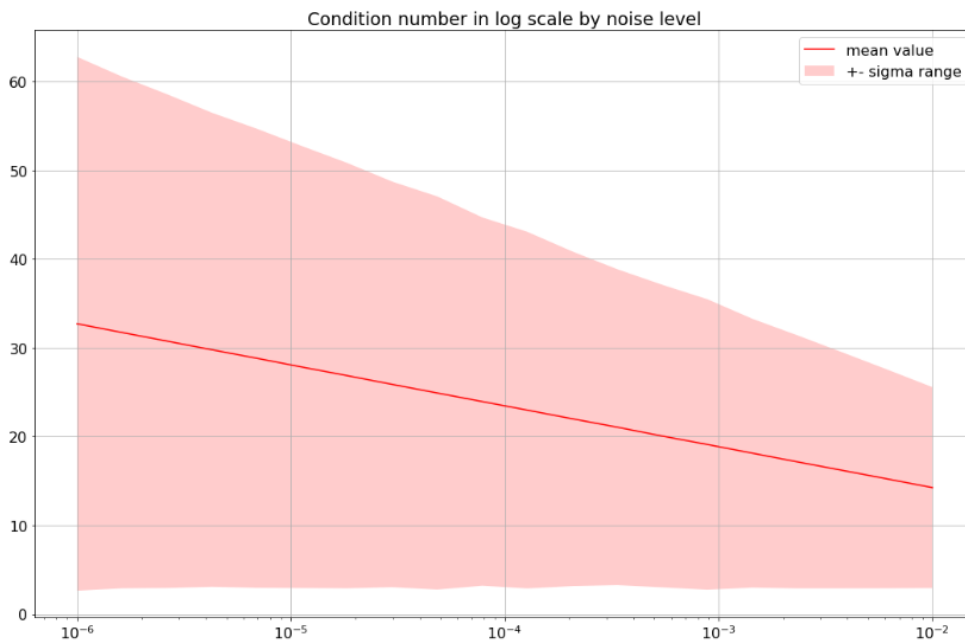
condition_numbers = np.array(condition_numbers)
vector_norms_list = np.array(vector_norms_list)
rmse_list = np.array(rmse_list)
```

Мы повторяем множество экспериментов, получаем решение с помощью градиентного спуска и стохастического градиентного спуска. Считаем среднюю квадратичную ошибку для каждого решения.

Зашумляем наши данные:

```
from utils_02 import visualise
```

```
visualise(
    np.log(condition_numbers[:, 0]),
    np.log(condition_numbers[:, 1]),
    noise_eps_seq,
    title='Condition number in log scale by noise level',
    greater_than_zero=True,
    log_scale=True
)
```



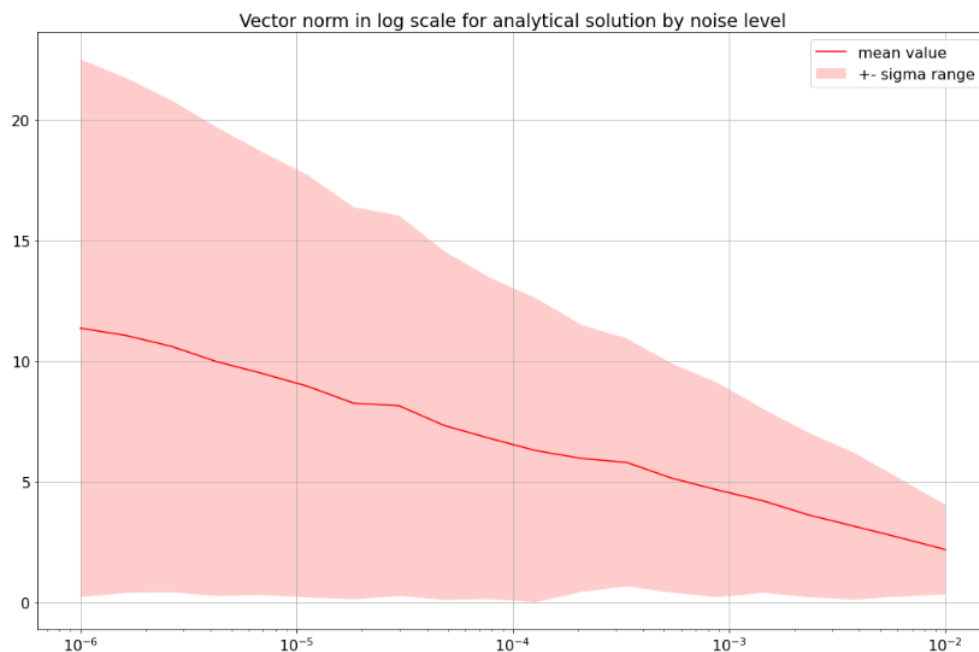
На графике по оси x — шум, по оси y — число обусловленности. Получаем, что чем больше шум, тем меньше число обусловленности, то есть тем больше похожи друг на друга признаки нашей выборки. Чем выше число обусловленности, тем менее устойчиво решение.

Посмотрим на норму вектора весов аналитического решения в зависимости от шума:

```
visualise(
    np.log(vector_norms_list[:, 0, 0]),
```



```
np.log(vector_norms_list[:, 1, 0]),
noise_eps_seq,
title='Vector norm in log scale for analytical solution by noise
level',
greater_than_zero=True,
log_scale=True
)
```

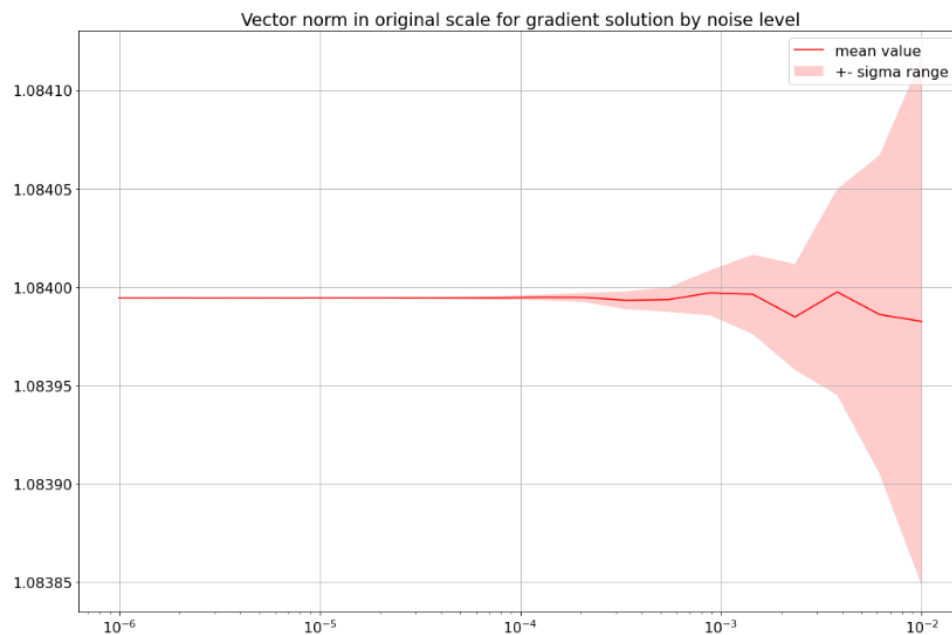


Чем больше шум, который позволяет нам делать признаки непохожими друг на друга, тем меньше дисперсия вектора весов.

Теперь посмотрим для градиентного решения:

```
visualise(
    vector_norms_list[:, 0, 1],
    vector_norms_list[:, 1, 1],
    noise_eps_seq,
    title='Vector norm in original scale for gradient solution by noise
```

```
level',
    greater_than_zero=True,
    log_scale=True
)
```



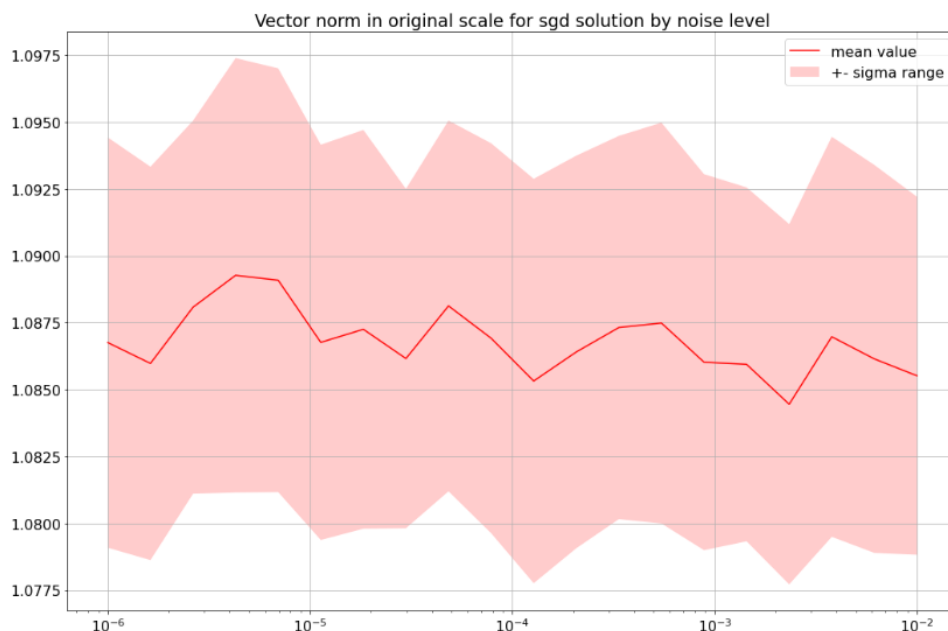
Мы видим, что градиентное решение всегда сходится в одну и ту же точку, поскольку здесь нет проблем с обращением матрицы, и нет численной ошибки. Но чем больше шума, тем больше мы будем получать различные решения.

Посмотрим на стохастический градиентный спуск:

```
visualise(
    vector_norms_list[:, 0, 2],
    vector_norms_list[:, 1, 2],
    noise_eps_seq,
    title='Vector norm in original scale for sgd solution by noise
level',
    greater_than_zero=True,
```

```
log_scale=True
```

```
)
```



Когда данные скоррелированы, нужно использовать регуляризацию. Стохастический градиентный спуск тоже дает неплохие результаты.

Линейная регрессия

Импортируем все необходимое:

```
from sklearn.linear_model import LinearRegression, Lasso, Ridge
```

Lasso — L1 регуляризация;

Ridge — L2 регуляризация;

LinearRegression дает нам аналитическое решение.

Чтобы оценить качество, будем использовать коэффициент детерминации:

$$R^2 = 1 - \frac{\sum_i (y_i - a(x_i))^2}{\sum_i (y_i - \bar{y}_i)^2}$$

Видим отношение средней квадратичной ошибки к дисперсии ответов. Будем использовать большую выборку:

```
n_features = 700
n_objects = 100000
num_steps = 150

w_true = np.random.uniform(-2, 2, (n_features, 1))

X = np.random.uniform(-100, 100, (n_objects, n_features))
Y = X.dot(w_true) + np.random.normal(0, 10, (n_objects, 1))
```

Измерим время для наших регрессий:

```
%%time

lr = LinearRegression()
lr.fit(X, Y)
print(f'R2: {lr.score(X, Y)}')
```

>>> 22.6 s — оптимизация для аналитического решения.

```
%%time

lr = Ridge(alpha=0.0, solver='sparse_cg')
lr.fit(X, Y)
print(f'R2: {lr.score(X, Y)}')
```

>>> 555 ms — оптимизация для градиентного спуска.

Теперь построим свою модель:

```
from sklearn.base import BaseEstimator, RegressorMixin
# also ClassifierMixin and TransformerMixin exist

class LinearRegressionSGD(BaseEstimator, RegressorMixin):
    '''LinearRegression with L2 regularization and SGD optimizer
    ...

    def __init__(
        self, C: float=1.0,
        batch_size: int=25,
        lr: float=1e-2,
        num_steps: int=200,
    ) -> None:
        self.C = C
        self.batch_size = batch_size
        self.lr = lr
        self.num_steps = num_steps

    def fit(self, X, Y):
        w = np.random.randn(X.shape[1])(:, None)
        n_objects = len(X)

        # this is just copied from above
        for i in range(self.num_steps):
            sample_indices = np.random.randint(n_objects,
            size=self.batch_size)
```

```
        w -= 2 * self.lr * np.dot(X[sample_indices].T,  
np.dot(X[sample_indices], w) - Y[sample_indices]) / self.batch_size  
  
        self.w = w  
        return self  
  
    def predict(self, X):  
        return X@self.w
```

Создаем датасет:

```
n_features = 700  
n_objects = 100000  
num_steps = 150  
  
w_true = np.random.uniform(-2, 2, (n_features, 1))  
  
X = np.random.uniform(-100, 100, (n_objects, n_features)) *  
    np.arange(n_features)  
Y = X.dot(w_true) + np.random.normal(0, 10, (n_objects, 1))
```

Разбиваем на train и валидацию:

```
from sklearn.model_selection import train_test_split  
x_train, x_test, y_train, y_test = train_test_split(X, Y)
```

Запускаем оптимизацию:

```
own_lr = LinearRergessionSGD().fit(x_train, y_train)
```

```
print(f'R2: {own_lr.score(x_test, y_test)}')
```

И у нас получается ошибка. Мы забыли отнормировать наши данные, получили слишком большие градиенты:

```
from sklearn.preprocessing import StandardScaler
```

Отнормируем наши данные:

```
scaler = StandardScaler()
scaler.fit(x_train)
x_scaled = scaler.transform(x_train)
```

Обучаем learning rate:

```
own_lr = LinearRergessionSGD().fit(x_scaled, y_train)
```

Преобразуем тестовые данные:

```
x_test_scaled = scaler.transform(x_test)
```

```
print(f'R2: {own_lr.score(x_test_scaled, y_test)}')
```

>>> 0.9965142196068342 — все хорошо работает.

Мы можем строить pipeline — последовательности операций:

```
from sklearn.pipeline import make_pipeline
```

```
pipe = make_pipeline(
    StandardScaler(),
    LinearRergessionSGD(),
```

```
)
```

```
pipe.fit(x_train, y_train)
print(f'R2: {pipe.score(x_test, y_test)}')
```

>>> 0.9964865310951883 — результат такой же.

Дополнительные материалы для самостоятельного изучения

1. [Jonathan Shewchuk. Introduction to Machine Learning](#)
2. https://raw.githubusercontent.com/girafe-ai/ml-course/22f_basic/week0_02_linear_regression/utils_02.py
3. [Diabetes](#)