# Обучение с учителем

# Оглавление

# Линейные модели

## 1.1 Введение. Виды машинного обучения

Поговорим о классификации различных подходов к машинному обучению.

Самым распространенным подходом является, конечно, **обучение с учителем**, или **supervise learning**. В таком случае у нас существует какое-то количество размеченных данных, данных, для которых определен корректный ответ и корректное предсказание. Мы пытаемся обучить нашу модель на этих данных и потом сделать предсказание на новых данных, которых мы еще не видели.

В таком случае мы обучаемся с учителем, и классическим примером задачи, которая была решена и решается до сих пор с помощью обучения с учителем и машинного обучения в целом, является **задача кредитного скоринга**. У нас существует какой-то банк и пул клиентов, для которых у нас есть информация, об этих клиентах, есть информация о том, например, отдали они кредит или не отдали. К нам приходит новый человек, у нас есть информация про него из какой-то анкеты, мы пытаемся определить, стоит ему дать кредит или нет. Если да, то на какую сумму. Это задача обучения с учителем, у нас есть новые данные, мы пытаемся сделать на них предсказание. Если мы просто пытаемся понять, стоит ему дать кредит или нет, то это **задача классификации**. Если мы пытаемся заодно понять, на какую сумму, то **задача регрессии**, потому что у нас ответ предсказания является вещественным числом. Область обучения с учителем и, в принципе, такие подходы наиболее распространены сейчас, и если вы слышите про то, что какая-то компания применяет машинное обучение в своей работе, то скорее всего это именно обучение с учителем. Эти модели и методы работают лучше всего, они лучше всего обучаются и чаще всего применяются.

Большая часть нейронных сетей как раз работает на каких-то размеченных данных. Тем не менее, существует еще один подход, которому не требуются размеченные данные, не требуется какая-то ручная разметка или даже автоматическая разметка, - это **обучение без учителя**, или **unsupervised learning**.

В таком случае у нас просто существуют какие-то данные, и мы пытаемся использовать зависимость внутри этих данных и делать какие-то выводы по нашим данным или как-то их преобразовать. Казалось бы, как мы можем использовать просто данные и что-то с ними делать, делать ли какие-то предсказания или выводы, если мы никак их не разметили? Однако как вы знаете, в этих данных существует какое-то количество реальных природных зависимостей, которые мы можем использовать. И например, классической задачей для обучения без учителя является **задача понижения размерности**. Например, у нас есть какой-то датасет, какое-то количество данных с большим количеством признаков. И наша модель хотела бы обучаться не на огромном количестве признаков, например, не на тысяче признаков, а только на ста, ста самых важных. Мы хотели

бы отбросить лишние признаки, сохраняя максимальную предиктивную способность, то есть так, чтобы мы могли потом использовать эти признаки в работе. И таким образом мы пытаемся понизить размерность наших данных, отбрасывая признаки, которые меньше всего влияют на смысл, на суть этих данных. Также мы можем не просто отбрасывать признаки, а приходить в какое-то новое пространство признаков, генерируя новые признаки или просто моделируя какое-то новое пространство.

Еще одной из задач по этому подходу в этой области является **задача кластеризации**. У нас просто есть какое-то большое количество данных, мы пытаемся найти в этих данных какие-то кластеры или группы похожих друг на друга объектов. В данном случае у нас тоже нет разметки, мы просто пытаемся группировать наши данные на какое-то количество кластеров.

Также огромное распространение в последнее время получает **задача моделирования статистического распределения наших данных**. Мы пытаемся построить модель, которая умеет моделировать распределение этих данных, чтобы потом, например, из этого распределения мы могли генерировать новые примеры, новые признаки, новые данные.

И одной из, опять же, самых встречающихся применений обучения без учителя в индустрии, в области, является **задача поиска аномалий**. Например, в нашем банке идет большое количество транзакций, и мы пытаемся анализировать эти транзакции и находить транзакции мошеннические. В данном случае у нас очень маленькое количество примеров аномальных транзакций, мошенничества, и возможно, их даже вообще нет. Мы пытаемся найти какие-то выбросы, какие-то аномальные транзакции, которые могут быть фродом, или мошенничеством. Точно то же самое можно применять для каких-то заводок и сложных установок, где у нас поломки случаются очень редко или даже никогда.

Ещё один метод, который немного отличается от того, что мы с вами только что обсуждали, - это **обучение с подкреплением**, или **reinforcement learning**. В этом случае у нас задача ставится немного по-другому. Существует определенная среда, или окружение, и агент, который в этой среде действует. То есть у нас заданы какие-то правила в нашем мире, какая-то физика в этом мире, и какой-то агент, какой-то актор пытается с этим миром взаимодействовать, получая вознаграждение или штрафы за свои действия. Таким образом, повторяя различные действия, наш агент может выучивать определенные политики, выучивать определенное поведение, так чтобы максимизировать, например, вознаграждение. Это очень мощная концепция, которая используется на данный момент достаточно мало где, однако в будущем скорее всего будет активно развиваться и применяться, о чем говорят многие исследователи.

**На данный момент большинство исследований, большинство статей на эту тему по данному подходу работают с каким-то игровыми средами**. Например, вы могли слышать про AlphaGo от компании DeepMind, который не так давно обыграл чемпионов мира по игре в Go. А также, эта же компания активно работает например, с играми Atari или с какими-то лабиринтами.

Но несмотря на то, что очень много исследований сосредоточено именно на игровых окружениях, также им находится применение в реальной жизни, например, с помощью обучения с подкреплением была **оптимизирована работа для центра Google** и сэкономлено электричество практически в два раза, просто за счет таких методов, за счет обучения с подкреплением.

Также активно обучение с подкреплением используется в **робототехнике**, потому что задача, когда у нас агент действует в окружении и получает вознаграждение, отлично ложится на историю, когда робот, например, учится ходить или летать, брать какие-то предметы, что-то рас-

познавать — в общем, взаимодействовать с реальным миром. Что интересно, в данном случае мы можем не только обучать нашего робота в реальном мире, но и предобучать его в какой-то виртуальной среде. И обучившись в виртуальной среде, наш робот может уже потом взаимодействовать с миром и дообучаться.

Также активно обучение с подкреплением используется, например, в **трейдинге**, когда боты какие-то могут совершать операции покупки или продажи, и существует четкое вознаграждение — эта прибыль, которая была получена или не получена в результате покупки или продажи каких-то активов. Мы можем обучать с помощью обучения с подкреплением каких-то ботов, которые действуют на рынках.

Существуют, конечно, какие-то градации, какие-то промежуточные подходы между ними. Например, существует **semi-supervised learning**, который говорит о том, что мы можем не просто использовать размеченные данные, но и в дополнение к размеченным данным можно использовать какое-то количество неразмеченных данных. Но тем не менее, в основном выделяют именно три категории, о которых мы поговорили выше.

## 1.2 Линейная регрессия

**Линейная регрессия решает задачу обучения с учителем**, а значит у нас есть какой-то размеченный набор данных на которые мы обучаемся и потом мы делаем предсказания на новых данных, которых мы еще не видели. В случае линейной регрессии мы предсказываем какую-то вещественную переменную и делаем это пытаясь моделировать **линейную зависимость** от наших признаков, то есть у нас есть какая-то переменная и она зависит, как мы ожидаем, от других признаков, и мы пытаемся зависимости найти.

Например, мы можем попытаться предсказать зарплату специалиста по его возрасту, опыту и, например, полу. Можно ожидать, что зарплата специалиста зависит от возраста и опыта и не зависит, например, от пола. Эту зависимость мы пытаемся моделировать.

Если у нас задача **парной регрессии**, то мы просто пытаемся построить зависимость одной переменной от другой, например, мы можем построить зависимость веса человека от его роста, и очевидно какая-то природная зависимость в этом действительно есть, мы можем построить такую прямую в нашем двумерном пространстве и потом использовать её для того, чтобы как-то предсказывать вес человека, которого мы еще не видели.

Однако, конечно в реальном мире, пространство данных является пространством гораздо большей размерности, и не нужно этого пугаться. А мы также строим линейные зависимости, то есть, например, какие-то плоскости или гиперплоскости, не нужно этого бояться.

Таким образом, что же делает линейная регрессия? У нас есть какая-то целевая переменная. Мы просто пытаемся разбросать веса, найти такие веса к разным признакам, которые лучше всего описывают наши данные.

$y = w_0 + w_1 x_1 + \cdots + w_k x_k = \langle w, x \rangle$, где $y$ - **целевая переменная**, $x$ - **признаки**, $w$ - **веса модели**.

Можно просто рассматривать это как взвешенную сумму различных признаков, и эти веса мы как раз в нашей модели и подбираем, а так же мы, например, потом можем использовать эти веса для того, чтобы понять, какие из наших признаков наиболее важны.

И обобщением линейной регрессии является **полиномиальная регрессия**, которая делает то же самое, только немного по-другому. Мы все также подбираем веса перед нашими признаками, однако, наши признаки могут быть какими-то квадратными, например, мы можем использовать не только исходные признаки, но и квадраты этих признаков или полиномы.

## 1.3 Функционал качества и градиентный спуск

В случае линейной регрессии мы решаем задачу обучения с учителем. У нас есть размеченные данные, мы на них обучаемся, пытаемся их хорошо как-то описать и потом делаем предсказание на новых данных. И предсказываем действительное число, как вы уже знаете. Что же такое «хорошо описывать данные»? Как нам хорошо описать тренировочную выборку? Для того, чтобы нам обучаться, нам **нужна функция ошибки, чтобы ее оптимизировать**.

В качестве функции ошибки, например, мы можем использовать **Mean Absolute Error**, или **среднюю абсолютную ошибку**, которая показывает на то, как далеко наши средние предсказания лежат от корректных ответов.

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

где $N$ - размер выборки, $y_i$ - корректный ответ, $\hat{y}_i$ - наше предсказание.

Это достаточно **логичная и хорошо интерпретируемая функция потерь, однако она не дифференцируемая**, поэтому ее нельзя, например, использовать в случае градиентного спуска или методов градиентной оптимизации в общем. Поэтому обычно используются более сложные и немного другие функции.

Например, **Mean Squared Error**, то есть **средняя квадратичная** ошибка, которая делает то же самое, только усредняет квадраты расстояний наших предсказаний от реальных ответов.

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

где $N$ - размер выборки, $y_i$ - корректный ответ, $\hat{y}_i$ - наше предсказание.

Ее уже можно оптимизировать, ее можно дифференцировать, и она как раз **используется очень часто в методах градиентного спуска**.

Однако важно заметить, что в случае линейной регрессии с такой функцией потерь мы можем решать задачу и аналитически, не обязательно использовать градиентный спуск. Но в пространствах большой размерности приходится вращать сложными матрицами, и не всегда это работает, иногда легче использовать действительно градиентные методы, которые используются практически всегда. О них и поговорим.

Итак, можно представить нашу функцию потерь, как какую-то поверхность, по которой мы хотим спуститься. То есть мы хотим минимизировать нашу функцию, найти ее минимум. Именно там наша модель и будет работать лучше всего. Как же это сделать? Как вы знаете, **градиент показывает направление наискорейшего роста функции, а антиградиент, что логично, показывает направление наискорейшего уменьшения функции**. Именно его и можно использовать. Например, мы можем взять какой-то набор параметров, какую-то точку в нашей

модели и посмотреть вектор антиградиента и по нему спуститься, по нему обновить параметры нашей модели и делать так, пока мы куда-то не сойдемся, или пока качество нашей модели нас не станет устраивать. Именно так и работает градиентный спуск. Мы последовательно обновляем параметры нашей модели, веса, например, нашей линейной регрессии в соответствии с вектором антиградиента.

Существует огромное количество модификаций градиентного спуска: мы можем менять скорость нашего шага, мы можем брать какие-то подвыборки объектов. Например, существует стохастический градиентный спуск, который использует не всю выборку для того, чтобы делать шаг по антиградиенту, а только один объект или **mini-batch** - градиентный спуск, который использует какой-то набор объектов. Огромное количество модификаций, и практически все методы оптимизации, которые используются как в линейной регрессии, так и, например, в нейросетях — это какие-то разновидности градиентного спуска.

## 1.4 Логистическая регрессия

В случае классификации мы пытаемся предсказать какой-то дискретный ответ. Это задача обучения с учителем, мы обучаемся на различных данных, делаем предсказание новых данных, и предсказываем дискретный ответ, то есть 0 либо 1 в случае **бинарной классификации**.

Например, мы можем пытаться классифицировать спам или делать более сложную **многоклассовую классификацию**, например, на кошечек, собачек или крокодилов. Самое простое, что может прийти в голову в случае бинарной классификации, это использовать уже знакомую вам линейную регрессию и смотреть на знак полученного числа. Линейная регрессия предсказывает действительное число, и, например, если у нас знак меньше 0, то мы считаем, что это 0, а если больше 0, можем считать, что это 1. Таким образом, мы пытаемся построить какую-то разделяющую поверхность. **Если у нас классы линейно разделимые, это будет иметь смысл**. И все точки, лежащие сверху мы, например, относим к классу 1, а все точки, лежащие снизу, мы относим к классу 0.

Однако, это не всегда работает, и, например, нам иногда хотелось бы предсказывать вероятность принадлежности к какому-то классу, и здесь нам на помощь приходит логистическая регрессия. **Логистическая регрессия** использует в своей основе **сигмоид-функцию**, в которую мы отправляем наш линейный классификатор, и сигмоид-функция возвращает число от 0 до 1, которое можно уже отнести в соответствие какой-то вероятности.

$$z = w_0 + w_1 x_1 + \cdots + w_k x_k,$$
$$y = sigmoid(z) = 1/(1 + e^{-z})$$

Чем ближе к 1 число, тем выше у нас вероятность принадлежности к какому-то классу. Мы можем построить наш классификатор, выбирая определенную отсечку, то есть **decision boundary**, в соответствии с тем, как мы хотим.

Для того чтобы оптимизировать логистическую регрессию, используется **логистическая функция потерь logloss**.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)),$$

где $N$ - размер выборки, $y_i$ - корректный ответ, $\hat{y}_i$ - наше предсказание.

Можно её представлять себе как какое-то взвешенное accuracy, то есть **logloss нас сильно штрафует за уверенность в неправильных предсказаниях**.

## 1.5 Применение линейных моделей

Поговорим про линейные модели, посмотрим как они применяются в библиотеке *sklearn* и разберем две задачи: регрессии и классификации.

Итак, начнем с импортов. Импортируем библиотеки для визуализации с которыми вы уже встречались *Matplotlib* и *seaborn*. Импортируем знакомую вам *pandas*, и будем работать с библиотекой *sklearn* и ее модулем *datasets*. В *datasets* содержится какой-то набор стандартных базовых dataset, на которых можно проверять работу ваших моделей, смотреть как они себя ведут и просто тренироваться.

```
%matplotlib inline

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

from sklearn import datasets
```

И для начала будем решать задачу регрессии, то есть предсказания какой-то вещественной переменной с помощью dataset *boston*, в котором содержится информация о цене и каких-то характеристиках домов в районах Бостона. Можно посмотреть на описание этого *dataset*: здесь есть разные характеристики в этих районах, и целевая переменная - это среднее значение цены дома в 1000 долларов.

```
boston = datasets.load_boston()
boston.keys()

print(boston.DESCR[100:1300])
```

То есть, вот у нас целевая переменная *MEDV* и различные характеристики вроде *CRIM_rate*, то есть уровень преступности в этом районе, возраст, количество комнат и так далее.

В общем, используя все эти переменные, хотелось бы уметь предсказывать целевое значение, то есть среднюю стоимость дома. Этим мы и займемся. Давайте сначала посмотрим на наши данные, всегда стоит смотреть на dataset, чтобы примерно представлять как он выглядит, есть ли там какие-то пропуски, какие там есть значения, какие категориальные фичи, какие признаки являются просто вещественными числами.

```
boston_df = pd.DataFrame(boston.data, columns=boston.feature_names)
boston_df.head()
```

Здесь у нас просто числа, и мы можем решать эту задачу, задачу предсказания вещественной переменной с помощью линейной регрессии, чем мы и займемся.

Давайте посмотрим на среднюю цену дома с помощью *distplot* из *seaborn*.

```
plt.figure(figsize=(6, 4))
sns.distplot(boston.target)

plt.xlabel('Price (in thousands)')
plt.ylabel('Count')
plt.tight_layout()
```

Средняя цена дома у нас в районе 20-30 тысяч. В принципе, если мы будем предсказывать всегда 20 или 30 тысяч по этому dataset, то скорее всего мы уже будем неплохо угадывать.

Однако, нам конечно интересно обучать линейной регрессию, для этого импортируем из модуля *linear_model* соответствующий класс *LinearRegression* и получим instance этого класса - модель, которую мы уже будем обучать. Для того чтобы **обучить модель в *sklearn***, нужно вызвать метод *fit*. Метод *fit* вызывается очень часто - это стандартный интерфейс, который используется в подавляющем большинстве моделей. Мы передаем методу *fit* наши данные. *Boston.data* это набор объектов, набор информации о признаках. *Target* - это как раз целевая переменная, набор, список целевых переменных по каждому дому, то есть средняя цена дома в этой области.

```
from sklearn.linear_model import LinearRegression

linear_regression = LinearRegression()
model = linear_regression.fit(boston.data, boston.target)
```

И мы обучили нашу модель и сохранили состояние в переменной *model*.

Итак, что же теперь мы можем делать? У нас есть обычная модель. Хотелось бы конечно посмотреть на то, как она себя ведет. Давайте посмотрим, во-первых, как она обучалась. Как мы знаем, линейная регрессия разбрасывает веса каждому признаку для того чтобы оценить то, какой они вносят вклад в предсказание вещественной переменной. Таким образом, если мы перемножим все веса с нашими признаками, то мы должны получить нашу целевую переменную, то есть среднюю цену дома.

```
feature_weight_df = pd.DataFrame(list(zip(boston.feature_names, model.coef_)))
feature_weight_df.columns = ['Feature', 'Weight']
print(feature_weight_df)
```

Как видим, у нас есть разные веса у разных признаков, мы это все вывели, а есть какие-то большие веса, а есть какие-то маленькие веса. Оценивать их абсолютное значение здесь не стоит, потому что мы никак данные не нормализовали. Однако, если бы мы их привели к какому-то одному порядку, можно было бы смотреть на то, как они действительно вносят вклад в предсказательную способность нашей модели.

Давайте **вручную произведем предсказание**, мы знаем что линейная регрессия просто

приумножает веса с признаками, давайте сделаем именно это. Причём **нужно не забыть свободный коэффициент**, то есть *intercept*, как он здесь называется, который всегда есть у обученной модели линейной регрессии.

```
import operator

first_predicted = sum(map(
    lambda pair: operator.mul(*pair),
    zip(model.coef_, boston.data[0])
))

first_predicted += model.intercept_

print(first_predicted)
```

У нас получилось примерно 30 тысяч долларов, что уже ближе к правде. Как мы помним у нас среднее значение как раз 20 или 30 тысяч, значит значение уже как минимум имеет смысл.

Однако конечно нам **не нужно каждый раз вручную перемножать все веса со значениями признаков, мы можем это делать с помощью стандартных методов**. Стандартный метод *predict* вызывается у модели на данных, чтобы получить конкретное предсказание, он делает то же самое, что мы делаем вручную, однако делает автоматически. Это тоже стандартный интерфейс - у всех моделей *sklearn*, кроме метода *fit*, есть также метод *predict*, который позволяет вам предсказывать значение.

```
predicted = model.predict(boston.data)

print(predicted[:10])
```

Мы можем, для того, чтобы оценить примерно качество нашей работы на то, как наше предсказание соответствует реальным значением, **вывести такую табличку**.

```
predictions_ground_truth_df = pd.DataFrame(list(zip(predicted, boston.target)))
predictions_ground_truth_df.columns = ['Prediction', 'Ground truth']
predictions_ground_truth_df.head()
```

Конечно в реальной жизни никто не смотрит глазами таблички, есть более качественные и лучшие методы оценки качества построенной модели. Например, мы можем **построить график**, на котором будем отображать предсказанное значение и настоящее значение.

```
plt.figure(figsize=(6, 4))
plt.scatter(predicted, boston.target)
plt.xlabel('Predicted')
plt.ylabel('Ground truth')

plt.plot([0, 50], [0, 50], color=\"red\")
plt.tight_layout()
```

В идеале мы ожидаем, что у нас все значения будут ложиться на линию под 45 градусов. Здесь

конечно у нас есть какая-то ошибка, какой-то разброс - что ожидаемо, потому что линейная модель, модель линейной регрессии достаточно простая модель, достаточно слабая модель.

Давайте коротко посмотрим на **задачи классификации** тоже на стандартном dataset. Будем классифицировать рак груди на доброкачественную опухоль и злокачественную опухоль. Это тоже стандартный dataset, его тоже можно **загрузить** с помощью модуля *datasets*.

```
cancer = datasets.load_breast_cancer()
cancer.keys()
```

И можем посмотреть на **описание этого dataset**.

```
print(cancer.DESCR[:760])
```

Здесь около 30 характеристик нашей конкретной опухоли, всего около 600 объектов. Мы можем на этом деле обучаться, чтобы предсказывать уже классификацию, то есть решать задачу классификации, предсказывать 0 или 1, доброкачественная или злокачественная у нас опухоль.

Давайте **посмотрим на наши данные**.

```
cancer_df = pd.DataFrame(cancer.data)
cancer_df.columns = cancer.feature_names
cancer_df.head()
```

Здесь у нас есть тоже числовые переменные, которые говорят о каких-то характеристиках нашего объекта.

Можно посмотреть также **распределения классов**.

```
plt.figure(figsize=(6, 4))
sns.countplot(cancer.target)

plt.xlabel('Class')
plt.ylabel('Count')
plt.tight_layout()
```

**Смотреть разделение классов в задаче квалификации всегда очень полезно и важно**. Во-первых, нужно посмотреть: нет ли там каких-то пропусков значения. Во-вторых, в зависимости от того, как распределяются классы, если допустим у нас одного класса больше, другого меньше, у нас по-разному будут вести себя модели. Всегда важно понимать: равное у нас распределение или нет. Здесь у нас одного класса примерно в два раза больше, что в нашем случае не страшно. Мы с вами чуть попозже поговорим о том, как это можно нормализовать.

Итак, давайте тоже **импортируем нашу линейную модель** - логистическую регрессию из того же самого модуля *linear_model*. Сделаем всё точно то же самое: инстанциируем наш класс и вызовем у инстанса метод *fit*. Мы передаем ему наши данные, описание наших объектов в соответствии с признаками и целевую переменную, которая в данном случае является дискретным ответом 0 или 1, то есть доброкачественная или злокачественная опухоль.

```
from sklearn.linear_model import LogisticRegression
logistic_regression = LogisticRegression()
model = logistic_regression.fit(cancer.data, cancer.target)
```

И мы можем посмотреть на **коэффициенты обычной модели**, здесь у нас тоже какие-то числа, потому что это линейная модель.

```
print(model.coef_)
```

И сразу можно вызвать метод *predict* и **посмотреть на предсказание**. Здесь мы уже предсказываем не вещественное число, а дискретный ответ: 0 или 1.

```
prediction = model.predict(cancer.data)
print(prediction[:10])
```

Также у нашей модели есть метод *predict_proba*, потому что у нас логистическая регрессия, и часто нам бывает полезно посмотреть не только на предсказанный класс, а ещё и на то, **насколько модель уверена в своем предсказании**, насколько она распределяет свою верность по разным классам. Например, мы можем предсказывать 0, однако делать это с уверенностью 0.6, что достаточно сильно отличается от уверенности 0.9. И часто это бывает полезно, например, когда мы решаем задачу многоклассовой классификации, когда у нас есть много ответов: не два ответа, а допустим 10, и мы хотим посмотреть в целом на распределение ответов в нашей модели, чтобы как-то это дело обрабатывать.

```
prediction = model.predict_proba(cancer.data)
print(prediction[:10])
```

Ну и у модели есть метод *score*, который возвращает *Accuracy*.

```
print('Accuracy: {}'.format(model.score(cancer.data, cancer.target)))
```

В данном случае мы обучились достаточно хорошо, это простой dataset, и мы можем с вероятностью примерно 0.95 угадывать доброкачественная или злокачественная опухоль у нашего объекта.

Понятное дело, что у моделей существует гораздо больше параметров, чем мы с вами рассмотрели. Вы можете посмотреть на них в документации и как-то поиграть с их параметрами, посмотреть, как ведут себя предсказания в зависимости от разных выборок, разных параметров.

# Измерение качества моделей

## 2.1 Данные и переобучение

Поговорим о том, как готовить данные и как обучать модели машинного обучения с учителем. Допустим, у нас есть наша модель, есть какой-то набор данных, какой-то dataset. Что же нам с ними делать?

Самое логичное, самое простое, что может прийти в голову, просто взять все эти данные, отправить их в нашу модель и обучать, например, нашу модель с помощью какого-то метода оптимизации, например, с помощью градиентного спуска. Таким образом мы обучаем нашу модель, а в модели будут какие-то параметры. Мы потом в дальнейшем можем эту модель использовать. Однако, как мы можем проверить, насколько хорошо наша модель работает с какими-то новыми данными? Ведь основная цель построения алгоритма обучения с учителем состоит в том, чтобы наша модель не только работала хорошо на тренировочной выборке, на тренировочных данных, но и хорошо работала в мире, чтобы мы ее отправили потом куда-то в продакшн, и она там делала корректные закономерные предсказания.

Если мы используем все наши данные для того чтобы обучать эту модель, мы можем выяснить, что она работает как-то некорректно или наоборот хорошо только когда мы её уже проверим в реальном мире, что, разумеется, является не очень хорошей практикой. Поэтому, чаще всего dataset разбивают на несколько частей. Например, **логичным является разбиение на тренировочную и тестовую выборку**. На тренировочной выборке мы обучаемся и настраиваем параметры нашей модели, а потом проверяем работу нашей модели на тестовой выборке, чтобы понять, насколько она хорошо делает предсказания.

Однако в этом случае могут быть различные проблемы, например, что если наша модель хорошо работает на тренировочной выборке и плохо работает на тестовой выборке? В таком случае наша модель очевидно переобучилась на тренировочную выборку - это называется **оверфиттинг**, и она описывает какие-то шумовые зависимости нашей тренировочной выборки, но не описывает природные зависимости, которые мы хотим моделировать.

Так конечно не годится, и для того чтобы бороться с этим, существуют различные методы. Во-первых, когда мы разбиваем наш dataset на тренировочную или тестовую выборку, нужно делать это с умом. **В большинстве случаев подходит разбиение просто случайно: мы перемешиваем весь наш dataset и разбиваем на две части**. Например, часто бывает разбиение на 80 или 20 процентов, или 90 и 10. Однако, все зависит от количества объектов. Например, если у вас большое количество объектов, вы можете выделить какую-то тысячу только для теста и тестировать на ней. Однако, нужно быть внимательными когда вы, например, решаете задачу предсказания по временным каким-то сущностям, по временным рядам, в таком случае пере-

мешивать конечно нельзя, иначе вы будете заглядывать в будущее, и вам для обучения нужно использовать данные разбитые как-то по времени. Или также нужно, например, следить, если у вас существует какой-то класс, у которого, например, мало объектов, нужно смотреть чтобы все объекты этого класса не оказались, например, в вашей тестовой выборке, иначе вы просто не обучитесь.

Итак, что же делать когда наша модель переобучилась? Например, в случае линейной регрессии, мы можем попытаться уменьшить мощность нашей модели, например, **построить полином меньшей степени**, в таком случае наша модель будет более простой и будет пытаться описывать действительную реальную природную зависимость данных, а не какие-то шумовые выбросы.

Еще одним подходом, конечно же является **регуляризация**. С помощью регуляризации мы можем попытаться найти модель, которая будет описывать данные лучше.

Что же такое регуляризация? Как вы можете догадаться, когда мы строим предсказательную модель, мы пытаемся найти какую-то функцию в большом семействе функций, которые могут нам подойти. То есть, не существует какого-то одного идеального решения, нам может подойти довольно много различных, допустим, линейных моделей, которые примерно одинаково будут работать. Однако у них у всех будут различные параметры, и они все будут какой-то различной сложности. **Регуляризация как раз пытается сделать так, чтобы мы нашли максимально простую модель**, модель, например, с наименьшими весами в случае линейной регрессии. Например, мы можем рассмотреть **регуляризацию L2**.

$$ridge\_loss = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 + \lambda\sum_{i=1}^{k} w_i^2,$$

где $N$ - размер выборки, $y_i$ - корректный ответ, $\hat{y}_i$ - наше предсказание, $\lambda$ - вес при регуляризаторе, $w_i$ - веса модели, $k$ - количество весов.

Таким образом, **мы штрафуем дополнительно за большие веса** и большие веса, соответственно, соответствуют каким-то большим пикам в нашей линейной регрессии. Мы пытаемся сделать так, чтобы их не было. Таким образом мы пытаемся найти функцию в семействе всех функций, которые нам могут подойти, которая будет попроще, которая будет хорошо описывать природную зависимость.

Также существует **L1 регуляризация**, которая использует модули весов.

$$ridge\_loss = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 + \lambda\sum_{i=1}^{k} |w_i|,$$

где $N$ - размер выборки, $y_i$ - корректный ответ, $\hat{y}_i$ - наше предсказание, $\lambda$ - вес при регуляризаторе, $w_i$ - веса модели, $k$ - количество весов.

Она, например, позволяет отбирать наши признаки таким образом, чтобы мы **зануляли признаки, которые не нужны для нашей модели**.

Итак, регуляризация помогает вам бороться с оверфиттингом, бороться с переобучением, позволяя выделить какую-то конкретную функцию в семействе всех функций, которая нам подходит, сделать так, чтобы она была простой.

Однако, как вы можете догадаться, даже если мы разбиваем на тренировочную на тестовую выборку наш dataset и последовательно обучаем нашу модель, допустим обучаем на тренировоч-

ном dataset, проверяем ее на тестовом dataset, выбираем какие-то параметры по тестовому dataset и делаем это несколько раз, делаем такие циклы, постепенно мы так или иначе переобучаемся еще и под тестовую выборку. Потому что мы модифицируем параметры нашей модели так, чтобы они хорошо работали на тренировочной и на тестовой выборке. Поэтому, чаще всего используются более сложные методы разбиения dataset. Например, **разбиение dataset на три части**. На **тренировочную выборку**, на которой мы тренируемся, на которой мы обучаем нашу модель, на **тестовую выборку**, на которой, например, мы подбираем параметры нашей модели, и еще одна выборка - **валидационная**, на которую мы не смотрим до последнего момента, а когда мы уже посылаем, например, нашу модель в продакшн или показываем ее гендиректору или публикуемся, мы считаем качество на валидационной выборке и уже считаем, что это является качество модели.

И ещё одним очень популярным методом валидации, методом разбиения, является **кросс-валидация**. Допустим, у вас есть небольшой dataset, вы не хотите какую-то часть данных для валидации хранить в каком-то темном помещении и вы хотите использовать все ваши данные, чтобы обучать вашу модель еще лучше. В таком случае используется кросс-валидация. Вы разбиваете ваш общий dataset на $n$ каких-то примерно равных частей и последовательно используете одну из этих частей в качестве тестовой выборки, остальные в качестве тренировочной. Ну и меняете. Таким образом, каждая часть ваших данных в отдельных случаях является тестовой, а остальные - тренировочными. Вы используете все ваши данные, тем не менее это позволяет вам не переобучаться. В конце вы усредняете какое-то качество по кросс-валидации и используете его уже как метрику по кросс-валидации.

Ещё одним важным моментом, о котором хотелось бы поговорить, является **bias-variance tradeoff**, или **компромисс между смещением и разбросом**. Когда мы обучаем модель, на самом деле мы пытаемся угнаться сразу за двумя зайцами. С одной стороны мы пытаемся максимально хорошо описать нашу тренировочную выборку, учесть все зависимости, которые там есть, а с другой стороны мы не хотели бы переобучаться под неё, потому что, на самом деле модель будет использоваться потом в мире, и мы хотим построить обобщаемую модель. С одной стороны, если мы построим модель, которая слишком хорошо описывает тренировочную выборку, у нас будет большой variance - когда мы попытаемся использовать нашу модель на других данных у нас будет большой разброс в оценках. С другой стороны, если мы построим слишком простую модель, которая слабо описывает тренировочную выборку, то у нас просто будет плохое предсказание с каким-то смещением. Таким образом мы пытаемся играть в эту игру, и это называется bias-variance tradeoff. Нужно понимать, **когда вы обучаете вашу модель, что вы всегда пытаетесь выбрать какое-то меньшее из двух зол**.

## 2.2 Метрики качества

Поговорим о том, как же оценивать качество построенных моделей машинного обучения. Допустим, вы обучили вашу линейную регрессию или классификацию и хотите узнать, насколько хорошо они работают.

Давайте начнём с классификации. Так, в случае классификации вы предсказываете 0 либо 1, то есть срабатывает модель, либо не срабатывает, и можно построить **матрицу ошибок**, которая

показывает, как соотносятся ваши предсказания с реальными данными. Например, если у вас модель сработала и показала 1, а на самом деле должен быть 0, то это **false positive (FP)**, а если вы показали 0 там, где должен быть 0, это **true negative (TN)**. Самый простая и логичная метрика, которую можно использовать, в задаче классификации, это **accuracy**, то есть доля правильных ответов в вашей модели.

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Однако, несмотря на то, что accuracy довольно часто используется, иногда совершенно не применима. Например, когда у нас есть серьёзный дисбаланс классов. Допустим, мы решаем задачу классификации пациентов на здоровый и нездоровый, и у нас есть какая-то редкая болезнь, которая встречается в одном случае, например, из 100. И если мы построим какой-то константный предсказать, который всегда говорит, что человек здоров, то наша accuracy будет, 99%. Однако, несмотря на то, что accuracy довольно высокая, наша модель ничего полезного не делает и ничего нового в мир не принесла, смысла не имеет, поэтому используются более сложные метрики, вроде precision или recall.

**Precision** показывает долю корректного срабатывания среди всех положительных срабатываний, то есть, если мы показываем 1, то насколько часто мы делаем это точно.

$$precision = \frac{TP}{TP + FP}$$

**Recall** говорит о том, насколько много 1 в целом мы нашли, как полно мы описали наши данные.

$$recall = \frac{TP}{TP + FN}$$

**Их можно использовать в случае дисбаланса классов, и часто именно так и делается**.

**Если мы хотим как-то объединить precision и recall**, можно использовать, например, гармоническое среднее между ними.

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

Также существует **logloss**, с которым вы уже встречались, когда мы обсуждали логистическую регрессию - довольно страшная функция, которую можно рассматривать как какое-то взвешенное accuracy, то есть наш logloss сильно штрафуют за уверенность в неправильном ответе.

И ещё двумя важными конструкциями, которые используются, пожалуй, наиболее часто, являются **ROC AUC** и **P-R AUC**.

В случае **ROC AUC**, мы откладываем **true positive rate** против **false positive rate**.

$$true\_positive\_rate = \frac{TP}{TP + FN}$$

$$false\_positive\_rate = \frac{FP}{FP + TN}$$

Важно знать, что у нас высчитывается площадь под нашей построенной кривой. Таким образом, **если наша площадь равна 1, если мы полностью закрыли единственный квадрат, то у нас наш классификатор является идеальным. Если у нас площадь около 0.5, то он, примерно, случайный и смысла большого не имеет**.

То же самое с **precision-recall кривой**, которая часто используется в случае дисбаланса классов. Здесь, если у нас 1, - у нас классификатор идеальный, и так далее.

В случае регрессии мы предсказывали какое-то действительное число, и с какими-то из метрик вы уже знакомились.

Например, **MAE**, или **средняя абсолютная ошибка**, показывает на то, как далеко наши ответы в среднем отстоят от корректных, то есть мы суммируем расстояние до корректных предсказаний от наших попыток описать данные.

**MSE** делает то же самое с квадратным расстоянием и усредняет по всем объектам.

Также мы можем получить **RMSE** метрику, которая часто используется.

$$RMSE = \sqrt{MSE}$$

Ещё одна новая метрика, с которой мы ещё не ознакомились, - это **R-squared**, или **коэффициент детерминации**.

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \hat{y})^2},$$

где $N$ - размер выборки, $y_i$ - корректный ответ, $\hat{y}_i$ - наше предсказание, $\hat{y}$ - среднее значение по выборке.

Коэффициент детерминации показывает, как хорошо мы можем описать дисперсию в наших данных. Важно знать, что коэффициент детерминации может лежать от $-\infty$ до 1. **В случае единицы мы показываем идеальное качество, а если у нас R-squared меньше нуля, то смысла наша модель большого не имеет**.

## 2.3 Применение метрик качества

Поговорим про то, как оценивать качество построенных моделей и как разбивать наши данные так, чтобы быть уверенными в том, что наша модель будет хорошо работать на новых данных, которых мы еще не видели, например, в реальном мире.

Будем работать с библиотекой *sklearn* и стандартными dataset, с которыми мы уже встречались, например, dataset по раку груди. Также импортируем новый модуль *metrics*, в котором как раз содержатся функции, помогающие нам оценить качество работы построенных моделей. И для начала будем работать с логистической регрессией, с которой уже знакомы.

```
from sklearn import datasets
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
```

**Обучаем нашу логистическую регрессию** с помощью метода *fit* на всех данных и посмотрим на *Accuracy*, которую возвращает метод *score* у нашей модели.

```
cancer = datasets.load_breast_cancer()
logistic_regression = LogisticRegression()
model = logistic_regression.fit(cancer.data, cancer.target)

print('Accuracy: {:.2f}'.format(model.score(cancer.data, cancer.target)))
```

Точно так же можно посмотреть на *Accuracy*, **построив предсказания** с помощью метода *predict* на всех наших данных, и вызвать у модуля *metrics* различные функции. Например, *accuracy_score* возвращает *Accuracy*. Существует достаточно большое количество различных метрик как для классификаций, так и для регрессий, которые можно вызывать, использовать и смотреть.

```
predictions = model.predict(cancer.data)

print('Accuracy: {:.2f}'.format(metrics.accuracy_score(cancer.target, predictions)))
print('ROC AUC: {:.2f}'.format(metrics.roc_auc_score(cancer.target, predictions)))
print('F1: {:.2f}'.format(metrics.f1_score(cancer.target, predictions)))
```

Итак, как мы с вами уже говорили, обучаться на всех данных не всегда здорово, потому что мы не можем достоверно оценить, как наша модель будет вести себя на каких-то новых данных. Например, когда мы запустим нашу модель в продакшн. Было бы здорово обучать наши модели на каких-то одних данных и потом смотреть как она будет генерелизоваться, то есть вести себя на каком-то новом наборе данных. Для этого нам может помочь *model_selection*.

```
from sklearn.model_selection import train_test_split
```

В *train_test_split* мы передаем наши данные, передаем наши признаки и передаем нашу целевую переменную, и будем разбивать наши данные на тестовую и тренировочную выборку, как мы с вами уже говорили. У нас будет тренировочная выборка, на которой мы обучаемся, в которую мы подбираем параметры нашей модели; а потом будем оценивать её качество уже на тестовой выборке, на новых объектах, которые мы ещё не видели, чтобы посмотреть, как наша модель выучила природную зависимость. Мы будем делать это с помощью какого-то случайного *seed* 12, для того чтобы у нас зафиксировалось качество раз навсегда.

```
x_train, x_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target,
    test_size=0.2, random_state=12
)
model = logistic_regression.fit(x_train, y_train)

print('Train accuracy: {:.2f}'.format(model.score(x_train, y_train)))
print('Test accuracy: {:.2f}'.format(model.score(x_test, y_test)))
```

На тесте у нас качество чуть хуже. Это достаточно логичная ситуация, мы обучались на одних данных, а смотрим качество на немного других. Если у нас отличие небольшое, как в данном случае, это не страшно. Если у нас бы было, например, 0.6, то скорее всего можно считать, что модель у нас переобучилась под тренировочную выборку.

Давайте посмотрим, как можно сделать то же самое для задачи регрессии. Здесь отличий немного, однако мы посмотрим на несколько новых классов, на другие возможности линейной регрессии.

Итак, импортируем из *linear_model* 3 класса - это *Lasso*, *Ridge* и *ElasticNet*, которые представляют из себя **единую регрессию с различными видами регуляризации**. *Lasso* - это L1 регуляризация, когда мы добавляем в нашу функцию ошибки первую норму весов, то есть мы добавляем функции сумму модулей весов, которые мы в данный момент имеем в нашей линейной регресси. *Ridge* - это грибневая регрессия, которая добавляет сумму квадратов весовю. *ElasticNet* - это и то и другое, мы добавляем как L1 так и L2 регуляризацию. Будем работать с dataset *boston*.

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

boston = datasets.load_boston()

lasso = Lasso()
ridge = Ridge()
elastic = ElasticNet()

for model in [lasso, ridge, elastic]:
x_train, x_test, y_train, y_test = train_test_split(\
    cancer.data, cancer.target,
    test_size=0.2
)
model.fit(x_train, y_train)

predictions = model.predict(x_test)
print(model.__class__)
print('MSE: {:.2f}\\n'.format(metrics.mean_squared_error(y_test, predictions)))
```

Как мы видим лучше всего ведет себя гребневая регрессия: в данном случае она показала MSE чуть ниже, чем остальные функции.

Мы также можем работать с другими функциями из модуля *metrics*, например, **L2 score**.

```
print('R2: {:.2f}'.format(model.score(x_test, y_test)))
print('R2: {:.2f}'.format(metrics.r2_score(y_test, predictions)))
```

Есть еще один полезный метод, полезный подход, который используется очень часто для, того чтобы не откладывать какой-то набор данных в черный ящик, чтобы потом на нем тестироваться, а использовать все данные; однако делается это так, чтобы знать, что наша модель не переобучается и работает хорошо. Метод, конечно, называется **кросс-валидация**.

Работать будем с новым dataset - *iris*, который тоже стандартный. Это dataset про цветы, про ирисы, решает задачу классификации.

```
from sklearn.model_selection import KFold, cross_val_score

iris = datasets.load_iris()
iris.keys()
```

У нас есть четыре признака у нашего цветка - это длина лепестка, ширина лепестка и длина чашелистника и его ширина. Мы пытаемся предсказать три разных класса ирисов, оказывается, что можно это делать только по этим характеристикам. То есть мы пытаемся угадать 1 из 3-ёх классов: 0, 1 или 2.

Итак, будем решать задачу с помощью логистической регрессии. Для того чтобы использовать кросс-валидацию, можно, например, в цикле использовать метод *split* и разбить наши данные, например, на 5 фолдов - непосредственно пять раз использовать каждый из них в качестве тестового блока.

```
logistic_regression = LogisticRegression()
cv = KFold(n_splits=5) # +StratifiedKFold

for split_idx, (train_idx, test_idx) in enumerate(cv.split(iris.data)):
    x_train, x_test = iris.data[train_idx], iris.data[test_idx]
    y_train, y_test = iris.target[train_idx], iris.target[test_idx]

    logistic_regression.fit(x_train, y_train)
    score = logistic_regression.score(x_test, y_test)
    print('Split {} Score: {:.2f}'.format(split_idx, score))
```

В данном случае *Accuracy* у нас немного варьируется: есть хорошие, есть плохие.

Но конечно, логичным вариантом было бы **усреднять это качество на кросс-валидации** и брать среднее. Однако, конечно, интерироваться так по крос-валидации не всегда приятно, было бы здорово, если бы это делалось автоматически. Конечно, есть такая функция: она называется *cross_val_score*.

```
cv_score = cross_val_score(
    logistic_regression, iris.data, iris.target,
    scoring='accuracy', cv=cv
)

print('Cross val score: {}'.format(cv_score))
print('Mean cross val score: {:.2f}'.format(cv_score.mean()))
```

# Ансамблевые модели

## 3.1 Решающие деревья

Вы уже знакомы с методами линейной классификации, линейной регрессии, в целом с линейными моделями, однако их можно использовать, как следует из их названия, только в случае каких-то линейных зависимостей. И они достаточно сильно отличаются от того, как на самом деле люди принимают решения. Например, мы можем представить себе ситуацию, когда пациент приходит к доктору, и доктор последовательно задает ему ряд вопросов, постепенно сходясь к какому-то своему диагнозу и решению. Например, он может спросить про температуру, болит ли горло и так далее. Точно так же и работают решающие деревья. Например, мы можем показать ему новый объект, и по каким-то критериям, по каким-то вопросам постепенно сойтись к листу. В случае классификации у нас в листе будет класс, в случае регрессии, например, какой-то диапазон.

**Можно представить себе решающие деревья как алгоритм, который последовательно разбивает пространство по каким-то различным критериям**. Делается это так, чтобы максимизировать количество похожих объектов в поддереве, в левом или правом, нормируя по количеству этих самых объектов. И строим мы наше дерево до конца, например, или до какой-то конкретной глубины, или до степени дискретизации ответа, если нам это интересно. А в листе, например, в случае классификации мы берем наиболее частый класс или какое-то среднее значение в случае регрессии.

На самом деле деревья сами по себе не используются практически никогда. Они используются только в какой-то композиции, в ансамбле, о котором мы поговорим попозже, потому что деревья очень легко переобучаются. Если мы построим достаточно глубокое дерево, смысла большого в нем не будет, оно будет переобучено на вашу тренировочную выборку, и использовать его где-то в дальнейшем будет практически невозможно. Тем не менее, они **очень полезны, когда используются в ансамблях**, о чем мы поговорим дальше.

## 3.2 Случайный лес

Поговорим про **случайный лес**, или **random forest**, который является, по сути, композицией большого количества решающих деревьев.

Итак, чтобы понять, почему и как работает случайный лес, важно знать, как на самом деле работают в целом все ансамблевые модели. В случае **ансамблевой модели** мы строим большое количество слабых разных базовых алгоритмов и потом используем их все вместе, как-то усредняем.

Сами по себе эти базовые алгоритмы не имеют большого смысла. Они сильно переобучены,

у них большой разброс, и использовать их по отдельности нет смысла. Однако вместе они дают достаточно сильные предсказания. Например, есть известная история про ярмарку в 19 веке, когда 800 крестьян пытались делать предсказания веса быка, который был на этой ярмарке, и никто из этих крестьян не угадал. Однако если усреднить все их предсказания, окажется, что полученное число с точностью до одного фунта угадывает вес этого быка. Так и работают в целом все композиционные методы. **Мы берем большое количество различных предсказаний, которые сами по себе достаточно сильно разбросаны, и как-то их усредняем, и потом используем уже в качестве достаточно сильного итогового алгоритма**.

**Как же работает случайный лес**? При построении случайного леса мы независимо строим большое количество различных решающих деревьев. Как же сделать их различными? Мы, например, можем брать какие-то подмножества объектов или признаков. Объекты, например, мы можем брать с помощью алгоритма, который называется **bootstrap**. Когда у нас есть, например, тренировочная выборка размера $m$, мы можем взять оттуда $n$ объектов с возвращением. Таким образом мы каждый раз берем объект и оставляем его там же, и можем, в принципе, взять его второй раз. В нашу тренировочную выборку, которая на самом деле используется, попадают примерно 63% объектов, и мы на них обучаемся. Остальные, например, мы можем использовать для валидации, это называется **out-of-bag**.

Таким образом мы строим глубокие деревья независимо. Мы строим наши решающие деревья достаточно глубоко, чтобы у них был большой разброс. А так как они **строятся независимо, мы можем все это дело, весь этот процесс параллелить, то есть использовать обучение, построение деревьев на различных машинах**, что часто используется в различных реализациях.

Когда мы строим наше дерево, мы можем при разбиении в листах, при разбиении в вершинах дерева использовать различные случайные подмножества признаков. Есть различные эвристики на количество признаков при разбиении.

Что важно знать про случайные леса? Что они **редко переобучаются**. Таким образом мы можем строить всё больше и больше различных решающих деревьев, их усреднять, и у нас будет всё лучше и лучше расти качество.

Однако есть и минусы. Например, нам **нужно строить действительно глубокие деревья, чтобы у них был большой разброс**. Иногда это действительно трудозатратно.

## 3.3 Градиентный бустинг

Как вы знаете, в случайном лесе наши базовые алгоритмы, наши решающие деревья строятся независимо. Таким образом их, например, можно параллелить, то есть процесс обучения можно производить на различных машинах.

А в случае **бустинга** наши базовые алгоритмы строятся зависимо, таким образом, чтобы каждый следующий алгоритм исправлял ошибки существующей модели. Таким образом у нас есть какой-то базовый алгоритм, мы добавляем к нему новый так, чтобы он исправил ошибку существующего и так далее. То есть делаем так до тех пор, пока наше итоговое качество не будет нас удовлетворять.

В отличие от случайного леса для градиентного бустинга над решающими деревьями, то есть когда у нас в качестве базовых алгоритмов используются решающие деревья, нам не обязательно строить наши деревья глубоко. Не обязательно их строить до конца. Таким образом нам достаточно каких-то небольших коротких деревьев. И это очень сильно влияет на производительность, на производительность при обучении. В случайном лесе нам нужно строить глубокие деревья и строить их много.

**Таким образом несмотря на то, что случайный лес часто дает достаточно хорошее качество, обучать его достаточно сложно**.

Для градиентного бустинга достаточно неглубоких деревьев, которые постепенно добавляются к нашему базовому алгоритму. Таким образом мы можем добавлять, например, наши новые базовые алгоритмы с шагом, чтобы наша модель не переобучалась. Тем не менее градиентный бустинг в отличие от случайных лесов действительно переобучается, и **нужно следить за тем, чтобы он не переобучался, добавляя различную регуляризацию**. Например, какой-то шаг, или можно использовать, например, стохастический градиентный бустинг.

## 3.4 Применение ансамблевых моделей

Поговорим о том, как работать с ансамблевыми моделями. А конкретно, со случайным лесом и градиентным бустингом в библиотеке *sklearn*. Импортируем библиотеки для визуализации, для работы с данными. И загрузим уже знакомый вам dataset *iris* из модуля *datasets*. Здесь мы будем решать задачу классификации, то есть, пересказывать вид цветка по характеристикам его лепестков.

```
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn import datasets

iris = datasets.load_iris()
```

Давайте посмотрим, **как эти характеристики распределяются в разных классах**. Построим *pairplot* с помощью библиотеки *seaborn*.

```
iris_df = pd.DataFrame(iris.data, columns=iris.feature_names)
iris_df['Species'] = np.array([iris.target_names[cls] for cls in iris.target])
sns.pairplot(iris_df, hue='Species')
```

И здесь мы видим, что у нас синий класс, то есть setosa, линейно разделим. Мы можем использовать даже какую-то линейную модель, для того чтобы определить, что наш цветок принадлежит именно к этому классу. Однако зеленый и красный класс гораздо более сильно похожи друг на друга, поэтому здесь уже нужно строить какую-то нелинейную зависимость, и строить более сложную модель, например, модель основанную на решающих деревьях. Именно это мы и будем делать.

Давайте импортируем из модуля ансамбль *RandomForestClassifier*, который позволяет решать задачу квалификации. Однако, конечно, с помощью случайного леса можно решать задачу регрессии, и соответствующий класс есть в этом модуле. Из модуля *metrics* мы будем импортировать *accuracy_score* и *confusion_matrix*, которую тоже будем использовать. Будем пользоваться стандартным *train_test_split* разбиением.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split
```

Давайте дистанцируем наш класс. У нас будет случайный лес, в котором сто деревьев, и зафиксируем какой-то случайный *seed*.

```
random_forest = RandomForestClassifier(n_estimators=100, random_state=42)

x_train, x_test, y_train, y_test = train_test_split(
    iris.data, iris.target,
    test_size=0.3, stratify=iris.target, random_state=42
)
rf_model = random_forest.fit(x_train, y_train)
```

Очень важно заметить, что у нас появился новый параметр *stratify*. **Очень важно, когда мы решаем задачу классификации, сделать так чтобы в нашем разбиении сохранялось распределение классов таким образом, чтобы мы не обучались на данных, в которых, например, нет какого-то одного класса, который потом появится в тестовой выборке**. В таком случае мы никак не сможем обучиться и узнать природную зависимость про этот класс. Нужно чтобы соотношение классов сохранялось при разбиении. То же самое важно, когда мы делаем кросс-валидацию.

Делаем **предсказание** мы уже на тестовой выборке и смотрим *Accuracy* с помощью стандартной метрики *accuracy_score*.

```
predictions = rf_model.predict(x_test)
print('Accuracy: {:.2f}'.format(accuracy_score(y_test, predictions)))
```

Давайте также посмотрим на *confusion_matrix*. **Посмотрим, какие предсказания мы делаем хорошо, а какие плохо**. Для этого посчитаем метрику *confusion_matrix* из стандартного модуля *metrics* и отобразим с помощью *seaborn hit map*.

```
confusion_scores = confusion_matrix(y_test, predictions)
confusion_df = pd.DataFrame(confusion_scores, columns=iris.target_names,
                            index=iris.target_names)
sns.heatmap(confusion_df, annot=True)
```

Как мы видим, и правда, setosa - это тот самый синий класс, достаточно хорошо разделим, и мы в нем не ошибаемся. Однако другие два класса немного смешиваются, потому что они похожи друг на друга.

Также с помощью моделей, основанных на решающих деревьях, всегда очень легко и просто посмотреть на *feature_importance*. То есть на то, **какой признак внёс наибольший вклад в**

**предсказания**. У каких признаков наибольшая предсказательная способность.

```
feature_importance = list(zip(iris.feature_names, rf_model.feature_importances_))
feature_importance_df = pd.DataFrame(feature_importance,
                                     columns=['Feature', 'RF Importance'])
feature_importance_df
```

В данном случае, случайный лес считает, что лучше всего определить класс цветка по ширине лепестка.

Конечно, у моделей существуют гораздо больше параметров, чем мы использовали. Можно включать и выключать bootstrap, ограничивать, например, глубину дерева, менять количество решающих деревьев и так далее. Например, из полезных можно всегда считать *oob_ score*, то есть смотреть качество на тех объектах, которые не использовались при обучении наших решающих деревьев.

```
rf_model.get_params()
```

Похожим образом можно также использовать **градиентный бустинг**. Здесь тоже он решает задачу классификации. Тоже можно решать задачу регрессии. Но для ирисов у нас классификация. Обучаем у нас 100 наших деревьев. Обучаем нашу модель на train и смотрим качество на тесте.

```
from sklearn.ensemble import GradientBoostingClassifier

gradient_boosting = GradientBoostingClassifier(n_estimators=100, random_state=42)
gb_model = gradient_boosting.fit(x_train, y_train)

print('Accuracy: {:.2f}'.format(gb_model.score(x_test, y_test)))
```

И в данном случае качество у нас чуть-чуть получше. Ну, так бывает.

Можно также посмотреть на качество, на то, как градиентный бустинг считает **важность у различных признаков**.

```
feature_importance_df['GB Importance'] = gb_model.feature_importances_
feature_importance_df
```

В данном случае мы видим, что качество распределяется немного по-другому, и признаки обладают немного другой важностью. Что логично, потому что это другая модель.

Из важных параметров десь тоже существует несколько принципиальных вещей. Например, мы можем видеть, что глубина дерева стандартная - всего три. Потому что в градиентном бустинге мы как раз хотим строить неглубокие деревья. Ну и другими параметрами тоже можно как-то играть и их варьировать.

```
gb_model.get_params()
```