Методические рекомендации. Регрессионный анализ

Введение

Обучение модели представляет собой последовательный процесс, состоящий из нескольких взаимосвязанных этапов, каждый из которых играет важную роль в достижении корректных и воспроизводимых результатов. Успех построения модели в значительной степени зависит от того, насколько качественно подготовлены данные, выбраны алгоритмы и настроены параметры обучения.

Первым и, пожалуй, самым важным шагом является подготовка данных. Сырые числовые и категориальные признаки редко могут быть использованы напрямую, так как они часто содержат различные масштабы, выбросы, пропущенные значения и несопоставимые категории. Для корректной работы алгоритмов необходимо провести масштабирование, нормализацию и стандартизацию признаков, устранить или преобразовать выбросы, а также выполнить корректное кодирование категориальных переменных. Эти операции обеспечивают устойчивость моделей и позволяют избежать искажений при вычислении зависимостей. Пренебрежение этапом предварительной обработки может привести к тому,

что модель начнет ошибочно считать одни признаки более значимыми, чем другие, или вовсе потеряет способность адекватно обучаться.

Не менее важной задачей является работа с пропущенными значениями. Их наличие может нарушить корректность вычислений и привести к смещению результатов. Поэтому данные либо очищаются, либо значения восстанавливаются с помощью статистических или модельных методов. Неверное обращение с пропусками может привести к потере информации или систематическим ошибкам, особенно если данные отсутствуют не случайно.

После подготовки данных и построения модели выполняется её оценка. Для этого используются методы перекрёстной проверки и различные метрики — среднеквадратичная ошибка, средняя абсолютная ошибка, коэффициент детерминации и другие. Правильная оценка позволяет определить, насколько хорошо модель обобщает информацию и сохраняет устойчивость к новым данным. Отсутствие этого этапа чревато переобучением и снижением качества предсказаний на реальных данных.

После завершения подготовки данных осуществляется построение модели. На этом этапе подбирается тип регрессионной модели — линейная, полиномиальная, регуляризованная (гребневая, лассо, эластичная сеть) — и выполняется процесс обучения на подготовленных данных. Неправильный выбор модели или её гиперпараметров может привести к переобучению, недообучению или искажению результатов.

После построения модели проводится её оценка с использованием метрик качества, таких как среднеквадратичная ошибка, средняя абсолютная ошибка и коэффициент детерминации.

В заключение выполняется отбор модели и настройка её гиперпараметров. Этот процесс позволяет определить оптимальную комбинацию гиперпараметров, при которой достигается наилучшее качество и стабильность работы. Использование процедур вроде исчерпывающего или рандомизированного поиска обеспечивает объективность выбора и предотвращает субъективные ошибки при подборе настроек.

Таким образом, процесс обучения модели включает в себя логически последовательные этапы: от подготовки данных и обработки признаков до оценки и отбора оптимальной модели. Каждый из них является необходимым условием для получения корректных, интерпретируемых и практически применимых результатов. В последующих разделах мы более подробно рассмотрим методы работы с числовыми и категориальными данными, способы оценки качества моделей, а также процедуры подбора и построения регрессионных моделей.

1 Работа с числовыми данными

Количественные данные что-то измеряют — будь то размер класса, ежемесячные продажи или оценки учащихся. Естественным способом представления этих величин является численное: например, 29 студентов, 529 392 долларов продаж.

Однако для корректной работы алгоритмов машинного обучения часто недостаточно просто подать такие данные «как есть». Требуется их преобразовать, чтобы они лучше соответствовали математическим предположениям моделей и не искажали результаты обучения. В этой части мы рассмотрим стратегии преобразования сырых числовых данных в признаки, которые целенаправленно готовятся для машинного обучения.

1.1 Масштабирование и нормализация признаков

MinMax-шкалирование признака

Когда мы работаем с реальными данными, очень часто встречается ситуация, что одни признаки измеряются в маленьких числах, например, в долях, а другие могут иметь десятки или даже тысячи. Для алгоритмов машинного обучения это проблема: модель может "считать", что признаки с большими значениями важнее, чем те, у которых значения маленькие, хотя на самом деле это не так.

Чтобы избежать такой несправедливости, данные принято **приводить к одному масштабу**. Один из самых простых и популярных способов — **MinMax-шкалирование**. Суть метода в том, что каждый признак преобразуется так, чтобы его минимальное значение стало равным нижней границе диапазона (обычно 0), а максимальное — верхней границе (обычно 1). Все остальные значения "растягиваются" пропорционально и также попадают в этот диапазон.

Для шкалирования числового признака можно использовать класс MinMaxScaler из библиотеки scikit-learn.

```
# Загрузить библиотеки
import numpy as np
from sklearn import preprocessing

# Создать признак
feature = np.array([[-500.5], [-100.1], [0], [100.1], [900.9]])

# Создать шкалировщик
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))

# Прошкалировать признак
scaled_feature = minmax_scale.fit_transform(feature)

# Показать прошкалированный признак
scaled_feature
```

Результат:

```
array([[0.        ],
       [0.28571429],
       [0.35714286],
       [0.42857143],
       [1.        ]])
```

Формула:

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

где:

- x — исходный вектор признака
- x_i — отдельный элемент
- x'_i — значение признака после шкалирования

В примере выше минимальное значение признака (−500.5) преобразуется в 0, а максимальное (900.9) — в 1. Все остальные значения оказываются пропорционально распределёнными внутри этого интервала.

Особенности применения

В классе MinMaxScaler есть два варианта работы:

- Использовать `fit` для вычисления минимального и максимального значений признака, а затем применять `transform` для масштабирования.
- Использовать `fit_transform`, который объединяет обе операции.

Математической разницы между ними нет. Однако на практике иногда полезно разделять шаги: например, когда нужно применить одно и то же преобразование к разным наборам данных (train, valid, test). В этом случае `fit` применяется только к обучающему набору, а `transform` — ко всем остальным. Это позволяет избежать утечки данных и сохранить воспроизводимость эксперимента.

Когда использовать и ограничения

- Используйте MinMax, если алгоритм чувствителен к относительным расстояниям и вы хотите сохранить форму распределения и порядок наблюдений.
- Будьте внимательны к выбросам: крайние значения определяют растяжение диапазона. При наличии выбросов рассмотрите робастные альтернативы.

1.2 Стандартизация признака (Z-оценка)

Когда признаки имеют разные диапазоны и распределения, модели машинного обучения могут работать нестабильно. Особенно это важно для методов, которые чувствительны к масштабу и статистическим свойствам данных, например линейная регрессия, логистическая регрессия, SVM.

Задача стандартизации — преобразовать числовой признак так, чтобы он имел:

- среднее значение, равное **0**,
- стандартное отклонение, равное **1**.

Иными словами, мы пересчитываем значения признака относительно его центра (среднего) и масштаба (разброса). После такого преобразования можно сравнивать разные признаки напрямую, так как они становятся сопоставимыми.

Пример: если у нас есть два признака — «возраст» (в годах) и «зарплата» (в тысячах рублей), то до стандартизации зарплата может иметь значения в сотнях, а возраст — максимум 100. После стандартизации оба признака будут выражены в «стандартных единицах» и окажутся на одном уровне влияния для модели.

Класс `StandardScaler` библиотеки scikit-learn выполняет оба преобразования:

```
# Загрузить библиотеки
import numpy as np
from sklearn import preprocessing

# Создать признак
x = np.array([[-1000.1],
```

```
            [-200.2],
            [500.5],
            [600.6],
            [9000.9]])

# Создать шкалировщик
scaler = preprocessing.StandardScaler()

# Преобразовать признак
standardized = scaler.fit_transform(x)

#Показать признак
standardized
```

Результат:

```
array([[-0.76058269],
       [-0.54177196],
       [-0.35009716],
       [-0.32271504],
       [ 1.97516685]])
```

Формула стандартизации:

$$x'_i = \frac{x_i - \mu}{\sigma}$$

где:

- x_i — исходный элемент признака
- μ — среднее значение признака
- σ — стандартное отклонение признака
- x'_i — стандартизированное значение (z-оценка)

Таким образом, каждое значение признака показывает, на сколько стандартных отклонений оно отличается от среднего.

Как выбрать между StandardScaler и MinMaxScaler

- Если данные имеют распределение, близкое к нормальному, и вы хотите центрировать признаки вокруг 0 с одинаковым разбросом → **StandardScaler**.
- Если важно сохранить пропорции и «растянуть» данные в фиксированный диапазон (например, для методов, где расстояния критичны) → **MinMaxScaler**.
- Если есть выбросы, стоит быть осторожным: у **StandardScaler** они смещают среднее и увеличивают дисперсию, у **MinMaxScaler** они растягивают диапазон и «сжимают» остальные данные. В таких случаях лучше применять **RobustScaler**, который учитывает медиану и межквартильный размах.
- Общее правило: если нет веской причины использовать альтернативу — по умолчанию стоит применять стандартизацию.

Проверка результата

```
# Напечатать среднее значение и стандартное отклонение
print("Среднее:", round(standardized.mean()))
print("Стандартное отклонение:", standardized.std())
```

Результат:

```
Среднее: 0.0
Стандартное отклонение: 1.0
```

1.3 Работа с выбросами

Если данные содержат значительные выбросы, они могут сильно исказить среднее и дисперсию, что снижает эффективность стандартизации. В таких случаях полезно использовать робастное шкалирование — преобразование на основе медианы и межквартильного размаха. В scikit-learn это реализуется через класс `RobustScaler`:

```
# Создать робастный шкалировщик
robust_scaler = preprocessing.RobustScaler()

# Преобразовать признак
robust_scaler.fit_transform(x)
```

Результат:

```
array([[-1.87387612],
       [-0.875     ],
       [ 0.        ],
       [ 0.125     ],
       [10.61488511]])
```

Когда использовать и ограничения

- Стандартизация устойчива для многих линейных моделей и алгоритмов, использующих градиентный спуск.
- При тяжёлых хвостах распределения и выбросах предпочитайте `RobustScaler`.

1.4 Генерирование полиномиальных и взаимодействующих признаков

Иногда исходных признаков недостаточно, чтобы модель уловила сложные зависимости. В таких случаях полезно создать новые признаки — полиномиальные (возведение признака в степень) и взаимодействующие (произведения разных признаков).

Например:

- если есть признак «x», можно добавить x^2, x^3 и т.д.;

- если есть признаки «x» и «y», можно добавить новый — произведение x·y, отражающее их взаимодействие.

Такие преобразования расширяют пространство признаков и позволяют моделям находить **нелинейные зависимости** без явного использования сложных алгоритмов.

Такие признаки обычно формируют вручную, но библиотека scikit-learn предлагает готовый инструмент — класс `PolynomialFeatures`.

```
# Загрузить библиотеки
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Создать матрицу признаков
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# Создать объект PolynomialFeatures
polynomial_interaction = PolynomialFeatures(degree=2, include_bias=False)

# Создать полиномиальные признаки
polynomial_interaction.fit_transform(features)
```

Результат:

```
array([[2., 3., 4., 6., 9.],
       [2., 3., 4., 6., 9.],
       [2., 3., 4., 6., 9.]])
```

Параметр `degree` определяет максимальный порядок полинома. Например:

- `degree=2` создаст признаки второй степени:

  x_1, x_2, x_1^2, $x_1 \cdot x_2$, x_2^2

- `degree=3` создаст признаки второй и третьей степени:

  x_1, x_2, x_1^2, $x_1 \cdot x_2$, x_2^2, x_1^3, $x_1^2 \cdot x_2$, $x_1 \cdot x_2^2$, x_2^3

Кроме того, по умолчанию `PolynomialFeatures` формирует не только степени отдельных признаков, но и признаки их взаимодействия. Если необходимо оставить только взаимодействия, используйте параметр `interaction_only=True`:

```
interaction = PolynomialFeatures(degree=2,
                                 interaction_only=True,
                                 include_bias=False)
interaction.fit_trans form(features)
```

Результат:

```
array([[2., 3., 6.],
       [2., 3., 6.],
```

```
       [2., 3., 6.]])
```

Полиномиальные признаки полезны, когда предполагается нелинейная зависимость между признаками и целевой переменной. Например, влияние возраста на вероятность заболевания может усиливаться с возрастом. Простое включение признака «возраст» может быть недостаточно, а добавление возраста в квадрате или кубе поможет уловить этот эффект.

Ещё один важный случай — взаимодействие признаков, когда эффект одного признака зависит от другого. Пример с кофе:

1. был ли кофе перемешан
2. добавляли ли сахар

По отдельности эти признаки не объясняют сладость. Однако их комбинация даёт точный ответ: кофе будет сладким только когда есть сахар и напиток перемешан. Такой эффект кодируется взаимодействующим признаком — произведением двух исходных признаков.

Таким образом, полиномиальные и взаимодействующие признаки помогают моделям улавливать более сложные зависимости, которые линейная модель в исходном виде заметить не способна.

Когда использовать и ограничения

- Эффективно для линейных моделей, которым требуется «обогатить» пространство признаков.
- Быстро растёт размерность. Используйте регуляризацию и отбор признаков.
- Масштабируйте признаки до генерации, чтобы коэффициенты были сопоставимы.

1.5 Преобразование существующих признаков

При подготовке данных бывают ситуации, когда стандартных инструментов недостаточно. Например, нужно взять логарифм от признака, преобразовать значения через синус или косинус, закодировать нестандартное соотношение между столбцами. Такие операции помогают лучше отразить скрытые закономерности в данных и повысить качество модели. Чтобы не писать весь процесс обработки вручную, в библиотеке **scikit-learn** есть специальный класс — `FunctionTransformer` . Он позволяет «обернуть» любую функцию и применить её к данным как полноценный шаг обработки признаков. Это удобно тем, что трансформация легко интегрируется в `Pipeline` и воспроизводится при повторных запусках эксперимента.

```
# Загрузить библиотеки
import numpy as np
from sklearn.preprocessing import FunctionTransformer

# Создать матрицу признаков
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# Определить простую функцию
def add_ten(x):
    return x + 10
```

```
# Создать преобразователь
ten_transformer = FunctionTransformer(add_ten)

# Преобразовать матрицу признаков
ten_transformer.transform(features)
```

Результат:

```
array([[12, 13],
       [12, 13],
       [12, 13]])
```

Такое же преобразование можно создать в библиотеке pandas с помощью метода `apply`:

```
# Загрузить библиотеку
import pandas as pd

# Создать фрейм данных
df = pd.DataFrame(features, columns=["признак_1", "признак_2"])

# Применить функцию
df.apply(add_ten)
```

	признак_1	признак_2
0	12	13
1	12	13
2	12	13

В этом решении мы создали очень простую функцию `add_ten`, которая прибавляла 10 к каждому входу, но нет причин, по которым мы не могли бы определить гораздо более сложную функцию.

1.6 Обработка выбросов

Для обработки выбросов, как правило, можно использовать три стратегии. Во-первых, мы можем их отбросить:

```
# Загрузить библиотеку
import pandas as pd

# Создать фрейм данных
houses = pd.DataFrame()
houses['Цена'] = [534433, 392333, 293222, 4322032]
houses['Ванные'] = [2, 3.5, 2, 116]
houses['Кв_футы'] = [1500, 2500, 1500, 48000]
```

```
# Отфильтровать наблюдения
houses[houses['Ванные'] < 20]
```

	Цена	Ванные	Кв_футы
0	534433	2.0	1500
1	392333	3.5	2500
2	293222	2.0	1500

Во-вторых, мы можем пометить их как выбросы и включить их в качестве признака:

```
# Загрузить библиотеку
import numpy as np

# Создать признак на основе булева условия
houses["Выброс"] = np.where(houses["Ванные"] < 20, 0, 1)

# Показать данные
houses
```

	Цена	Ванные	Кв_футы	Выброс
0	534433	2.0	1500	0
1	392333	3.5	2500	0
2	293222	2.0	1500	0
3	4322032	116.0	48000	1

Наконец, мы можем преобразовать признак, чтобы ослабить эффект выброса:

```
# Взять логарифм признака
houses["Логарифм_кв_футов"] = [np.log(x) for x in houses["Кв_футы"]]

# Показать данные
houses
```

	Цена	Ванные	Кв_футы	Выброс	Логарифм_кв_футов
0	534433	2.0	1500	0	7.313220
1	392333	3.5	2500	0	7.824046
2	293222	2.0	1500	0	7.313220
3	4322032	116.0	48000	1	10.778956

Если мы точно знаем, что выброс — это ошибка измерения или просто «мусорное» значение, его можно убрать. Особенно это помогает, когда выбросы небольшие по числу, но сильно искажают средние, дисперсию и другие статистики. Минус в том, что при маленьком наборе данных удаление даже нескольких точек может сильно повлиять на результаты, а если выбросы отражают редкие, но реальные события, мы можем потерять ценную информацию.

Иногда выбросы сами по себе важны. Например, супербольшая покупка может говорить о VIP-клиенте. В этом случае логично добавить отдельный бинарный признак «выброс/не

выброс». Так модель узнает, что эти точки особенные. Минус — размерность данных немного растет, и иногда модель может начать «зацикливаться» на редких случаях, что не всегда полезно.

Если выбросы слишком сильно искажают распределение, но удалять их нельзя, можно преобразовать сам признак. Например, взять логарифм дохода или масштабировать значения так, чтобы экстремальные точки не давили на модель. Это помогает алгоритмам, чувствительным к масштабу и распределению (линейная регрессия, градиентный спуск, PCA). Но важно правильно выбрать функцию преобразования: неверная формула может исказить данные и ухудшить интерпретируемость признака. К тому же эффект выбросов не всегда полностью нивелируется.

1.7 Дискретизация признаков

Иногда числовой признак слишком разрозненный или имеет широкий диапазон, и модель плохо справляется с его обработкой. В таких случаях полезно разбить значения на **дискретные категории**, или «корзины». Это упрощает данные, делает их более интерпретируемыми и иногда помогает алгоритмам лучше уловить зависимости.

Современные методы позволяют автоматизировать процесс и находить более «умные» границы для группировки данных. Рассмотрим два подхода.

Первый подход — `KBinsDiscretizer`**.** Этот инструмент из библиотеки scikit-learn автоматически преобразует числовой признак в категории. Он поддерживает разные стратегии:

- `uniform` — равные интервалы
- `quantile` — корзины по квантилям (равные по числу объектов)
- `kmeans` — кластеры по алгоритму k-средних

Такой метод удобен, если мы хотим быстро и автоматически получить интерпретируемые категории без ручной настройки порогов.

```
# Загрузить библиотеки
import numpy as np
from sklearn.preprocessing import KBinsDiscretizer

# Создать признак
age = np.array([[6],
                [12],
                [20],
                [36],
                [65]])

# Создать дискретизатор: 3 корзины по квантилям
kbd = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='quantile')

# Преобразовать признак
kbd.fit_transform(age)
```

Результат:

```
array([[0.],
       [0.],
       [1.],
       [1.],
       [2.]])
```

Здесь значения автоматически разбились на три группы по квантилям. Такой подход сохраняет относительные различия и делает группы примерно равными по размеру.

Второй подход — Optimal Binning. Этот метод подбирает интервалы так, чтобы они максимально хорошо отражали зависимость признака от целевой переменной. Например, он может использовать критерии на основе информации (Information Value, χ^2 и др.). Optimal Binning широко применяется, где важно построить простые и интерпретируемые группы с высокой объяснительной силой.

```
# Установка: pip install optbinning
import numpy as np
import pandas as pd
from optbinning import OptimalBinning

# Признак (возраст) и целевая переменная (например, 1 — купил товар, 0 — нет)
age = np.array([6, 12, 20, 36, 65])
target = np.array([0, 0, 1, 1, 1])

# Создать оптимальный биннинг
optb = OptimalBinning(name="age", dtype="numerical", solver="cp")
optb.fit(age, target)

# Получить категории
optb.transform(age, metric="bins")
```

Результат (пример):

```
array([0, 0, 1, 2, 2])
```

Алгоритм сам определил оптимальные границы, разделив возраст на три категории, которые лучше всего объясняют целевую переменную.

Дискретизация признаков может быть хорошей стратегией, когда числовые данные слишком разрозненные, имеют широкий разброс или содержат шум, мешающий модели корректно выявлять закономерности. Она упрощает данные, делает их более интерпретируемыми и позволяет алгоритмам лучше справляться с предсказанием категориальных зависимостей.

Например, если у нас есть признак «возраст клиента», который варьируется от 18 до 80 лет, модель может плохо различать влияние отдельных чисел. Разбив возраст на категории «молодые», «средний возраст» и «пожилые», мы упрощаем задачу для модели и одновременно делаем результаты более понятными для анализа.

1.8 Удаление наблюдений с пропущенными значениями

Удаление наблюдений с пропущенными значениями выполняется легко с помощью умной конструкции в библиотеке NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу признаков
features = np.array([[1.1, 11.1],
                     [2.2, 22.2],
                     [3.3, 33.3],
                     [4.4, 44.4],
                     [np.nan, 55]])

# Оставить только те наблюдения, которые не (помечены ~) пропущены
features[~np.isnan(features).any(axis=1)]
```

Результат:

```
array([[ 1.1, 11.1],
       [ 2.2, 22.2],
       [ 3.3, 33.3],
       [ 4.4, 44.4]])
```

В качестве альтернативы можно удалить наблюдения, содержащие пропущенные значения, с помощью библиотеки pandas:

```
# Загрузить библиотеку
import pandas as pd

# Загрузить данные
dataframe = pd.DataFrame(features, columns=["признак_1", "признак_2"])

# Удалить наблюдения с отсутствующими значениями
dataframe.dropna()
```

	признак_1	признак_2
0	1.1	11.1
1	2.2	22.2
2	3.3	33.3
3	4.4	44.4

Большинство алгоритмов не могут обрабатывать пропущенные значения в массивах целей и признаков. По этой причине мы не можем игнорировать пропущенные значения в наших данных и должны решить эту проблему во время предобработки.

> Ряд современных моделей (например, градиентные бустинги) умеют обрабатывать пропущенные значения напрямую, но в классических алгоритмах (логистическая регрессия, SVM, kNN) это невозможно.

Самым простым решением является удаление каждого наблюдения, содержащего одно или несколько пропущенных значений. Эта задача быстро и легко выполняется с помощью библиотек NumPy или pandas.

Вместе с тем мы должны всячески воздерживаться от удаления наблюдений с отсутствующими значениями. Их удаление является крайним средством, поскольку наш алгоритм теряет доступ к полезной информации, содержащейся в непропущенных значениях наблюдения.

Не менее важно и то, что в зависимости от причины появления пропущенных значений удаление наблюдений может привести к смещению наших данных. Существует три типа пропущенных данных:

1. Пропущены совершенно случайно (Missing Completely At Random, MCAR)

Пропуски происходят полностью случайно, и их появление никак не связано ни с другими признаками, ни с самим значением. Например, при сборе данных о температуре воздуха термометр иногда случайно выходит из строя, и несколько измерений пропадают.

2. Пропущены случайно (Missing At Random, MAR)

Пропуски зависят от других известных признаков, но не от самого пропущенного значения. Например, в интернет-магазине пользователи с более высокой частотой покупок реже оставляют отзывы о продуктах, при этом мы знаем историю их покупок. Здесь пропуски можно объяснить другими имеющимися признаками.

3. Пропущены не случайно (Missing Not At Random, MNAR)

Пропуски зависят от самого значения или от информации, которой нет в данных. Например, в анкете о доходах богатые респонденты чаще скрывают свои заработки — если у нас нет других признаков, которые могли бы объяснить их поведение.

Иногда допустимо удалять наблюдения, если они пропущены совершенно случайно (MCAR) или просто случайно (MAR). Однако если значение пропущено не случайно (MNAR), то факт, что значение пропущено, сам по себе является информацией. Удаление наблюдений MNAR может привнести в наши данные смещение, потому что мы удаляем наблюдения, порожденные некоторым ненаблюдаемым систематическим эффектом.

Удаление — простая, но крайняя мера. Понимание механизма пропусков помогает избежать смещения.

1.9 Заполнение пропущенных значений

Если объем данных небольшой, то предсказать пропущенные значения с помощью k ближайших соседей (KNN):

```
from sklearn.impute import KNNImputer
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

# Создать матрицу признаков
```

```
features, _ = make_blobs(n_samples=1000, n_features=2, random_state=1)

# Стандартизировать признаки
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)

# Заменить значение на пропущенное
true_value = standardized_features[0, 0]
standardized_features[0, 0] = np.nan

# Импутировать KNN
imputer = KNNImputer(n_neighbors=5)
features_knn_imputed = imputer.fit_transform(standardized_features)

print("Истинное значение:", true_value)
print("Заполненное значение:", features_knn_imputed[0, 0])
```

Результат:

```
Истинное значение: 0.8730186114
Заполненное значение: 1.09553327131
```

В качестве альтернативы мы можем использовать `SimpleImputer` библиотеки scikit-learn для заполнения пропущенных значений средними, медианными или наиболее частыми значениями признаков. Вместе с тем мы, как правило, получаем результаты хуже, чем с k ближайшими соседями:

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="mean")
features_mean_imputed = imputer.fit_transform(standardized_features)

print("Истинное значение:", true_value)
print("Заполненное значение:", features_mean_imputed[0, 0])
```

Результат:

```
Истинное значение: 0.8730186113995938
Заполненное значение: -3.058372724614996
```

Существуют две основные стратегии замены пропущенных данных подстановочными значениями, каждая из которых имеет свои сильные и слабые стороны.

Во-первых, для предсказания значений пропущенных данных мы можем использовать машинное обучение. Для этого признак с пропущенными значениями рассматривается как вектор целей и оставшееся подмножество признаков используется для предсказания пропущенных значений. Хотя для вычисления значений можно применять широкий спектр алгоритмов, популярным выбором является алгоритм k ближайших соседей (KNN). Алгоритм KNN будет более подробно рассмотрен в следующей лабораторной работе, но краткое его

объяснение заключается в том, что в данном алгоритме для предсказания пропущенного значения используется k ближайших наблюдений (в соответствии с некоторой метрикой расстояния). В нашем решении мы предсказали пропущенное значение, используя пять ближайших наблюдений.

Недостаток алгоритма KNN: для того чтобы знать, какие наблюдения наиболее близки к пропущенному значению, необходимо вычислить расстояние между пропущенным значением и каждым отдельным наблюдением. Это разумно в небольших наборах данных, но быстро становится проблематичным, если набор данных содержит миллионы наблюдений. Также стоит упомянуть, что **KNN чувствителен к масштабу признаков,** поэтому стандартизация обязательна.

Альтернативной и более масштабируемой стратегией является заполнение всех пропущенных значений некоторым средним значением. Например, в нашем решении мы использовали библиотеку scikit-learn для заполнения пропущенных значений средним значением признака. Заполненное значение часто не так близко к истинному значению, как при использовании KNN, но мы можем легко масштабировать заполнение на основе среднего для данных, содержащих миллионы наблюдений.

2 Работа с категориальными данными

В реальных наборах данных часто встречаются признаки, описывающие не количество, а качество объектов. Такая информация называется **категориальной** и отражает принадлежность наблюдения к определённой группе (например, пол, цвет или марка автомобиля).

При этом важно различать два основных типа категориальных признаков:

- **Номинальные:** категории не имеют естественного упорядочения.
  - красный, синий, зелёный;
  - банан, яблоко, клубника;
  - мужчина, женщина.
- **Порядковые (с порядком):** категории можно расположить по возрастанию или убыванию, хотя расстояния между ними условные.
  - низкий, средний, высокий;
  - согласен, нейтрален, не согласен;
  - младший, средний, старший.

Большинство алгоритмов машинного обучения напрямую работают только с числовыми данными. Поэтому строковые категории необходимо **преобразовать в числовой формат**. Простое присвоение чисел (`1, 2, 3`) может быть ошибочным, если категории номинальные: модель будет интерпретировать порядок и расстояние там, где их нет.

Именно поэтому для категориальных данных разработаны специальные методы кодирования, которые сохраняют их смысловую структуру. В следующих разделах мы рассмотрим основные техники:

- преобразование категориальных признаков в двоичные индикаторы (**one-hot encoding**);

- преобразование порядковых категорий с учётом их естественного порядка (**ordinal encoding**);
- современные подходы для работы с признаками с большим числом уникальных значений (таргет-энкодинг, частотное кодирование, хэширование).

Таким образом, корректная обработка категориальных признаков является ключевым шагом в подготовке данных: от выбранного способа кодирования напрямую зависит, насколько эффективно модель сможет использовать категориальную информацию.

2.1 Кодирование номинальных категориальных признаков

В данных часто встречаются признаки с **номинальными категориями**, то есть с классами, которые не имеют внутреннего порядка и не могут быть количественно упорядочены. Примеры таких признаков — любимый фрукт («яблоко», «груша», «банан»), город проживания или цвет автомобиля. Модели машинного обучения не умеют напрямую работать с текстовыми категориями, поэтому их необходимо преобразовать в числовой формат.

Для кодирования таких признаков удобно использовать **one-hot кодирование**, то есть преобразование в набор бинарных признаков, где каждая категория представлена отдельным столбцом. В Python это реализуется с помощью класса `OneHotEncoder` из библиотеки **scikit-learn**. Каждый объект получает единицу в столбце соответствующей категории, а остальные столбцы заполнены нулями. Такой подход сохраняет информацию о принадлежности к категории, не вводя искусственного порядка между значениями.

```
# Импортировать библиотеки
import numpy as np
from sklearn.preprocessing import OneHotEncoder, MultiLabelBinarizer

# Создать признак
feature = np.array([["Texas"],
                    ["California"],
                    ["Texas"],
                    ["Delaware"],
                    ["Texas"]])

# Создать кодировщик OneHotEncoder
one_hot = OneHotEncoder(sparse_output=False)

# Преобразовать признак в one-hot представление
encoded_feature = one_hot.fit_transform(feature)
```

```
array([[0, 0, 1],
       [1, 0, 0],
       [0, 0, 1],
       [0, 1, 0],
       [0, 0, 1]])
```

Для вывода классов можно воспользоваться атрибутом `classes_`:

```
# Взглянуть на классы признака
one_hot.classes_
```

```
array(['California', 'Delaware', 'Texas'], dtype='<U10')
```

Если требуется обратить кодирование с одним активным состоянием, то можно применить метод `inverse_transform`:

```
# Обратить кодирование с одним активным состоянием
one_hot.inverse_transform(one_hot.transform(feature))
```

```
array(['Texas', 'California', 'Texas', 'Delaware', 'Texas'], dtype='<U10')
```

Для преобразования признака в кодировку с одним активным состоянием можно даже использовать библиотеку pandas:

```
# Импортировать библиотеку
import pandas as pd

# Создать фиктивные переменные из признака
pd.get_dummies(feature[:,0])
```

	California	Delaware	Texas
0	0	0	1
1	1	0	0
2	0	0	1
3	0	1	0
4	0	0	1

Одной из полезных возможностей библиотеки scikit-learn является обработка ситуации, когда в каждом наблюдении перечисляется несколько классов:

```
# Создать мультиклассовый признак
multiclass_feature = [("Texas", "Florida"),
                      ("California", "Alabama"),
                      ("Texas", "Florida"),
                      ("Delaware", "Florida"),
                      ("Texas", "Alabama")]

# Создать мультиклассовый кодировщик, преобразующий признак
# в кодировку с одним активным состоянием
one_hot_multiclass = MultiLabelBinarizer()

# Кодировать мультиклассовый признак в кодировку с одним активным состоянием
one_hot_multiclass.fit_transform(multiclass_feature)
```

```
array([[0, 0, 0, 1, 1],
       [1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1],
       [0, 0, 1, 1, 0],
       [1, 0, 0, 0, 1]])
```

Опять-таки, классы можно увидеть с помощью атрибута `classes_`:

```
# Взглянуть на классы
one_hot_multiclass.classes_
```

```
array(['Alabama', 'California', 'Delware', 'Florida', 'Texas'], dtype=object)
```

Можно подумать, что правильная стратегия состоит в том, чтобы назначать каждому классу числовое значение (например, Texas = 1, California = 2). Однако, когда классы не имеют внутренней упорядоченности (например, Техас не "меньше" Калифорнии), числовые значения ошибочно создают порядок, которого нет.

Правильной стратегией является создание в исходном признаке бинарного признака для каждого класса. Это нередко называется **кодированием с одним активным состоянием.** Признаком в нашем решении был вектор, содержащий три класса (т. е. Техас, Калифорния и Делавэр). В кодировании с одним активным состоянием каждый класс становится одноэлементным признаком с единицами, когда класс появляется, и нулями в противном случае. Поскольку наш признак имел три класса, кодирование с одним активным состоянием вернуло три бинарных признака (по одному для каждого класса).

Используя кодирование с одним активным состоянием, можно фиксировать принадлежность наблюдения к классу, сохраняя при этом сведения о том, что в классе отсутствует какая-либо иерархия.

Наконец, стоит отметить, что после кодирования признака в кодировку с одним активным состоянием часто рекомендуется отбрасывать один из закодированных в результирующей матрице признаков, чтобы избежать линейной зависимости (это важно **только для линейных моделей**).

2.2 Кодирование порядковых категориальных признаков

Во многих наборах данных встречаются **порядковые признаки** — категориальные переменные, значения которых имеют естественный порядок. В отличие от **номинальных признаков** (например, «яблоко», «груша», «банан»), здесь важна не только принадлежность к классу, но и относительное расположение классов.

Классический пример — **шкала Лайкерта** в социологических опросах:

- полностью согласен
- согласен
- нейтрален
- не согласен
- категорически не согласен

Для таких признаков удобно использовать `OrdinalEncoder` из библиотеки scikit-learn. Он позволяет явно задать порядок категорий и преобразовать их в числовые значения:

```
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder

# Данные
df = pd.DataFrame({
    "оценка": ["низкая", "средняя", "высокая", "средняя", "низкая"]
})

# Определяем порядок категорий
categories = [["низкая", "средняя", "высокая"]]

# Кодируем
encoder = OrdinalEncoder(categories=categories)
df["оценка_encoded"] = encoder.fit_transform(df[["оценка"]])

print(df)
```

```
    оценка  оценка_encoded
0    низкая            0.0
1   средняя            1.0
2   высокая            2.0
3   средняя            1.0
4    низкая            0.0
```

Такой подход хорошо работает, если интервалы между категориями можно считать одинаковыми. В **линейных моделях** (линейная регрессия, логистическая регрессия, SVM) использование `OrdinalEncoder` может быть опасным. Модель начнёт воспринимать категориальные значения как числовую ось, что может ввести сильное искажение.

Пример: если классы кодируются как 0, 1, 2, то расстояние от 0→1 и 1→2 будет одинаковым, хотя в реальности оно может не иметь смысла.

Альтернативный подход

Когда интервалы между категориями неоднородны, а порядок не так важен, используют **target encoding**. Здесь каждая категория заменяется на статистику по целевой переменной (обычно среднее).

```
import category_encoders as ce

# Пример: зависимая переменная — успех/неуспех
df = pd.DataFrame({
    "оценка": ["низкая", "средняя", "высокая", "средняя", "низкая"],
    "успех": [0, 0, 1, 1, 0]
})

# Target encoding
```

```
encoder = ce.TargetEncoder(cols=["оценка"])
df["оценка_encoded"] = encoder.fit_transform(df["оценка"], df["успех"])

print(df)
```

```
    оценка  успех  оценка_encoded
0   низкая      0             0.0
1  средняя      0             0.5
2  высокая      1             1.0
3  средняя      1             0.5
4   низкая      0             0.0
```

2.3 Заполнение пропущенных значений классов

В реальных данных почти всегда встречаются **пропуски** в категориальных признаках: кто-то забыл ответить в анкете, произошёл сбой при сборе данных или информация просто отсутствует. Перед обучением модели такие пропуски нужно обработать, иначе алгоритм может выдавать ошибки или давать некорректные результаты.

Самый простой способ — заполнить пропуски **наиболее частым значением признака** или ввести отдельную категорию `"missing"`. Этот метод быстрый и надёжный, особенно если пропусков немного.

```
# Загрузить библиотеки
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer

# Создать данные с пропущенными значениями
data = pd.DataFrame({
   "city": ["Moscow", np.nan, "Berlin", "Paris", np.nan, "Paris"]
})

# Заполнение наиболее частым значением
imputer_freq = SimpleImputer(strategy="most_frequent")
data["city_filled_freq"] = imputer_freq.fit_transform(data[["city"]])

# Заполнение отдельной категорией "missing"
imputer_const = SimpleImputer(strategy="constant", fill_value="missing")
data["city_filled_const"] = imputer_const.fit_transform(data[["city"]])
```

Результат:

```
     city city_filled_freq city_filled_const
0  Moscow          Moscow          Moscow
1     NaN           Paris         missing
2  Berlin          Berlin          Berlin
3   Paris           Paris           Paris
```

```
4     NaN          Paris         missing
5   Paris          Paris         Paris
```

Заполнение пропусков таким способом просто и быстро, но оно не учитывает взаимосвязь между признаками. Если пропусков много или данные сложные, этот метод может вносить смещение.

Если нужно учесть взаимосвязи между признаками и повысить качество заполнения, используют **мультивариантную импутацию** с помощью `IterativeImputer`. Для категориальных признаков перед этим используют `OrdinalEncoder`, чтобы преобразовать строки в числа.

```
# Загрузить библиотеки
import pandas as pd
import numpy as np
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.preprocessing import OrdinalEncoder

# Создать данные
data = pd.DataFrame({
   "education": ["low", np.nan, "medium", "high", np.nan, "medium"],
   "age": [25, 30, 28, 40, 22, 35]
})

# Кодируем категории в числа
encoder = OrdinalEncoder()
data["education_encoded"] = encoder.fit_transform(data[["education"]])

# Итеративная импутация
imputer = IterativeImputer(max_iter=10, random_state=42)
data_imputed = imputer.fit_transform(data[["education_encoded", "age"]])

# Декодируем обратно
data["education_filled"] = encoder.inverse_transform(data_imputed[:, [0]])
```

Результат:

```
  education  age  education_encoded education_filled
0       low   25                1.0            low
1       NaN   30                NaN         medium
2    medium   28                  2.0          medium
3      high   40                0.0           high
4       NaN   22                NaN             low
5    medium   35                  2.0          medium
```

Итеративная импутация позволяет учитывать зависимость между признаками и получать более реалистичные значения пропусков. Недостатком является более высокая вычислительная стоимость и необходимость кодирования категорий перед импутацией.

3 Оценивание моделей

В этом разделе мы рассмотрим практические стратегии оценивания качества моделей машинного обучения. Может показаться нелогичным обсуждать оценку моделей до того, как мы детально изучили алгоритмы их создания, но для этого есть веские причины.

Почему оценивание — это первый шаг? Модель полезна настолько, насколько точны её предсказания в реальных условиях. Создать модель технически просто — достаточно нескольких строк кода. Но создать модель, которая действительно работает на новых данных и решает бизнес-задачи, — это совсем другая история. Без правильной оценки мы рискуем получить модель, которая отлично «запоминает» обучающие данные, но полностью проваливается на практике.

Понимание того, как правильно оценивать модели, позволит вам более осознанно подходить к выбору и настройке алгоритмов обучения, которые мы будем изучать в дальнейшем. Это фундамент, на котором строится вся практическая работа с машинным обучением.

3.1 Кросс-валидация моделей

Перекрестная проверка (cross-validation) — это фундаментальная техника оценивания моделей машинного обучения, которая позволяет надежно определить, насколько хорошо модель будет работать на новых данных. Основная идея заключается в том, чтобы многократно разделять данные на обучающую и тестовую части разными способами, обучать модель на каждой обучающей части и оценивать на соответствующей тестовой части, а затем усреднять полученные результаты. Такой подход дает гораздо более стабильную и реалистичную оценку производительности модели по сравнению с однократным разделением данных. Перекрестная проверка особенно важна на ранних этапах разработки модели, когда нужно сравнивать различные алгоритмы, выбирать гиперпараметры и понимать, стоит ли вообще продолжать работу с конкретным подходом.

```
# Загрузить библиотеки
from sklearn import datasets, metrics
from sklearn.model_selection import StratifiedKFold, cross_validate
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

# Загрузить набор данных рукописных цифр
digits = datasets.load_digits()
features = digits.data
target = digits.target

# Создать конвейер стандартизации и логистической регрессии
pipeline = make_pipeline(
    StandardScaler(),
    LogisticRegression(max_iter=1000)
)

# Создать стратифицированную 10-блочную перекрестную проверку
cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
```

```
# Выполнить перекрестную проверку, усреднив метрику точности
cv_results = cross_validate(
    pipeline,
    features,
    target,
    cv=cv,
    scoring="accuracy",
    return_train_score=False,
    n_jobs=-1
)

# Средняя точность модели
cv_results["test_score"].mean()
```

```
0.964931719428926
```

При первом рассмотрении оценивание обучающихся моделей может выглядеть прямолинейной задачей: натренировать модель, а затем вычислить, насколько хорошо она работает, используя некоторые показатели результативности (точность, квадратические ошибки и т. д.). Однако этот подход в корне ошибочен. Если мы тренируем модель, используя наши данные, а затем оцениваем, насколько хорошо она работает с этими же данными, то мы не достигаем желаемой цели. Наша цель состоит не в том, чтобы оценить, насколько хорошо модель работает с нашими тренировочными данными, а в том, насколько хорошо она работает с данными, которые она никогда не видела раньше (например, новый клиент, новое изображение). По этой причине наш метод оценивания должен помочь нам понять, насколько хорошо модели способны делать предсказания на основе данных, которые они никогда не видели раньше.

Одной из стратегий может быть откладывание среза данных для тестирования. Он называется **контрольным** (или **отложенным**) **набором**. В обычной проверке наши наблюдения (признаки и цели) разбиваются на два набора, традиционно называемые **тренировочным набором** и **тестовым набором**. Мы берем тестовый набор и откладываем его в сторону, делая вид, что никогда его раньше не видели. Затем мы тренируем нашу модель, применяя наш тренировочный набор с использованием признаков и вектора целей, чтобы научить модель тому, как делать наилучшее предсказание. Наконец, мы симулируем, что никогда раньше не видели внешних данных, и оцениваем то, как наша модель, натренированная на нашем тренировочном наборе, работает на нашем тестовом наборе. Однако этот проверочный подход имеет два основных недостатка. Во-первых, результативность модели может сильно зависеть от того, какое количество наблюдений было выбрано для тестового набора. Во-вторых, модель не тренируется с использованием всех имеющихся данных и не оценивается по всем имеющимся данным.

Более оптимальная стратегия, которая преодолевает эти недостатки, называется **k-блочной перекрестной проверкой** (k-fold cross-validation, KFCV). В k-блочной перекрестной проверке мы разделяем данные на k частей, называемых "блоками". Модель обучается с помощью k - 1 блоков, объединенных в один тренировочный набор, и затем последний блок используется в качестве тестового набора. Мы повторяем это k раз, на очередном шаге в качестве тестового

набора используя другой блок. Затем результативность модели для каждой из k итераций усредняется для получения общей меры.

В нашем решении мы провели k-блочную перекрестную проверку с использованием 10 блоков и вывели оценки в `cv_results`:

```
# Взглянуть на оценки для всех 10 блоков
cv_results
```

```
array([0.97222222, 0.97777778, 0.95555556, 0.95      , 0.95555556,
       0.98333333, 0.97777778, 0.96648045, 0.96089385, 0.94972067])
```

Когда мы используем k-блочную перекрестную проверку, следует учесть несколько важных моментов. Во-первых, k-блочная перекрестная проверка исходит из того, что каждое наблюдение было создано независимо от другого (т. е. данные являются **независимыми одинаково распределенными**). Если данные являются независимыми одинаково распределенными, то при назначении блоков рекомендуется наблюдения перетасовывать. В библиотеке scikit-learn можно установить `shuffle=True` для выполнения перетасовки.

Во-вторых, когда мы используем k-блочную перекрестную проверку для оценивания классификатора, часто полезно иметь блоки, содержащие примерно одинаковый процент наблюдений из каждого отдельного целевого класса. Такая проверка называется **стратифицированной k-блочной**. Например, если бы наш вектор целей содержал пол, и 80% наблюдений были бы мужского пола, то каждый блок содержал бы 80% наблюдений с мужским полом и 20% наблюдений с женским полом. В библиотеке scikit-learn можно проводить стратифицированную k-блочную перекрестную проверку, поменяв класс `KFold` на класс `StratifiedKFold`.

Наконец, при использовании перекрестно-проверочных наборов или перекрестной проверки важно предварительно обработать данные на основе тренировочного набора, а затем применить эти преобразования к обоим наборам: тренировочному и тестовому. Например, когда мы выполняем подгонку (с помощью метода `fit`) нашего объекта стандартизации, `standardizer`, мы вычисляем среднее и дисперсию только тренировочного набора. Затем мы применяем это преобразование (с помощью метода `transform`) и к тренировочному, и к тестовому наборам:

```
from sklearn.model_selection import train_test_split

# Разделить данные
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1
)

# Подогнать стандартизатор только на тренировочном наборе
standardizer = StandardScaler()
standardizer.fit(features_train)

# Применить к обоим наборам
```

```
features_train_std = standardizer.transform(features_train)
features_test_std = standardizer.transform(features_test)
```

Здесь мы делаем вид, будто тестовый набор содержит неизвестные данные. Если мы выполним подгонку обоих наших препроцессоров, используя наблюдения из тренировочного и тестового наборов, то часть информации из тестового набора просочится в наш тренировочный набор. Это правило применимо для любого шага предобработки, например для отбора признаков.

Пакет `pipeline` библиотеки scikit-learn упрощает задачу при использовании методов перекрестной проверки. Сначала мы создаем пайплайн, который предварительно обрабатывает данные (например, `standardizer`), а затем тренирует модель (логистическая регрессия, `logit`):

```
# Создать конвейер
pipeline = make_pipeline(standardizer, logit)
```

Использование `Pipeline` упрощает этот процесс, автоматически применяя предобработку и обучение модели внутри перекрестной проверки:

```
cv_results = cross_validate(
    pipeline,
    features,
    target,
    cv=cv,
    scoring="accuracy",
    n_jobs=-1
)
```

Параметр `cv` задаёт стратегию разбиения (KFold, StratifiedKFold, Leave-One-Out и др.), `scoring` — метрику качества, а `n_jobs=-1` позволяет использовать все ядра CPU.

3.2 Метрики регрессионных моделей

Представьте, что вы построили модель машинного обучения, которая предсказывает цены на квартиры. Модель выдала свои прогнозы, но как понять, насколько хорошо она работает? Именно для этого и нужны метрики качества. Метрики — это числовые показатели, которые помогают измерить точность предсказаний модели.
На практике метрики решают несколько важных задач. Во-первых, они позволяют понять, насколько сильно модель ошибается в своих предсказаниях. Во-вторых, с их помощью можно сравнить несколько разных моделей и выбрать лучшую. В-третьих, метрики помогают отслеживать улучшение модели в процессе настройки параметров. Без метрик мы работали бы вслепую, не понимая, стала ли модель лучше после наших изменений или наоборот ухудшилась.
Разные метрики показывают разные аспекты качества модели. Некоторые больше реагируют на большие ошибки, другие одинаково учитывают все отклонения. Поэтому важно понимать особенности каждой метрики и выбирать подходящую для конкретной задачи.

Теперь давайте посмотрим, как все эти метрики работают на практике. В коде ниже мы создадим синтетический датасет, обучим на нём линейную регрессию и

среднеквадратическую ошибку (mean squared error, MSE):

```
# Загрузить библиотеки
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_predict
from sklearn.linear_model import LinearRegression
from sklearn.datasets import make_regression
from sklearn.metrics import mean_squared_error

# Сгенерировать матрицу признаков и вектор целей
features, target = make_regression(
    n_samples=100,
    n_features=3,
    n_informative=3,
    n_targets=1,
    noise=50,
    coef=False,
    random_state=1
)

# Создать объект линейной регрессии
ols = LinearRegression()

predictions = cross_val_predict(ols, features, target, cv=3)
mse = mean_squared_error(target, predictions)
```

```
array([-1718.22817783, -3103.4124284 , -1377.17858823])
```

Ещё одним распространённым метрическим показателем регрессии является коэффициент детерминации R^2:

```
# Перекрёстно проверить линейную регрессию,
# используя показатель R-квадрат
cross_val_score(ols, features, target, scoring="r2")
```

```
array([0.87804558, 0.76395862, 0.89154377])
```

MSE является одним из наиболее распространённых оценочных показателей для регрессионных моделей. Формально MSE — это среднее значение квадратов ошибок предсказаний относительно истинных значений. Это мера суммарной квадратичной ошибки между предсказанными и истинными значениями. Чем выше значение MSE, тем больше общая квадратичная ошибка и тем хуже модель.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

где:

- n - общее количество примеров (наблюдений) в наборе данных.
- y_i - истинное (реальное) значение для i-го примера.
- $\hat{y}_i$ - предсказанное моделью значение для i-го примера.

Возведение ошибок в квадрат имеет ряд математических преимуществ (в частности, делает все вклады положительными), но часто неочевидное следствие состоит в том, что несколько крупных ошибок штрафуются сильнее, чем множество мелких, даже если суммарная абсолютная ошибка одинакова.

Например, представьте две модели, A и B, каждая с двумя наблюдениями:

- модель A имеет ошибки 0 и 10, следовательно, MSE = $0^2 + 10^2 = 100$
- модель B имеет две ошибки по 5, следовательно, MSE = $5^2 + 5^2 = 50$

Обе модели имеют одинаковую суммарную абсолютную ошибку 10, однако MSE посчитает модель A (100) хуже, чем модель B (50). На практике это последствие редко является проблемой (и может быть теоретически полезным), и MSE хорошо работает как оценочная метрика.

Важное замечание: по умолчанию в библиотеке scikit-learn аргументы параметра scoring настроены так, что большее значение метрики считается лучшим, чем меньшее. Это не относится к MSE, где большее значение означает худшую модель. По этой причине scikit-learn использует отрицательный показатель MSE через аргумент "neg_mean_squared_error".

Среднеквадратическая ошибка (root mean squared error, RMSE) рассчитывается просто: берём корень из MSE. Формула выглядит следующим образом:

$$\mathrm{RMSE} = \sqrt{\mathrm{MSE}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Главное преимущество RMSE перед MSE заключается в том, что эта метрика возвращает нас к исходным единицам измерения. Если вы предсказываете цены в рублях, RMSE тоже будет в рублях, а не в рублях в квадрате. Это делает метрику намного понятнее: вы сразу видите типичную величину ошибки вашей модели.
RMSE стоит использовать, когда вам нужна интерпретируемая метрика в тех же единицах, что и целевая переменная. Она хорошо подходит для оценки типичной величины ошибки и, как и MSE, сильно штрафует большие отклонения. Однако чувствительность к выбросам остаётся проблемой: несколько аномально больших ошибок могут существенно исказить общую картину качества модели.

Средняя абсолютная ошибка (mean absolute error, MAE) вычисляется по формуле:

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Здесь мы просто берём модуль разницы между реальным и предсказанным значением, а затем усредняем.
В отличие от MSE и RMSE, MAE не возводит ошибки в квадрат, а просто берёт их по модулю. Это делает метрику более робастной к выбросам. Ошибка в 10 единиц добавит к MAE ровно

10, а не 100, как было бы в случае с MSE. Все ошибки влияют на метрику линейно, независимо от их величины.
MAE особенно полезна, когда в ваших данных есть выбросы, которые не должны чрезмерно влиять на оценку качества модели. Также её стоит применять, когда для вас все ошибки одинаково важны, независимо от их размера. Метрика легко интерпретируется: она показывает среднюю величину абсолютной ошибки в тех же единицах, что и целевая переменная. Если MAE равна 5000 рублей при предсказании цен на квартиры, это значит, что в среднем модель ошибается на 5000 рублей в ту или иную сторону.

Коэффициент детерминации R² рассчитывается по формуле:

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}i)^2}{\sum i = 1^n (y_i - \bar{y})^2}$$

где:

- SS_{res} - Сумма квадратов остатков (residuals), которая показывает, насколько предсказания модели ŷᵢ отклоняются от реальных значений yᵢ.
- SS_{tot} - Общая сумма квадратов (total), которая показывает, насколько реальные значения yᵢ отклоняются от их среднего значения ȳ.

Эта метрика показывает, какую долю дисперсии целевой переменной объясняет ваша модель. Значения R² лежат в диапазоне от 0 до 1, хотя теоретически могут быть и отрицательными, если модель работает хуже, чем простое предсказание среднего значения. Значение R² = 0.85 означает, что модель объясняет 85% вариативности в данных, а оставшиеся 15% обусловлены факторами, которые модель не учитывает.
R² удобно использовать для оценки общего качества модели и сравнения разных моделей на одних и тех же данных. Поскольку метрика нормализована и не зависит от масштаба данных, её легко интерпретировать. Однако есть важный недостаток: R² всегда увеличивается при добавлении новых признаков в модель, даже если эти признаки совершенно бесполезны и не улучшают предсказания на новых данных. Это может привести к переобучению.

4 Отбор модели

В машинном обучении мы настраиваем модель так, чтобы она хорошо предсказывала целевую переменную по признакам. Для этого алгоритм подбирает параметры модели, минимизируя функцию потерь (например, MSE в регрессии).

Почему нужен отбор модели:

- У разных алгоритмов и их настроек разная обобщающая способность. То, что отлично работает на одном наборе данных, может переобучаться или недообучаться на другом.
- Мы хотим найти конфигурацию, которая показывает наилучшее качество не только на обучающей выборке, но и в перекрестной проверке, то есть устойчиво обобщает.
- Отбор модели помогает системно сравнить варианты и избежать субъективных решений.

Гиперпараметры — это внешние настройки алгоритма, которые не обучаются из данных, а задаются заранее. Примеры: сила регуляризации alpha у Ridge/Lasso, степень полинома в PolynomialFeatures. В отличие от параметров модели (веса линейной регрессии), гиперпараметры не оцениваются градиентными методами внутри обучения, а подбираются снаружи, обычно с помощью перекрестной проверки.

Отбор модели — это процесс выбора наилучшей комбинации: алгоритм + значения его гиперпараметров. Например, если мы сравниваем Ridge и Lasso и для каждого перебираем 10 значений alpha, общее число кандидатов — 20. С помощью процедур вроде GridSearchCV или RandomizedSearchCV мы оцениваем качество каждого кандидата на кросс-валидации и выбираем лучший.

4.1 Отбор наилучших моделей с помощью исчерпывающего поиска

Поскольку многие алгоритмы чувствительны к настройке гиперпараметров, необходимо провести поиск по заранее определённому диапазону их возможных значений. Такой подход позволяет систематически перебрать комбинации гиперпараметров и найти оптимальные настройки.
Для этого в библиотеке **scikit-learn** предусмотрен класс `GridSearchCV`**:**

```
# Загрузить библиотеки
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект логистической регрессии
logistic = linear_model.LogisticRegression(solver='saga')

# Создать диапазон вариантов значений штрафного гиперпараметра
penalty = ['l1', 'l2']

# Создать диапазон вариантов значений регуляризационного гиперпараметра
C = np.logspace(0, 4, 10)

# Создать словарь вариантов гиперпараметров
hyperparameters = dict(C=C, penalty=penalty)

# Создать объект решеточного поиска
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, verbose=0)

# Выполнить подгонку объекта решеточного поиска
best_model = gridsearch.fit(features, target)
```

Класс `GridSearchCV` реализует отбор модели с использованием перекрестной проверки. В частности, пользователь определяет наборы возможных значений для одного или нескольких гиперпараметров, а затем объект класса `GridSearchCV` тренирует модель, используя каждое значение и/или комбинацию значений. Модель с лучшей оценкой результативности отбирается как наилучшая. Например, в нашем решении в качестве обучающегося алгоритма

мы использовали логистическую регрессию, содержащую два гиперпараметра: C и регуляризационный штраф. Не переживайте, если не знаете, что такое C и регуляризация; мы рассмотрим их дальше. Просто для начала уясните, что гиперпараметры C и регуляризационный штраф могут принимать диапазон значений, которые должны быть указаны до начала тренировки. Для гиперпараметра C мы определяем 10 возможных значений:

```
np.logspace(0, 4, 10)
```

```
array([1.00000000e+00, 2.78255940e+00, 7.74263683e+00, 2.15443469e+01,
       5.99484250e+01, 1.66810054e+02, 4.64158883e+02, 1.29154967e+03,
       3.59381366e+03, 1.00000000e+04])
```

И точно так же мы определяем два возможных значения регуляризационного штрафа: `['l1', 'l2']`. Для каждой комбинации значений гиперпараметров C и регуляризационного штрафа мы тренируем модель и оцениваем ее с помощью k-блочной перекрестной проверки. В нашем решении мы имели 10 возможных значений C, 2 возможных значения регуляризационного штрафа и 5 блоков. Они создали 10 × 2 × 5 = 100 вариантов моделей, из которых была выбрана лучшая.

После завершения работы объекта `GridSearchCV` можем увидеть гиперпараметры лучшей модели:

```
# Взглянуть на наилучшие гиперпараметры
print('Лучший штраф:', best_model.best_estimator_.get_params()['penalty'])
print('Лучший C:', best_model.best_estimator_.get_params()['C'])
```

```
Лучший штраф: l1
Лучший C: 7.742636826811269
```

По умолчанию после определения наилучших гиперпараметров объект `GridSearchCV` может переобучить модель, используя наилучшие гиперпараметры, на всем наборе данных (вместо откладывания блока для перекрестной проверки). Эту модель можно использовать для предсказания значений, как и любую другую модель библиотеки scikit-learn:

```
# Предсказать вектор целей
best_model.predict(features)
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Стоит отметить один параметр `verbose` объекта `GridSearchCV`. В нем нет особой необходимости, однако он может служить обнадеживающей мерой во время длительных процессов поиска, чтобы получать уведомления о том, что поиск прогрессирует. Параметр `verbose` задает объем сообщений, выводимых во время поиска, при этом 0 обозначает отсутствие сообщений, а значения от 1 до 3 — вывод сообщений с большей детализацией.

4.2 Отбор наилучших моделей с помощью рандомизированного поиска

Хотя полный перебор гиперпараметров `GridSearchCV` позволяет найти оптимальную комбинацию параметров, он может оказаться слишком затратным по времени и вычислительным ресурсам, особенно если количество гиперпараметров велико или их диапазоны широкие. В таких случаях практичнее использовать более лёгкий метод, который не требует проверки всех возможных комбинаций, но при этом способен найти хорошие, а иногда и почти оптимальные решения.
Для этого в библиотеке **scikit-learn** существует класс `RandomizedSearchCV`**:**

```
# Загрузить библиотеки
from scipy.stats import loguniform
from sklearn import linear_model, datasets
from sklearn.model_selection import RandomizedSearchCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект логистической регрессии
logistic = linear_model.LogisticRegression()

# Создать диапазон вариантов значений штрафного гиперпараметра
penalty = ['l1', 'l2']

# Создать диапазон вариантов значений регуляризационного гиперпараметра
C = loguniform(1e-3, 1e3)

# Создать словарь вариантов гиперпараметров
hyperparameters = dict(C=C, penalty=penalty)

# Создать объект рандомизированного поиска
randomizedsearch = RandomizedSearchCV(
    logistic, hyperparameters, random_state=1, n_iter=100,
    cv=5, verbose=0, n_jobs=-1
)

# Выполнить подгонку объекта рандомизированного поиска
best_model = randomizedsearch.fit(features, target)
```

В случае с объектом класса `RandomizedSearchCV`, если мы зададим распределение, то библиотека scikit-learn будет случайным образом отбирать без возврата образцы значений гиперпараметров из этого распределения. В качестве примера общего принципа работы здесь мы случайным образом отберем 10 значений из равномерного распределения в диапазоне от 0.001 до 1000:

```
# Определить равномерное распределение между 0.001 и 1000,
# отобрать 10 значений
loguniform(1e-3, 1e3).rvs(10, random_state=1)
```

```
[0.018, 0.45, 0.12, 7.34, 1.23, 0.005, 45.6, 3.21, 0.89, 120.0]
```

В качестве альтернативы, если мы зададим список значений, таких как два значения гиперпараметра регуляризационного штрафа, `['l1', 'l2']`, объект класса `RandomizedSearchCV` будет выполнять случайный отбор с заменой из списка.

Так же как с объектом класса `GridSearchCV`, мы можем увидеть значения гиперпараметров наилучшей модели:

```
# Взглянуть на лучшие гиперпараметры
print('Лучший штраф:', best_model.best_estimator_.get_params()['penalty'])
print('Лучший C:', best_model.best_estimator_.get_params()['C'])
```

```
Лучший штраф: l1
Лучший C: 1.668088018810296
```

И так же как с `GridSearchCV`, после завершения поиска `RandomizedSearchCV` выполняет подгонку новой модели, используя наилучшие гиперпараметры, на всем наборе данных. Эту модель можно использовать, как и любую другую в scikit-learn; например, делать предсказания:

```
# Предсказать вектор целей
best_model.predict(features)
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2,
       2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Количество отобранных комбинаций гиперпараметров (т. е. количество вариантов натренированных моделей) задается установкой параметра `n_iter` (количества итераций).

5 Линейная регрессия

Линейная регрессия является одним из самых простых алгоритмов машинного обучения в нашем инструментарии. В этом разделе мы рассмотрим различные методы линейной регрессии (и некоторые расширения) для создания работающих предсказательных моделей.

5.1 Построение линейной регрессии

Линейная регрессия — это один из самых базовых и понятных инструментов машинного обучения. Она используется тогда, когда мы предполагаем, что между входным признаком (или набором признаков) и целевой переменной существует линейная зависимость. По сути, модель пытается провести «прямую линию» (или гиперплоскость при нескольких признаках), которая наилучшим образом описывает взаимосвязь между данными. Такая простая модель полезна не только для прогнозирования, но и для анализа того, как именно признаки влияют на результат.
Для реализации линейной регрессии удобно использовать класс `LinearRegression` из библиотеки **scikit-learn:**

```
# Загрузить библиотеки
from sklearn.linear_model import LinearRegression
from sklearn.datasets import fetch_california_housing

data = fetch_california_housing(as_frame=True)
features = data.data.iloc[:, :2]  # первые два признака
target = data.target

# Создать объект линейной регрессии
regression = LinearRegression()

# Выполнить подгонку линейной регрессии
model = regression.fit(features, target)
```

Линейная регрессия исходит из того, что связь между признаками и вектором целей является приблизительно линейной, т. е. эффект (также называемый коэффициентом, весом или параметром) признаков на вектор целей является постоянным.

В нашем решении для целей объяснения мы натренировали нашу модель, используя всего два признака. Это значит, что наша линейная модель будет:

$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$

где y — наша цель; x_1 — данные для одного признака; β_1 и β_2 — коэффициенты, идентифицированные путем подгонки модели; ε — ошибка.

После выполнения подгонки нашей модели мы можем взглянуть на значение каждого параметра. Например, β_0, так называемое смещение или пересечение, можно просмотреть с помощью атрибута `intercept_`:

```
# Взглянуть на точку пересечения
model.intercept_
```

```
-0.10189032759082606
```

И β_1, и β_2 можно показать с помощью `coef_`:

```
# Взглянуть на коэффициенты признаков
model.coef_
```

```
array([0.43169191, 0.01744134])
```

В нашем наборе данных целевым значением является средняя стоимость дома в районе MedianHouseValue в Калифорнии, в сотнях тысяч долларов . Поэтому цена первого дома в наборе данных:

```
# Первое значение в векторе целей
target[0]
```

```
4.5260
```

Используя метод `predict`, мы можем предсказать значение для этого дома:

```
# Предсказать целевое значение первого наблюдения, умноженное на 1000
model.predict(features)[0]*1000
```

```
4.2071262638211786
```

Наша модель сместилась всего на 0.32!
Основным преимуществом линейной регрессии является её интерпретируемость: коэффициенты модели показывают, как изменение признака на единицу влияет на целевую переменную. Например, первый признак нашего решения — это **MedInc**, медианный доход жителей района в тысячах долларов. Коэффициент этого признака нашей модели составил **0.43**. Это значит, что при увеличении **MedInc** на 1 тысячу долларов средняя стоимость дома увеличится примерно на **0.43 × 100,000 = 43,000 долларов**, так как целевая переменная `target` измеряется в сотнях тысяч долларов.

```
# Первый коэффициент, умноженный на 100_000
model.coef_[0]*100_000
```

```
43169.19075449539
```

5.2 Учёт взаимодействий между признаками

В реальных данных часто встречаются ситуации, когда влияние одного признака на целевую переменную меняется в зависимости от значения другого признака. Например, эффект дохода на стоимость жилья может быть различным в районах с разной плотностью населения. Если использовать только линейную комбинацию признаков, такие зависимости не

будут учтены, и модель может оказаться слишком грубой. Чтобы уловить эти эффекты, в модель вводят специальные взаимодействующие признаки.

Для автоматического создания таких взаимодействующих членов удобно использовать класс `PolynomialFeatures` из библиотеки **scikit-learn:**

```
# Загрузка библиотек
from sklearn.linear_model import LinearRegression
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import PolynomialFeatures

# Загрузить данные с двумя признаками
data = fetch_california_housing()
features = data.data[:, :2]   # MedInc и HouseAge
target = data.target

# Создать взаимодействующие признаки
interaction = PolynomialFeatures(degree=2, include_bias=False, interaction_only=True)
features_interaction = interaction.fit_transform(features)

# Обучить модель
regression = LinearRegression()
model = regression.fit(features_interaction, target)

# Посмотреть признаки первого наблюдения
print(features[0])
print(features_interaction[0])
```

```
[ 8.3252 41.    ]
[  8.3252  41.     341.3332]
```

Здесь в `features[0]` мы видим значения первых двух признаков (`MedInc` и `HouseAge`), а в `features_interaction[0]` дополнительно появляется произведение этих признаков.

Иногда полезно явно контролировать взаимодействия. Например, перемножим медианный доход и возраст дома:

```
import numpy as np

interaction_term = np.multiply(features[:, 0], features[:, 1])
print(interaction_term[0])
```

```
341.33320000000003
```

Вместо ручного добавления взаимодействий удобно встроить этот шаг в пайплайн с помощью `ColumnTransformer` + `FunctionTransformer`:

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import FunctionTransformer
from sklearn.pipeline import Pipeline

# Определяем функцию для взаимодействия
def interaction_medinc_houseage(X):
    return (X[:, 0] * X[:, 1]).reshape(-1, 1)

# Создаём трансформер
interaction_transformer = FunctionTransformer(interaction_medinc_houseage, validate=False)

# Указываем, что берём только первые два признака (MedInc, HouseAge)
preprocessor = ColumnTransformer(
    transformers=[
        ("original", "passthrough", [0, 1]),
        ("interaction", interaction_transformer, [0, 1])
    ]
)

# Строим модель
pipeline = Pipeline([
    ("features", preprocessor),
    ("regression", LinearRegression())
])

pipeline.fit(features, target)

# Проверим первые признаки после трансформации
print(preprocessor.fit_transform(features)[:1])
```

Здесь `ColumnTransformer` позволяет объединять исходные признаки и новое взаимодействие в единую матрицу, а `FunctionTransformer` задаёт пользовательскую функцию для генерации взаимодействия. Такой подход более гибкий, чем `PolynomialFeatures`, и хорошо работает в сложных пайплайнах.

5.3 Моделирование нелинейных зависимостей

Один из наиболее понятных способов смоделировать нелинейную связь — это построить **полиномиальную регрессию**, то есть расширить исходные данные полиномиальными признаками и обучить на них обычную линейную модель. Для этого в Python можно воспользоваться классом `PolynomialFeatures` из библиотеки **scikit-learn:**

```
# Загрузить библиотеки
from sklearn.datasets import fetch_california_housing
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
```

```
# Загрузить данные с одним признаком
data = fetch_california_housing(as_frame=True)
feature = data.data[["MedInc"]]
target = data.target

# Создать полиномиальные признаки x^2 и x^3
polynomial = PolynomialFeatures(degree=3, include_bias=False)
features_polynomial = polynomial.fit_transform(feature)

# Создать объект линейной регрессии
regression = LinearRegression()

# Выполнить подгонку линейной регрессии
model = regression.fit(features_polynomial, target)
```

До сих пор мы обсуждали моделирование только линейных связей. Примером линейной связи может служить количество этажей здания и его высота. В линейной регрессии мы исходим из того, что эффект количества этажей и высоты здания примерно постоянен, а это означает, что 20-этажное здание будет примерно в два раза выше, чем 10-этажное, которое будет примерно в два раза выше, чем 5-этажное здание. Однако многие представляющие интерес связи не являются строго линейными.

Нередко требуется смоделировать нелинейную связь — например, связь между количеством часов, которые учащийся тратит на учебу, и оценкой, которую он получает за контрольную работу. Интуитивно мы можем себе представить, что в оценках между учащимися, которые учились в течение одного часа, и учащимися, которые вообще не учились, есть большая разница. Вместе с тем существует гораздо меньшая разница в оценках между учащимся, который учился в течение 99 часов, и учащимся, который учился в течение 100 часов. Влияние одного часа учебы на итоговую оценку учащегося уменьшается по мере увеличения количества часов.

Полиномиальная регрессия — это расширение линейной регрессии, позволяющее моделировать нелинейные связи. Для того чтобы создать полиномиальную регрессию, следует конвертировать линейную функцию, которую мы использовали в разделе 5.1:

$y = \beta_o + \beta_1 x_1 + \varepsilon$

в полиномиальную функцию путем добавления полиномиальных признаков:

$y = \beta_o + \beta_1 x_1 + \beta_2 x_1^2 + \ldots + \beta^d x_1^d + \varepsilon$

где d — степень полинома. Как использовать линейную регрессию для нелинейного признака? Ответ заключается в том, что мы ничего не изменяем в том, как линейная регрессия подбирает модель, а только добавляем полиномиальные признаки. То есть, линейная регрессия не "знает", что x^2 является квадратичным преобразованием x. Она просто рассматривает его, как еще одну переменную.

Для того чтобы смоделировать нелинейные связи, мы можем создать новые признаки, которые возводят существующий признак, x, в некоторую степень: x^2, x^3 и т. д. Чем больше этих новых признаков мы добавим, тем более гибкой будет "линия", созданная нашей моделью. Для того чтобы более четко отразить суть, представьте, что мы хотим создать полином третьей степени. Для простоты мы сосредоточимся только на одном наблюдении (первом наблюдении в наборе данных) x_o:

```
# Взглянуть на первое наблюдение
feature.iloc[0, 0]
```

```
8.3252
```

Для того чтобы создать полиномиальный признак, мы возведем первое значение наблюдения во вторую степень — x^2:

```
# Взглянуть на первое наблюдение, возведенное во вторую степень, x^2
feature.iloc[0, 0]**2
```

```
69.30895504
```

Это будет нашим новым признаком. Затем мы также возведем значение первого наблюдения в третью степень — x^3:

```
# Взглянуть на первое наблюдение, возведенное в третью степень, x^3
feature.iloc[0, 0]**3
```

```
577.010912499
```

Включив все три признака (x, x^2 и x^3) в матрицу признаков, а затем выполнив линейную регрессию, мы провели полиномиальную регрессию:

```
# Взглянуть на значения первого наблюдения для x, x^2 и x^3
features_polynomial[0]
```

```
array([8.3252,  69.30895504, 577.0109125])
```

Полиномиальные признаки имеют два важных параметра. Во-первых, `degree` определяет максимальное число степеней для полиномиальных признаков. Например, `degree=3` будет генерировать x^2 и x^3. Наконец, по умолчанию объект `PolynomialFeatures` включает в себя признаки, содержащие одни единицы (называемые смещением). Мы можем удалить их, установив `include_bias=False`.

Более гибкий способ учёта нелинейных связей — использование **сплайнов**. В scikit-learn это реализовано через `SplineTransformer`. В отличие от полиномов, сплайны позволяют «гибко изгибать» модель только на отдельных участках признака, избегая чрезмерных колебаний на краях диапазона.

```
from sklearn.preprocessing import SplineTransformer
from sklearn.pipeline import make_pipeline

# Модель со сплайнами
spline_model = make_pipeline(
    SplineTransformer(degree=3, n_knots=5),
```

```
    LinearRegression()
)

spline_model.fit(X, y)
```

Здесь мы разбиваем диапазон `MedInc` на 5 узлов и аппроксимируем зависимость кусочными кубическими функциями. Это даёт более устойчивую и интерпретируемую модель, чем полиномиальная регрессия высокой степени.

5.4 Снижение дисперсии с помощью регуляризации

Линейная регрессия проста и удобна, но у неё есть серьёзный недостаток — она склонна к переобучению, особенно если признаков много или они сильно коррелируют между собой. В таких случаях модель может подстраиваться под шум в данных, что приводит к высокой дисперсии предсказаний: на обучающей выборке она показывает отличные результаты, а на новых данных резко теряет качество. Чтобы избежать этого, нужно контролировать сложность модели и ограничивать слишком «вольные» коэффициенты. Такой подход называется регуляризацией.

Для снижения дисперсии и повышения устойчивости модели используют алгоритмы, которые добавляют штраф за слишком большие веса. Классические примеры — **гребневая регрессия (Ridge)** и **лассо-регрессия (Lasso)**. Обе они реализованы в библиотеке **scikit-learn** и позволяют уменьшить разброс предсказаний, сохранив при этом интерпретируемость модели.

```
# Загрузить библиотеки
from sklearn.linear_model import Ridge
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler

# Загрузить данные
data = fetch_california_housing(as_frame=True)
features = data.data
target = data.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект гребневой регрессии со значением альфа
regression = Ridge(alpha=0.5)

# Выполнить подгонку линейной регрессии
model = regression.fit(features_standardized, target)
```

В обычной линейной регрессии цель модели проста: найти такие коэффициенты, чтобы разница между реальными значениями целевой переменной и предсказанными значениями

была как можно меньше. Эта разница измеряется через сумму квадратов ошибок (Residual Sum of Squares, RSS):

$$RSS = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Регуляризованные версии линейной регрессии работают немного иначе. Они тоже стараются минимизировать RSS, но к функции ошибки добавляют дополнительный штраф за слишком большие коэффициенты. Такой штраф ещё называют «сжимающим», потому что он буквально «сжимает» веса модели, делая её проще и устойчивее.

Есть два популярных подхода к регуляризации: **гребневая регрессия (Ridge)** и **лассо-регрессия (Lasso)**. Разница между ними только в виде штрафа. Ridge использует сумму квадратов коэффициентов, умноженную на гиперпараметр α:

$$RSS + \alpha \sum_{j=1}^{p} \beta_j^2$$

А Lasso — сумму модулей коэффициентов:

$$RSS + \alpha \sum_{j=1}^{p} |\hat{\beta}_j|$$

Зачем это нужно? Ridge обычно даёт более стабильные предсказания, особенно если признаков много и они сильно коррелируют. Lasso же имеет полезное свойство «обнулять» некоторые коэффициенты, что делает модель проще и понятнее. Если хочется объединить плюсы обоих подходов, используют **эластичную сеть (Elastic Net)**, которая комбинирует оба вида штрафа.

Ключевая роль здесь у параметра **α**. Чем он больше, тем сильнее мы наказываем большие коэффициенты и тем проще становится модель. Оптимальное значение **α** подбирается так же, как и другие гиперпараметры. В scikit-learn это делается через параметр `alpha`. А для удобного подбора библиотека предлагает класс `RidgeCV`, который автоматически ищет лучшее значение **α** на основе перекрёстной проверки.

Библиотека scikit-learn включает класс `RidgeCV`, реализующий метод, который позволяет отбирать идеальное значение для α:

```
# Загрузить библиотеку
from sklearn.linear_model import RidgeCV

# Создать объект гребневой регрессии с тремя значениями alpha
regr_cv = RidgeCV(alphas=[0.1, 1.0, 10.0])

# Выполнить подгонку линейной регрессии
model_cv = regr_cv.fit(features_standardized, target)

# Взглянуть на коэффициенты
model_cv.coef_
```

```
array([ 0.8293461 ,  0.11939823, -0.26422311,  0.30398067, -0.00427544,
       -0.03936068, -0.8937389 , -0.86433656])
```

Затем мы можем легко просмотреть значение α наилучшей модели:

```
# Взглянуть на alpha
model_cv.alpha_
```

```
10.0
```

И последнее замечание: поскольку в линейной регрессии значение коэффициентов частично определяется шкалой признака, а в регуляризованных моделях все коэффициенты суммируются вместе, перед тренировкой мы должны убедиться, что стандартизировали признак.

5.5 Уменьшение количества признаков с помощью лассо-регрессии

Когда признаков в данных слишком много, линейная регрессия может становиться перегруженной и плохо интерпретируемой. К тому же не все признаки одинаково полезны: часть из них почти не влияет на целевую переменную или дублирует информацию других. В таких случаях важно не только построить качественную модель, но и отобрать наиболее значимые факторы, чтобы упростить анализ и повысить устойчивость к переобучению.
Для этой цели отлично подходит **лассо-регрессия (Lasso)**. Её особенность в том, что за счёт использования L1-штрафа она может обнулять коэффициенты у незначимых признаков. В результате модель «отбирает» наиболее важные переменные и отбрасывает лишние. Это делает Lasso удобным инструментом одновременно для регуляризации и для автоматического отбора признаков. В библиотеке **scikit-learn** это реализовано в классе `Lasso`

```
# Загрузить библиотеки
from sklearn.linear_model import Lasso
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler

# Загрузить данные
data = fetch_california_housing(as_frame=True)
features = data.data
target = data.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект лассо-регрессии со значением alpha
regression = Lasso(alpha=0.01)
```

```
# Выполнить подгонку линейной регрессии
model = regression.fit(features_standardized, target)
```

Например, в нашем решении мы установили `alpha` на уровне 0.01 и видим, что многие коэффициенты равны 0. Иными словами, их соответствующие признаки в модели не используются:

```
# Взглянуть на коэффициенты
model.coef_
```

```
array([ 0.77722333,  0.12486709, -0.12940585,  0.16912537, -0.        ,
       -0.02944551, -0.79543737, -0.75899738])
```

Однако, если мы увеличим α до гораздо более высокого значения, мы увидим, что буквально ни один из признаков не используется:

```
# Создать лассо-регрессию с высоким alpha
regression_a10 = Lasso(alpha=1)
model_a10 = regression_a10.fit(features_standardized, target)
model_a10.coef_
```

```
array([ 0.,  0.,  0., -0., -0., -0., -0., -0.])
```

Практическая выгода от этого эффекта в следующем: он означает, что мы можем включить в нашу матрицу признаков 100 признаков, а затем путем настройки гиперпараметра α лассо-регрессии создать модель, которая использует только (например) 10 наиболее важных признаков. Это позволяет нам уменьшить дисперсию, улучшая интерпретируемость нашей модели (поскольку меньшее количество признаков легче объяснить).

5.6 Эластичная сеть (Elastic Net)

Эластичная сеть — это разновидность линейной регрессии с регуляризацией, которая объединяет лучшие стороны двух других методов: **гребневой регрессии (Ridge)** и **лассо-регрессии (Lasso)**. Если Ridge помогает бороться с переобучением за счёт сглаживания коэффициентов, а Lasso умеет автоматически «отбрасывать» незначимые признаки, то Elastic Net сочетает оба этих эффекта. Именно поэтому этот метод часто выбирают, когда нужно не просто сделать модель устойчивой, но и при этом сохранить интерпретируемость — то есть понять, какие признаки действительно важны.

В основе Elastic Net лежит та же линейная модель, что и у обычной регрессии, но к функции ошибки добавляются два штрафа: один по типу Ridge (за большие значения коэффициентов), и другой по типу Lasso (за сам факт наличия слишком большого количества признаков).

В эластичной сети минимизируется следующая функция ошибки:

$$RSS + \alpha_1 \sum_{j=1}^{p} |\beta_j| + \alpha_2 \sum_{j=1}^{p} \beta_j^2$$

или в более привычной форме:

$$RSS + \alpha \left(l_1 \sum_{j=1}^{p} |\beta_j| + (1 - l_1) \sum_{j=1}^{p} \beta_j^2 \right)$$

Эти штрафы регулируются двумя параметрами:

- **α** — общий уровень регуляризации,
- **l1_ratio** — доля L1-компоненты, то есть насколько сильно метод склоняется в сторону Lasso.

Если `l1_ratio=1`, то это чистое Lasso; если `l1_ratio=0`, то чистый Ridge; а при промежуточных значениях — компромисс между ними.

```
from sklearn.datasets import fetch_california_housing
from sklearn.linear_model import ElasticNetCV
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

# Загружаем данные
data = fetch_california_housing(as_frame=True)
X = data.data
y = data.target

# Пайплайн: стандартизация + эластичная сеть
model = make_pipeline(
    StandardScaler(),
    ElasticNetCV(
        l1_ratio=[0.1, 0.5, 0.9],
        alphas=[0.001, 0.01, 0.1, 1.0],
        cv=5,
        random_state=42
    )
)

model.fit(X, y)
elastic = model.named_steps["elasticnetcv"]

print("Оптимальное значение alpha:", elastic.alpha_)
print("Оптимальное значение l1_ratio:", elastic.l1_ratio_)
print("Коэффициенты модели:", elastic.coef_)
```

Результат:

```
Оптимальное значение alpha: 0.001
Оптимальное значение l1_ratio: 0.9
Коэффициенты модели: [ 0.82493291  0.11971443 -0.25304234  0.29313236 -0.00323944
-0.03850743 -0.88885652 -0.85894042]
```

После обучения модель сама подберёт оптимальные значения параметров.

Результат можно интерпретировать так: часть коэффициентов окажется обнулённой (как в Lasso), а часть просто уменьшится (как в Ridge). В итоге модель становится одновременно **устойчивой и интерпретируемой**.

Главное достоинство эластичной сети — это её **гибкость**. Она позволяет управлять балансом между «жёсткостью» регуляризации и отбором признаков. Если признаки сильно коррелируют между собой (что часто бывает в реальных данных), Elastic Net ведёт себя гораздо стабильнее, чем Lasso, который в таких ситуациях может случайно выбрать один признак и полностью проигнорировать другой, не менее важный. Кроме того, Elastic Net предотвращает переобучение и обычно улучшает способность модели обобщать новые данные.

С другой стороны, у метода есть и свои ограничения. Во-первых, он требует подбора двух гиперпараметров (`alpha` и `l1_ratio`), что делает процесс обучения немного сложнее. Во-вторых, результат зависит от масштабирования данных — без стандартизации признаки с большими значениями будут доминировать в функции потерь. Наконец, если данных очень мало, регуляризация может чрезмерно «загладить» модель и ухудшить точность.

На практике стоит **использовать** `ElasticNetCV` — эта версия автоматически подбирает параметры по кросс-валидации и избавляет от ручного подбора. Перед обучением необходимо **масштабировать признаки** с помощью `StandardScaler`, иначе регуляризация будет работать некорректно. Если вы подозреваете сильную корреляцию между признаками, Elastic Net почти всегда будет **более надёжным выбором**, чем Lasso.