Занятие 6. Задача классификации

Что такое машинное обучение?

Машинное обучение — как секс в старших классах. Все говорят о нем по углам, единицы понимают, а занимается только препод. Статьи о машинном обучении делятся на два типа: это либо трёхтомники с формулами и теоремами, которые я ни разу не смог дочитать даже до середины, либо сказки об искусственном интеллекте, профессиях будущего и волшебных дата-саентистах.

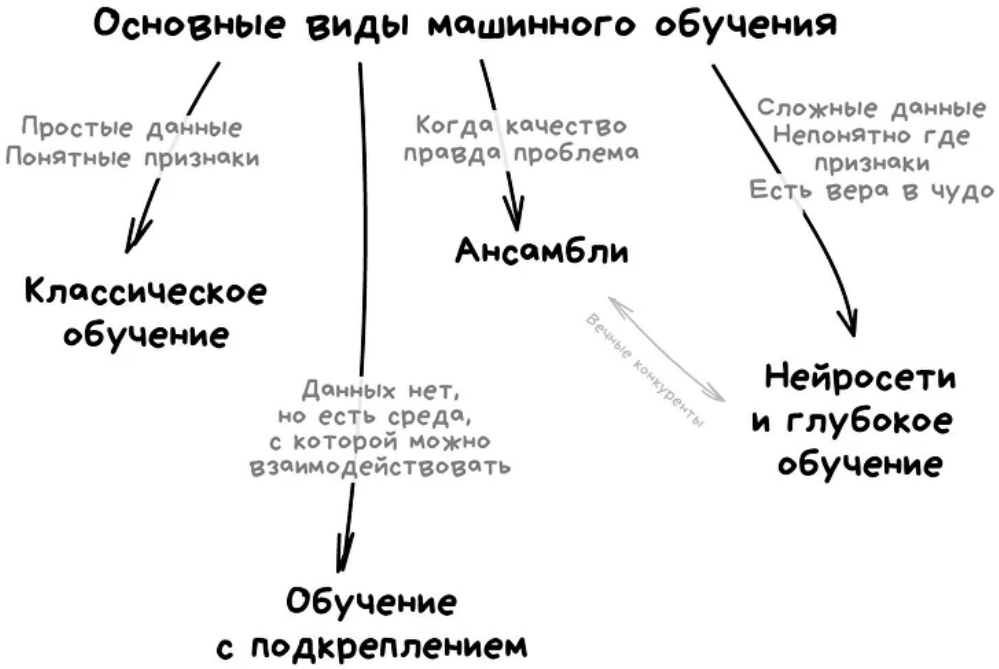
Цель машинного обучения — предсказать результат по входным данным. Чем разнообразнее входные данные, тем проще машине найти закономерности и тем точнее результат.

Итак, если мы хотим обучить машину, нам нужны три вещи:

- **Данные**. Хотим определять спам — нужны примеры спам-писем, предсказывать курс акций — нужна история цен, узнать интересы пользователя — нужны его лайки или посты. Данных нужно как можно больше. Десятки тысяч примеров — это самый злой минимум для отчаянных.
- **Признаки**. Мы называем их фичами (features), так что ненавистникам англицизмов придётся страдать. Фичи, свойства, характеристики, признаки — ими могут быть пробег автомобиля, пол пользователя, цена акций, даже счетчик частоты появления слова в тексте может быть фичей. Машина должна знать, на что ей конкретно смотреть. Хорошо, когда данные просто лежат в табличках — названия их колонок и есть фичи. А если у нас сто гигабайт картинок с котами? Когда признаков много, модель работает медленно и неэффективно. Зачастую отбор правильных фич занимает больше времени, чем всё остальное обучение. Но бывают и обратные ситуации, когда кожаный мешок сам решает отобрать только «правильные» на его взгляд признаки и вносит в модель субъективность — она начинает дико врать.
- **Алгоритм**. Одну задачу можно решить разными методами примерно всегда. От выбора метода зависит точность, скорость работы и размер готовой модели. Но есть один нюанс: если данные говно, даже самый лучший алгоритм не поможет. Не зацикливайтесь на процентах, лучше соберите побольше данных.

Искусственный интеллект и где начинается машинное обучение

Машина может	Машина не может
Предсказывать	Создавать новое
Запоминать	Резко поумнеть
Воспроизводить	Выйти за рамки задачи
Выбирать лучшее	Убить всех людей



Классическое обучение

Первые алгоритмы пришли к нам из чистой статистики еще в 1950-х. Они решали формальные задачи — искали закономерности в циферках, оценивали близость точек в пространстве и вычисляли направления. Сегодня на классических алгоритмах держится добрая половина интернета. Когда вы встречаете блок «Рекомендованные статьи» на сайте, или банк блокирует все ваши деньги на карточке после первой же покупки кофе за границей — это почти всегда дело рук одного из этих алгоритмов. Да, крупные корпорации любят решать все проблемы нейросетями. Потому что лишние 2% точности для

них легко конвертируются в дополнительные 2 миллиарда прибыли. Остальным же стоит включать голову. Когда задача решаема классическими методами, дешевле реализовать сколько-нибудь полезную для бизнеса систему на них, а потом думать об улучшениях. А если вы не решили задачу, то не решить её на 2% лучше вам не особо поможет.

Обучение с учителем

Классическое обучение любят делить на две категории — с учителем и без. Часто можно встретить их английские наименования — Supervised и Unsupervised Learning. В первом случае у машины есть некий учитель, который говорит ей как правильно. Рассказывает, что на этой картинке кошка, а на этой собака. То есть учитель уже заранее разделил (разметил) все данные на кошек и собак, а машина учится на конкретных примерах. В обучении без учителя, машине просто вываливают кучу фотографий животных на стол и говорят «разберись, кто здесь на кого похож». Данные не размечены, у машины нет учителя, и она пытается сама найти любые закономерности. Об этих методах поговорим ниже. Очевидно, что с учителем машина обучится быстрее и точнее, потому в боевых задачах его используют намного чаще. Эти задачи делятся на два типа:

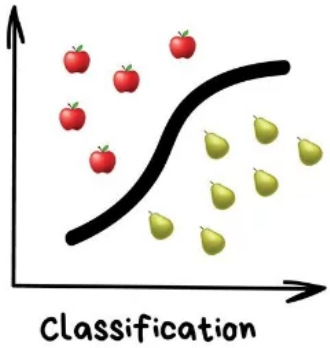
- **классификация** — предсказание категории объекта,
- **регрессия** — предсказание места на числовой прямой.

Что такое классификация

«Разделяет объекты по заранее известному признаку. Носки по цветам, документы по языкам, музыку по жанрам»

Сегодня используют для:

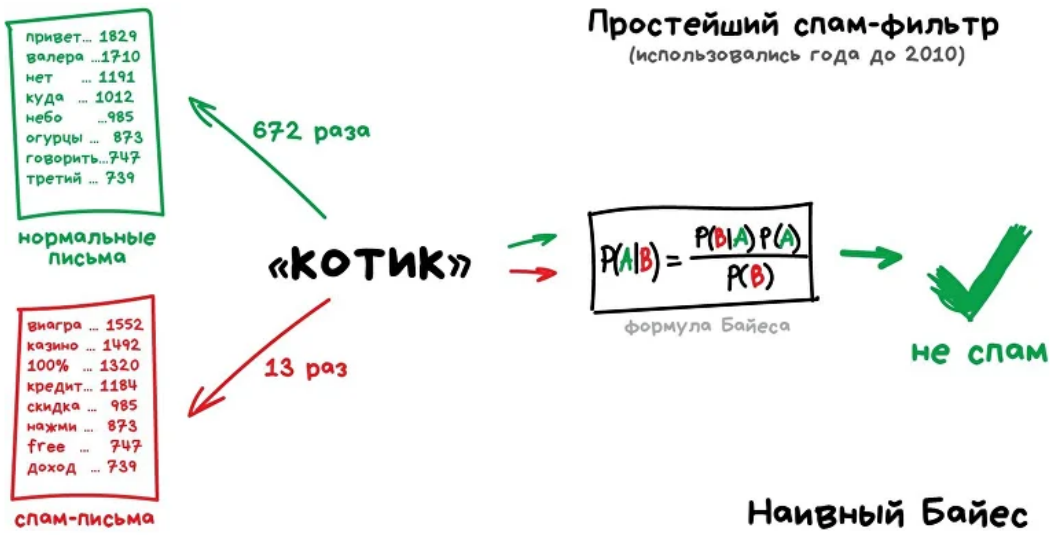
- Спам-фильтры
- Определение языка
- Поиск похожих документов
- Анализ тональности
- Распознавание рукописных букв и цифр
- Определение подозрительных транзакций



Классификация вещей — самая популярная задача во всём машинном обучении. Машина в ней как ребёнок, который учится раскладывать игрушки: роботов в один ящик, танки в другой. Опа, а если это робот-танк? Штош, время расплакаться и выпасть в ошибку.

Для классификации всегда нужен учитель — размеченные данные с признаками и категориями, которые машина будет учиться определять по этим признакам. Дальше классифицировать можно что угодно: пользователей по интересам — так делают алгоритмические ленты, статьи по языкам и тематикам — важно для поисковиков, музыку по жанрам — вспомните плейлисты Спотифая и Яндекс. Музыки, даже письма в вашем почтовом ящике.

Раньше все спам-фильтры работали на алгоритме Наивного Байеса. Машина считала сколько раз слово «виагра» встречается в спаме, а сколько раз в нормальных письмах. Перемножала эти две вероятности по формуле Байеса, складывала результаты всех слов и бац, всем лежать, у нас машинное обучение!



Позже спамеры научились обходить фильтр Байеса, просто вставляя в конец письма много слов с «хорошими» рейтингами. Метод получил ироничное название *Отравление Байеса*, а фильтровать спам стали другими алгоритмами. Но метод навсегда остался в учебниках как самый простой, красивый и один из первых практически полезных.

Возьмем другой пример полезной классификации. Вот берёте вы кредит в банке. Как банку удостовериться, вернёте вы его или нет? Точно никак, но у банка есть тысячи профилей других людей, которые уже брали кредит до вас. Там указан их возраст, образование, должность, уровень зарплаты и главное — кто из них вернул кредит, а с кем возникли проблемы.

Да, все догадались, где здесь данные и какой надо предсказать результат. Обучим машину, найдём закономерности, получим ответ — вопрос не в этом. Проблема в том, что банк не может слепо доверять ответу машины, без объяснений. Вдруг сбой, злые хакеры или неадекватный сисадмин решил скриптик исправить.

MNIST

В этом разделе будет использоваться набор данных MNIST (Mixed National Institute of Standard and Technology -- смешанный набор данных Национального института стандартов и технологий США), который содержит 70 000 небольших изображений цифр, написанных от руки учащимися средних школ и служащими. Бюро переписи населения США. Каждое изображение помечено цифрой, которую оно представляет. Этот набор изучался настолько досконально, что его часто называют первым примером ("Hello world") машинного обучения. Всякий раз, когда люди строят новый алгоритм классификации, им любопытно посмотреть, как он будет выполняться на наборе MNIST, и любому, кто осваивает МО, рано или поздно доведётся столкнуться с MNIST.

In [1]:

```
import warnings
warnings.filterwarnings('ignore')

from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
```

Out[1]:

```
dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_names', 'DESCR', 'details', 'u
rl'])
```

Наборы данных, загруженные в Scikit-Learn, обычно имеют похожие словарные структуры, в состав которых входят:

- ключ DESCR, описывающий набор данных;
- ключ data, содержащий массив с одной строкой на образец и одним столбцом на признак;
- ключ target, содержащий массив с метками.

Взглянем на указанные массивы:

In [2]:

```
X, y = mnist['data'], mnist['target']
X.shape
```

Out[2]:

```
(70000, 784)
```

In [3]:

```
y.shape
```

Out[3]:

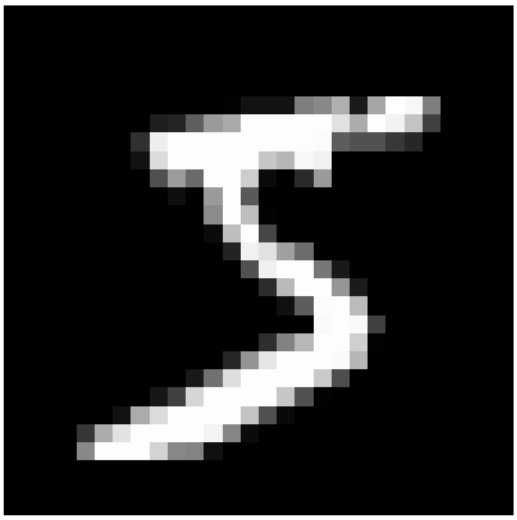
```
(70000,)
```

Существует 70 000 изображений и с каждым изображением связано 784 признака. Дело в том, что каждое изображение имеет размер 28x28 пикселей, а каждый признак просто представляет интенсивность одного пикселя, от 0 (белый) до 255(чёрный). Мы можем посмотреть на одно такое изображение. Достаточно всего лишь извлечь вектор признаков образца, придать ему форму массива 28x28 и отобразить содержимое, используя функцию imshow() из библиотеки matplotlib.

In [4]:

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

some_digit = np.array(X.iloc[0])
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap="gray")
plt.axis('off')
plt.show();
```



In [5]:

```
y[0] # метка
```

Out[5]:

```
'5'
```

Обратите внимание, что метка является строкой. Большинство алгоритмов МО ожидают числа, поэтому приведем к целочисленному типу:

In [6]:

```
y = y.astype(np.uint8)
```

In [7]:

```
X_train, X_test, y_train, y_test = \
       X[:60_000], X[60_000:], y[:60_000], y[60_000:]
```

Обучающий набор уже перетасован и это хорошо, поскольку гарантирует, что все блоки перекрестной проверки будут похожи. Кроме того, некоторые алгоритмы обучения чувствительны к порядку следования обучающих образцов и плохо выполняются, если получют в одной строке много похожих образцов. **Тасование набора данных гарантирует, что подобного не случится.**

Обучение двоичного классификатора

Давайте пока упростим задачу и попробуем опознать только одну цифру -- скажем, 5. Такой "детектор пятёрок" будет примером *двоичного классификатора (binary classifier)*, способного проводить различие между всего лишь двумя классами, "пятерка" и "не пятёрка". Создадим целевые

векторы для этой задачи классификации:

In [8]:

```
y_train_5 = (y_train == 5) # True для всех пятерок
y_test_5 = (y_test == 5) # False для все остальных цифр
```

А теперь возьмём классификатор и обучим его. Хорошим местом для старта будет классификатор на основе метода стохастического градиентного спуска, который применяет класс SGDClassifier из Scikit-Learn. Преимущество такого классификатора в том, что он способен эффективно обрабатывать очень крупные наборы данных. Отчасти это связано с тем, что SGD использует обучающие образцы независимым образом по одному за раз.

In [9]:

```
from sklearn.linear_model import SGDClassifier # импорт класса модели случайного стохастического спуска
sgd_clf = SGDClassifier(random_state=2023) # random_state -- специальный параметр,
                                           # который задается для воспроизводимости результатов
sgd_clf.fit(X_train, y_train_5) # методом fit() обучает модель
```

Out[9]:

```
SGDClassifier(random_state=2023)
```

Показатели эффективности

Перед тем, как строить модели, хорошо бы понять, а как мы будем измерять качество их работы? В разных задачах критичны разные виды ошибок. На основе этих ошибок можно придумать много метрик.

Измерение правильности с использованием перекрестной проверки

Хороший способ оценки модели предусматривает применение перекрестной проверки. **Перекрестная проверка** -- это способ оценки модели машинного обучения и её поведения на независимых данных. При оценке модели имеющиеся в наличии данные разбиваются на k частей. Затем на k−1 частях данных производится обучение модели, а оставшаяся часть данных используется для тестирования. Процедура повторяется k раз; в итоге каждая из k частей данных используется для тестирования. В результате получается оценка эффективности выбранной модели с наиболее равномерным использованием имеющихся данных.

Можно использовать функцию cross_val_score() для оценки модели SGDClassifier с применением перекрестной проверки по k блокам, имея три блока.

In [10]:

```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring='accuracy')
```

Out[10]:

```
array([0.9587 , 0.9572 , 0.96035])
```

Отлично! *Правильность (accuracy*, количество корректных прогнозов) свыше 93% на всех блоках перекрестной проверки? Выглядит поразительно, не так ли? Прежде чем приходить в слишком сильное возбуждение, посмотрим на очень глупый классификатор, который просто относит все изображения без исключения к классу "не пятерка":

In [11]:

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass

    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)

never_5_clf = Never5Classifier()
cross_val_score(never_5_clf, X_train, y_train_5, cv=3,
                scoring='accuracy')
```

Out[11]:

```
array([0.91125, 0.90855, 0.90915])
```

Всё так, правильность выше 90%! Дело в том, что лишь около 10% изображений являются пятерками, поэтому если неизменно полагать, что изображение -- *не* пятерка, то и будете правы примерно в 90% случаев. Нострадамус побеждён.

Это демонстрирует причину, почему правильность (accuracy) обычно не считается предпочтительным показателем эффективности классификаторов, особенно если вы работаете с *ассиметричными наборами данных* (то есть когда одни классы встречаются намного чаще других).

Матрица неточностей

Гораздо лучший способ оценки эффективности классификатора предусматривает просмотр *матрицы неточностей* (confusion matrix). Общая идея заключается в том, чтобы подсчитать сколько раз образцы класса A были отнесены к классу B. Например, для выяснения, сколько раз классификатор путал изображения пятерок с тройками, вы могли бы заглянуть в пятую строку и третий столбец матрицы неточностей.

Для расчета матрицы неточностей первым делом необходимо иметь набор прогнозов, чтобы их можно было сравнивать с фактическими целями. Вы могли бы выработать прогнозы на испытательном наборе, но давайте пока оставим его незатронутым (вспомните, что вы хотите применять испытательный набор только в самом конце проекта после того, как у вас будет классификатор, готовый к запуску). Взамен вы можете использовать функцию cross_val_predict().

In [12]:

```
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Теперь можно получить матрицу неточностей с применением функции confusion_matrix(). Нужно просто передать ей целевые классы (y_train_5) и спрогнозированные классы (y_train_pred).

In [13]:

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_train_5, y_train_pred)
```

Out[13]:

```
array([[53205,  1374],
       [ 1101,  4320]])
```

Каждая строка в матрице представляет собой *фактический класс*, а каждый столбец -- *спрогнозированный класс*. Первая строка матрицы учитывает изображения не пятерок (отрицательный класс (negative class)): 53 205 из них были корректно классифицированы как не пятерки (*истинно отрицательные классификации* (true negative -- TN)), тогда как оставшиеся 1374 были ошибочно классифицированы как пятерки (ложноположительные классификации (false positive -- FP)). Вторая строка матрицы учитывает изображения пятерок (положительный класс (positive class)): 1101 были ошибочно распознаны как не пятерки (ложноотрицательные классификации (false negative -- FN)), в то время как оставшиеся 4320 были корректно классифицированы как пятерки (*истинно положительные классификации* (true positive -- TP)).

Безупречный классификатор имел бы только истинно положительные и истинно отрицательные классификации, так что его матрица неточностей содержала бы ненулевые значения только на своей главной диагонали (от левого верхнего до правого нижнего угла):

In [14]:

```
y_train_perfect_prediction = y_train_5 # притворимся, что
                                       # достигли безупречности
confusion_matrix(y_train_5, y_train_perfect_prediction)
```

Out[14]:

```
array([[54579,     0],
       [    0,  5421]])
```

Матрица неточностей даёт много информации, но иногда можно предпочесть более сжатую метрику. Интересной метрикой такого рода является аккуратность положительных классификаций; она называется **точностью** (precision) классификатора.

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

Точность отражает то, насколько мы можем доверять алгоритму, если он спрогнозировал единичку. TP -- это количество истинно положительных классификаций, а FP -- количество ложноположительных классификаций.

Тривиальный способ добиться совершенной точности заключается в том, чтобы вырабатывать одну полложительную классификацию и удостоверяться в её корректности (точность = 1/1 = 100%). Но это было бы не особенно полезно, так как классификатор игнорировал бы все, кроме одного положительного образца. Таким образом, точность обычно используется наряду с ещё одной метрикой под названием **полнота** (recall), также называемой чувствительностью или долей *истинно положительных классификаций* (TPR -- true positive rate): коэффициент положительных образцов, которые корректно обнаружены классификатором.

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

Показывает, как много объектов первого класса наш алгоритм находит.

Если запутались с матрицей, то вот наглядно:

Точность и полнота

В библиотеке Scikit-Learn предлагается несколько функций для подсчёта метрик классификаторов, в том числе точности и полноты.

In [15]:

```
from sklearn.metrics import precision_score, recall_score
precision_score(y_train_5, y_train_pred)
```

Out[15]:

```
0.7586933614330874
```

In [16]:

```
recall_score(y_train_5, y_train_pred)
```

Out[16]:

```
0.7969009407858328
```

Теперь этот детектор пятерок не выглядит настолько блестящим, как при просмотре его правильности. Его заявление о том, что изображение представляет пятерку, корректно только 75.8% времени. Кроме того, он обнаруживает только 79.6% пятерок.

Точность и полноту часто удобно объединять в единственную метрику под названием F-мера, особенно если вам нужен простой способ сравнения двух классификаторов. F-мера -- это *среднее гармоническое точности и полноты*.

$$F = \frac{1}{\beta\frac{1}{P} + (1-\beta)\frac{1}{R}}$$

Параметр β в данном случае определяет вес точности в метрике. $\beta = 1$ вносит точность и полноту с одинаковыми весами. Изменяя его, можно отдавать предпочтения либо одному либо другому.

В то время как обычное среднее трактует все значения одинаково, среднее гармоническое придаёт низким значениям больший вес. В результате классификатор получит высокую F-меру, только если высокими являются и полнота, и точность.

In [17]:

```
from sklearn.metrics import f1_score
f1_score(y_train_5, y_train_pred)
```

Out[17]:

```
0.777327935222672
```

F-мера поддерживает классификаторы, которые имеют подобные точность и полноту. Это не всегда то, что вы хотите: в одних контекстах вы главным образом заботитесь о точности, а в других -- фактически о полноте. Например, если вы обучаете классификатор выявлять видеоролики, безопасные для просмотра детьми, тогда наверняка отдадите предпочтение классификатору, который отклоняет многие хорошие видеоролики (низкая полнота), но сохраняет только безопасные видеоролики (высокая точность), а не классификатору, имеющему более высокую полноту, но позволяющему проникнуть в продукт нескольким по-настоящему нежелательным видеороликам.

К сожалению, невозможно получить то и другое одновременно: увеличение точности снижает полноту и наоборот. Это называется соотношением точность/полнота.

Соотношение точность/полнота

Чтобы понять данное соотношение, посмотрим, каким образом класс SGDClassifier принимает свои решения по классификации. Для каждого образца он вычисляет сумму очков на основе **функции принятия решения**. Если сумма очков больше какого-то порогового значения, образец приписывается положительному классу, а иначе -- отрицательному классу.

Библиотека Scikit-Learn не позволяет напрямую устанавливать порог, но предоставляет доступ к суммам очков, управляющих решением, которые она применяет для выработки прогнозов. Вместо вызова метода predict() классификатора можно вызвать метод decision_function(), который возвращает сумму очков для каждого образца, и затем вырабатывать прогнозы на основе этих сумм очков, используя любой желаемый порог:

In [18]:

```
y_scores = sgd_clf.decision_function([some_digit])
y_scores
```

Out[18]:

```
array([767.13846228])
```

In [19]:

```
threshold = 0
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

Out[19]:

```
array([ True])
```

Класс SGDClassifier применяет порог, равный 0, так что предыдщуий код возвращает такой же результат, как и метод predict(), т. е. True. А если поднять порог?

In [20]:

```
threshold = 8_000
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

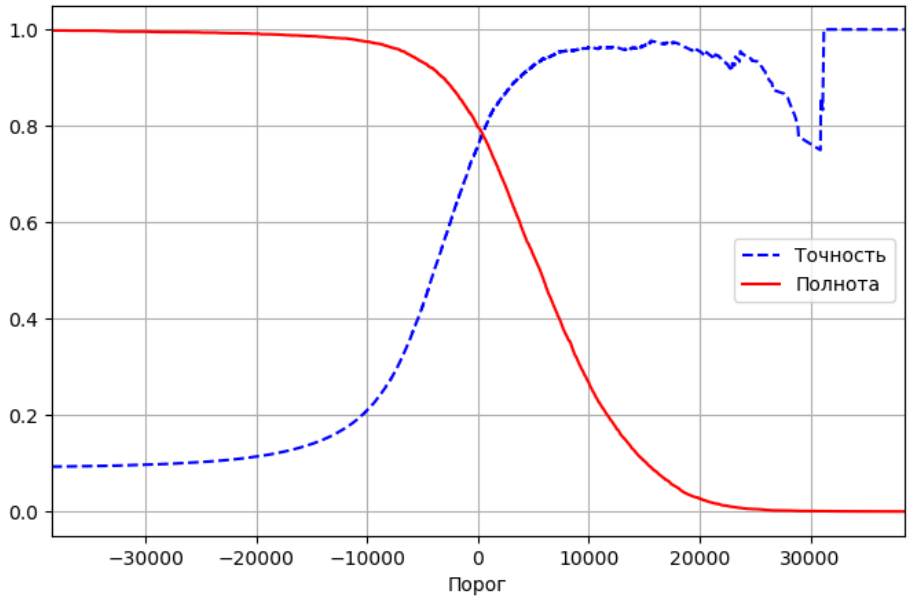
Out[20]:

```
array([False])
```

Результат подтверждает, что поднятие порога снижает полноту. Изображение фактически представляет пятерку и классификатор опознаёт её, когда порог равен 0, но не замечает, когда порог увеличивается до 8_000. Как принять решение, какой порог использовать? Для начала -- получить суммы очков всех образцов в обучающем наборе, снова применяя функцию cross_val_predict(), но на этот раз с указанием того, что вместо прогнозов она должна возвратить суммы очков, управляющие решением:

In [21]:

```
from sklearn.metrics import precision_recall_curve
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method='decision_function')
# имея суммы очков, можно вычислить точность и полноту для всех возможных порогов
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
# имея полученные значения, можно их графически отобразить
plt.figure(figsize=(8, 5))
plt.plot(thresholds, precisions[:-1], 'b--', label='Точность')
plt.plot(thresholds, recalls[:-1], 'r-', label='Полнота')
plt.grid(True)
plt.xlabel('Порог')
plt.xlim(-thresholds.max(), thresholds.max())
plt.legend()
plt.show();
```



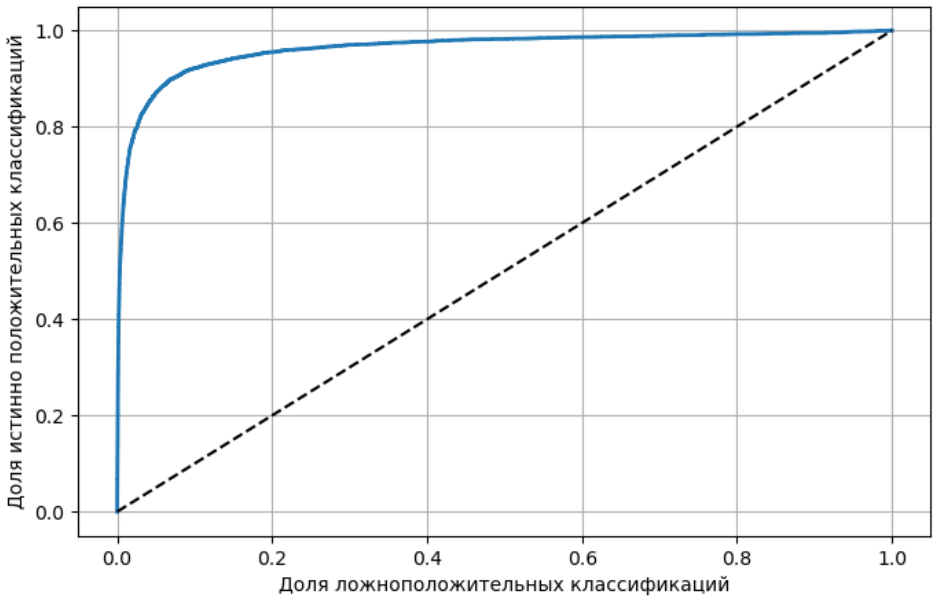
Кривая ROC

Кривая *рабочей характеристики приемника* (ROC) представляет собой ещё один распростанённый инструмент, применяемый с двоичными классификаторами. Она очень похожа на кривую точности-полноты, но вместо вычерчивания точности в сравнении с полнотой кривая ROC изображает долю истинно положительных классификаций по отношению к доле ложноположительных классификаций (false positive rate -- FPR). Доля FPR -- это пропорция отрицательных образцов, которые были некорректно классифицированы как положительные. Она равна единице минус *доля истинно отрицательных классификаций* (TNR), представляющая собой пропорцию отрицательных образцов, которые были корректно классифицированы как отрицательные. Доля TNR также называется *специфичностью*. Следовательно, кривая ROC изображает полносту в сравнении с 1 минус специфичность.

Чтобы вычертить кривую ROC, сначала необходимо вычислить TPR и FPR для разнообразных значений порога, используя функцию roc_curve():

In [22]:

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
# затем можно вычертить сам график кривой
plt.figure(figsize=(8, 5))
plt.plot(fpr, tpr, linewidth=2)
plt.plot([0, 1], [0, 1], 'k--') # пунктирная диагональ
plt.xlabel('Доля ложноположительных классификаций')
plt.ylabel('Доля истинно положительных классификаций')
plt.grid(True)
plt.show();
```



И снова имеется соотношение: чем выше полнота (TPR), тем больше классификатор дает ложноположительных классификаций (FPR). Пунктирная линия представляет ROC кривую чисто случайного классификатора; хороший классификатор отстоит от указанной линии насколько возможно далеко (стремясь к левому верхнему углу).

Один из способов сравнения классификаторов предусматривает измерение *площади под кривой* (area under the curve -- AUC). Безупречный классификатор будет иметь площадь под кривой ROC (ROC AUC), равную 1, тогда как чисто случайный классификатор -- площадь, равную 0.5.

In [23]:

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
```

Out[23]:

```
0.9633563460298149
```

Давайте теперь обучим классификатор RandomForestClassifier и сравним его кривую ROC и показатель ROC AUC с таковыми у классификатора SGDCLassifier. Первым делом понадобится получить показатели для каждого образца в обучающем наборе. Но из-за особенностей работы класс RandomForestClassifier не имеет метода decision_function(). Взамен у него есть метод predict_proba(). Классификаторы Scikit-Learn обычно имеют тот или другой метод или оба.

Метод predict_proba() возвращает массив, который содержит строку на образец и столбец на класс, содержащий вероятность того, что заданный образец принадлежит заданному классу (скажем, 70%-й шанс, что изображение представляет пятерку):

In [24]:

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(random_state=2023)
y_probs_forest = cross_val_predict(forest_clf, X_train, y_train_5,
                                   cv=3, method='predict_proba')
```

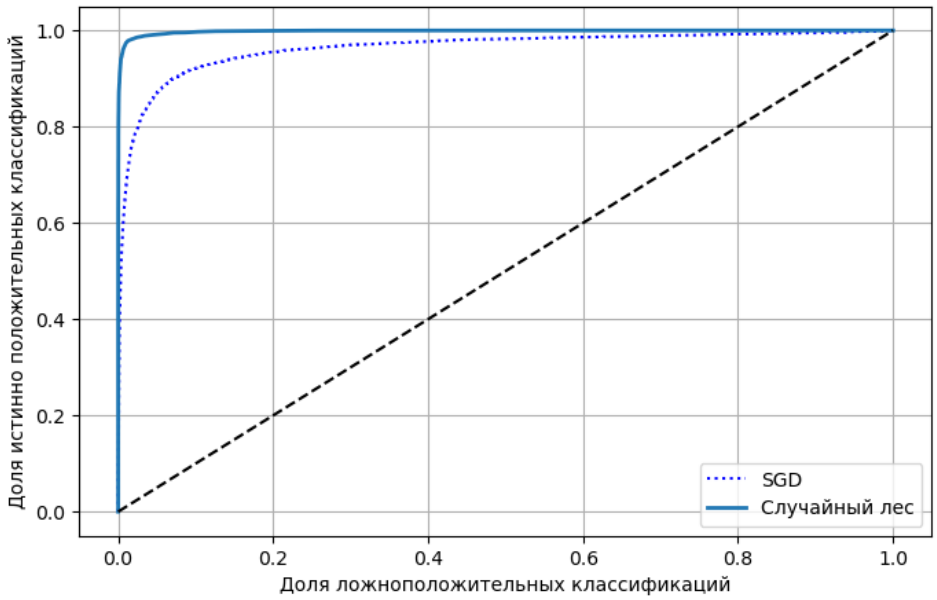
Функция roc_curve() ожидает метки и показатели, но вместо показателей ей можно предоставить вероятности классов. Применим вероятность положительного класса в качестве показателя:

In [25]:

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # пунктирная диагональ
    plt.xlabel('Доля ложноположительных классификаций')
    plt.ylabel('Доля истинно положительных классификаций')
    plt.grid(True)
```

In [26]:

```
y_scores_forest = y_probs_forest[:, 1]
fpr_forest, tpr_forest, \
      thresholds_forest = roc_curve(y_train_5, y_scores_forest)
# можно вычертить и roc-кривую
plt.figure(figsize=(8, 5))
plt.plot(fpr, tpr, 'b:', label='SGD')
plot_roc_curve(fpr_forest, tpr_forest, 'Случайный лес')
plt.legend(loc='lower right')
plt.show();
```



На рисунке выше видно, что ROC-кривая случайного леса выглядит лучше, чем кривая SGDClassifier.

К настоящему моменту вам известно, как обучать двоичные классификаторы, выбирать подходящую метрику для своей задачи, оценивать классификаторы с использованием перекрестной проверки, устанавливать соотношение точность/полнота, которое соответствует имеющимся потребностям, и сравнивать разнообразные модели с применением кривых ROC и показателей ROC AUC. А теперь попробуем распознать больше, чем только пятерки.

Многоклассовая классификация

В то время как двоичные классификаторы проводят различие между двумя классами, *многоклассовые классификаторы*, также называемые *полиномиальными классификаторами* могут различать более двух классов.

Некоторые алгоритмы классификации (такие как SGC, классификаторы на основе случайных лесов или наивные байесовские классификаторы) изначально способны обрабатывать несколько классов. Другие (вроде классификаторов на базе логистической регрессии или методов опорных векторов) являются строго двоичными классификаторами. Однако существуют разнообразные стратегии, которые можно использовать для проведения многоклассовой классификации с помощью множества двоичных классификаторов.

Один из способов создания системы, которая способна систематизировать изображения цифр в 10 классов (от 0 до 9), предполагает обучение 10 двоичных классификаторов, по одному для каждой цифры (детектор нулей, детектор единиц, детектор двоек и так далее). Затем, когда необходимо классифицировать изображение, вы получаете из каждого классификатора сумму баллов решения для данного изображения и выбираете класс, чей классификатор выдал наивысший балл. Такой приём называется стратегией "один против остальных" (one-versus-the-rest, OvR), также называемой "один против всех" (one-versus-all, OvA).

Другая стратегия предусматривает обучение двоичного классификатора для каждой пары цифр: одного для проведения различия между нулями и единицами, одного для разлчичения нулей и двоек, одного для единиц и двоек и так далее. Это называется стратегией "один против одного" (one-versus-one, OvO). Если есть N классов, то понадобится обучить N * (N-1)/2 классификаторов. Для задачи с набором данных MNIST получается, что нужно обучить 45 двоичных классификаторов. Главное преимущество стратегии OvO в том, что каждый классификатор нуждается в обучении только на части обучающего набора для двух классов, которые он обязан различать.

Некоторые алгоритмы плохо масштабируются с ростом размера обучающего набора. Для таких алгоритмов стратегия OvO предпочтительнее, потому что обучить много классификаторов на небольших обучающих наборах быстрее, чем несколько классификаторов на крупных обучающих наборах. Тем не менее, для большинства алгоритмов двоичной классификации предпочтительной будет стратегия OvR.

Библиотека Scikit-Learn обнаруживает попытку использования алгоритма двоичной классификации для задачи многоклассовой классификации и в зависимости от алгоритма автоматически инициирует стратегию OvR или OvO. Давайте проверим сказанное на классификаторе методом опорных векторов, применив класс sklearn.svm.SVC:

In [27]:

```
from sklearn.svm import SVC
svm_clf = SVC()
svm_clf.fit(X_train, y_train)
svm_clf.predict([some_digit])
```

Out[27]:

```
array([5], dtype=uint8)
```

Вызвав метод decision_function(), вы увидите, что он возвращает 10 сумм баллов на образец (вместо только одного), т. е. одну на класс:

In [28]:

```
some_digit_scores = svm_clf.decision_function([some_digit])
some_digit_scores
```

Out[28]:

```
array([[ 1.72501977,  2.72809088,  7.2510018 ,  8.3076379 , -0.31087254,
         9.3132482 ,  1.70975103,  2.76765202,  6.23049537,  4.84771048]])
```

Самый высокий балл действительно соответствует классу 5:

In [29]:

```
np.argmax(some_digit_scores)
```

Out[29]:

```
5
```

Если вы хотите заставить Scikit_Learn применять стратегию OvO или OvR, тогда можете использовать классы OneVsOneClassifier или OneVsRestClassifier. Просто создайте экземпляр класса, передав его конструктору классификатор (который даже не обязан быть двоичным). Например, показанный ниже код создает многоклассовый классификатор, применяющий стратегию OvR, на основе SVC:

In [30]:

```
from sklearn.multiclass import OneVsRestClassifier
ovr_clf = OneVsRestClassifier(SVC())
ovr_clf.fit(X_train, y_train)
ovr_clf.predict([some_digit])
```

Out[30]:

```
array([5], dtype=uint8)
```

Анализ ошибок

В реальном проекте вы бы следовали шагам контрольного перечня для проекта машинного обучения. Вы бы исследовали варианты подготовки данных, обробовали множество моделей (включив в окончательный список лучшие модели и точно настроив их гиперпараметры с применением GridSearchCV), а также обеспечили максимально возможную автоматизацию. Здесь мы будем предполагать, что вы уже отыскали перспективную модель и хотите найти способы её усовершенствования. Одним таким способом будет анализ типов ошибок, допускаемых моделью.

Первым делом посмотрите матрицу неточностей. Вам необходимо выработать прогнозы, используя функцию cross_val_predict(), и затем вызвать функцию confusion_matrix(), как вы поступали ранее:

In [31]:

```
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train, cv=3)
conf_mx = confusion_matrix(y_train, y_train_pred)
conf_mx
```

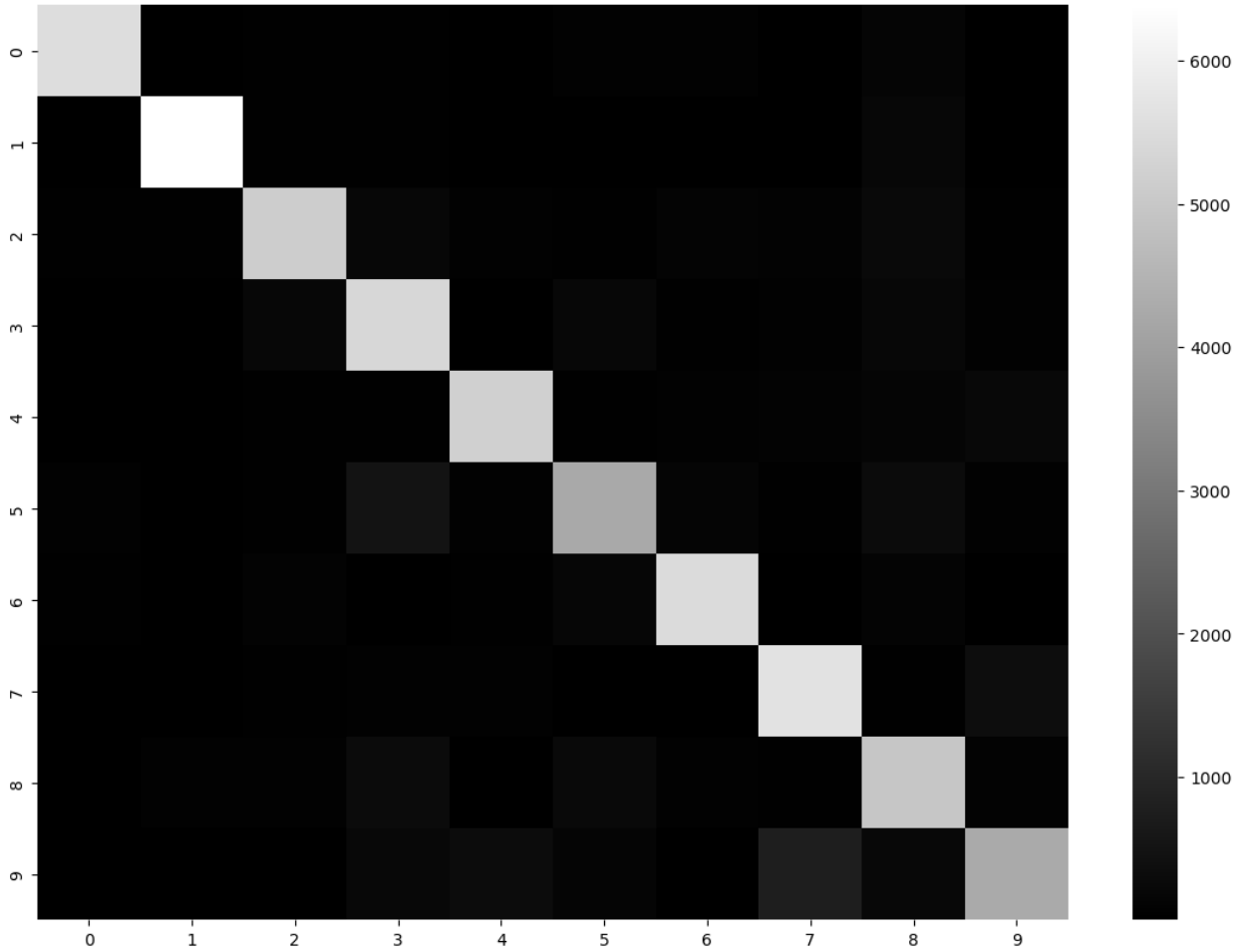
Out[31]:

```
array([[5528,    1,   40,   45,   13,   66,   59,    8,  148,   15],
       [   1, 6389,   50,   35,   10,   24,   16,   17,  189,   11],
       [  27,   44, 5126,  184,   67,   43,  110,   91,  239,   27],
       [  10,   18,  176, 5389,   13,  176,   36,   69,  176,   68],
       [  12,   24,   35,   31, 5202,   48,   54,   77,  138,  221],
       [  52,   19,   48,  498,   64, 4219,  148,   33,  276,   64],
       [  27,   12,   76,    5,   38,  156, 5484,    7,  110,    3],
       [  22,   20,   45,   63,   66,   21,    5, 5643,   50,  330],
       [  22,   70,   58,  280,   23,  239,   57,   44, 4958,  100],
       [  17,   19,   24,  208,  309,  126,    3,  771,  216, 4256]])
```

Получилось много чисел. Часто гораздо удобнее смотреть на представление матрицы неточностей в виде изображения, для чего применяется функция matshow() из matplotlib или heatmap() из библиотеки seaborn.

In [32]:

```
import seaborn as sns
plt.rcParams["figure.figsize"] = [14, 10]
sns.heatmap(conf_mx, fmt='.1f', cmap=plt.cm.gray);
```



Итоговая матрица неточностей выглядит достаточно неплохо, так как большинство изображений расположено на глвной диагонали, что говорит об их корректной классификации. Пятерки лишь слегка темнее других цифр; это могло бы означать, что в наборе данных имеется меньшее количество изображений пятерок или классификатор работает на пятерках не настолько хорошо, как на других цифрах. В действительности вы можете проверить оба предположения.

Давайте сконцентрируем график на ошибках. Прежде всего, **каждое значение в матрице неточностей понадобится разделить на число изображений в соответствующем классе, чтобы можно было сравнивать частоты ошибок вместо их абсолютных количеств (что сделало бы изобилующие ошибками классы несправедливо плохими).**

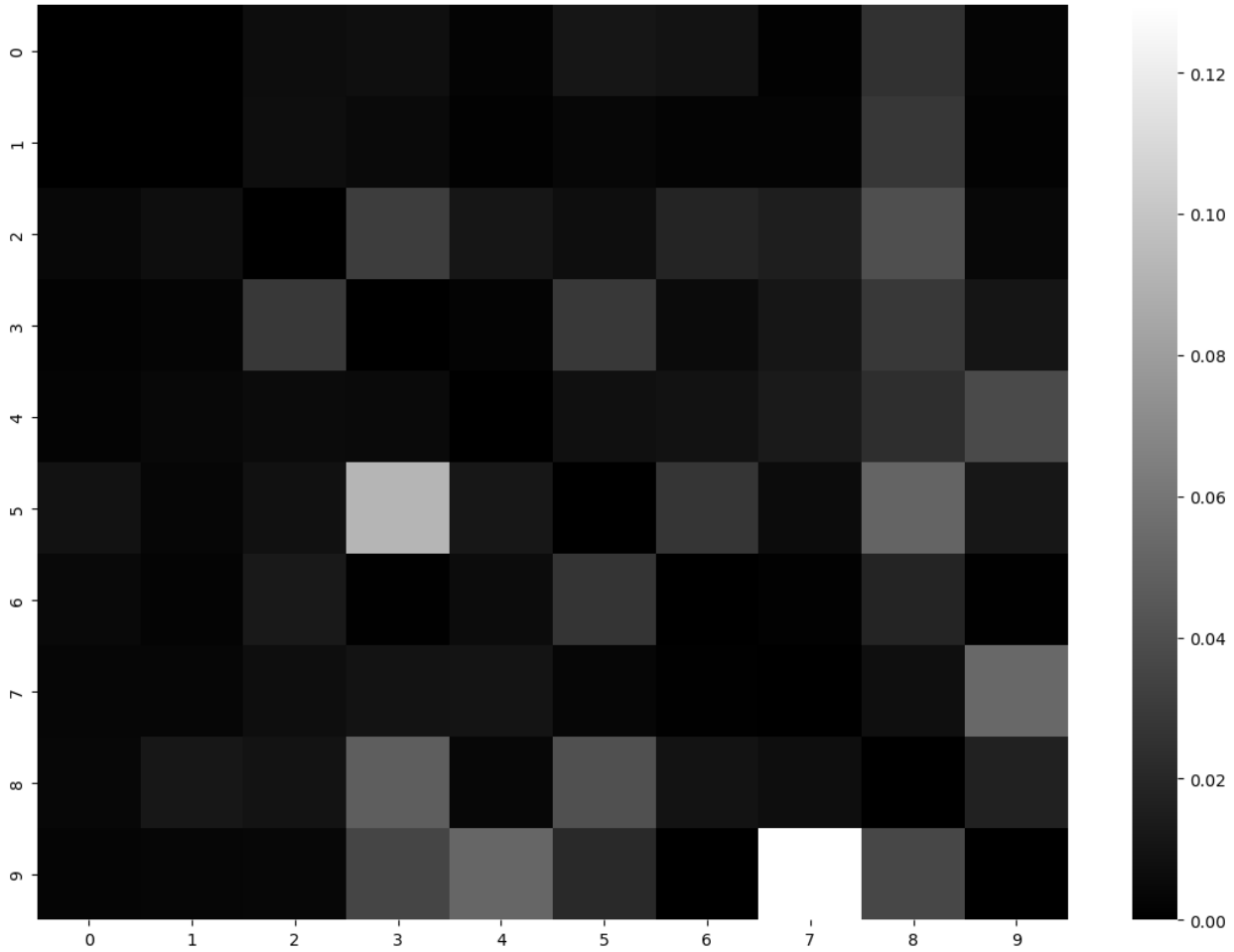
In [33]:

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

Заполним диагональ нулями, сохранив только ошибки, и вычертим результирующий график.

In [34]:

```
np.fill_diagonal(norm_conf_mx, 0)
sns.heatmap(norm_conf_mx, fmt='.1f', cmap=plt.cm.gray);
```



Здесь можно явно наблюдать типы ошибок, которые допускают ккласификатор. Вспомните, что строки представляют фактические классы, а столбцы -- спрогнозированные классы. Столбец для класса 8 выглядит довольно светлым, указывая на то, что многие изображения были неправильно классифицированы как восьмерки. Однако строка для класса 8 не так уж плоха и сообщает о том, что фактические восьмерки в целом надлежащим образом были классифицированы как восьмерки. Как видите матрица неточностей необязательно симметрична.

Анализ матрицы неточностей часто подсказывает способы улучшения классификатора. Глядя на этот график, кажется, что усилия должны быть направлены на снижение ложной классификации восьмерок. Скажем, можно быо бы накопить больше обучающих данных для цифр, которые выглядят походими на восьмерки (но не являются ими), чтобы классификатор смог научиться отличать их от настоящих восьмерок. Или же вы могли бы сконструировать новые признаки, которые помогали бы классификатору -- например, реализовать алгоритм для подсчета колчичества замкнутых контуров (8 имеет два таких контура, 6 -- один, 5 -- ни одного). Либо вы могли бы предварительно обработать изображение (скажем, используя библиотеку Scikit-Learn или OpenCV), чтобы паттерны вроде замкнутых контуров стали более заметными.

Многозначная классификация

До сих пор каждый образец всегда назначался только одному классу. В ряде случае может быть желательно, чтобы классификатор выдавал множество классов для каждого образца. Возьмем классификатор для распознавания лиц: что он должен делать в ситуации, если на одной фотографии распознано несколько людей? Конечно, классификатор обязан прикреплять по одной метке на распознанную особу. Пусть классификатор был обучен распознавать три лица, Алису, Боба и Чарли. Позже, когда он увидит фотографию Алисы и Чарли, то должен выдать [1, 0, 1] (т.е. "Алиса есть, Боб отсутствует, Чарли есть"). Такая система классификации, выдающая множество двоичных меток, называется системой *многозначной классификации*.

Прямо сейчас не будем вдаваться распознавание лиц глубоко, но в иллюстративных целях сделаем другой пример:

In [35]:

```
from sklearn.neighbors import KNeighborsClassifier
y_train_large = (y_train >= 7)  # отбираем из всех цифр только самые большие
y_train_odd = (y_train % 2 == 1) # отбираем только нечетные
y_multilabel = np.c_[y_train_large, y_train_odd] # кодируем в виде массива, который будет содержать пары True/False
                                                # первая - большая или маленькая цифра, вторая - четность

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

Out[35]:

```
KNeighborsClassifier
KNeighborsClassifier()
```

In [36]:

```
y_multilabel
```

Out[36]:

```
array([[False,  True],
       [False, False],
       [False, False],
       ...,
       [False,  True],
       [False, False],
       [ True, False]])
```

In [37]:

```
# вырабатываем прогноз
knn_clf.predict([some_digit])
```

```
Exception ignored on calling ctypes callback function: <function _ThreadpoolInfo._find_modules_with_dl_iter
ate_phdr.<locals>.match_module_callback at 0x7efc99843b00>
Traceback (most recent call last):
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 400, in
match_module_callback
    self._make_module_from_path(filepath)
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 515, in
_make_module_from_path
    module = module_class(filepath, prefix, user_api, internal_api)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 606, in
__init__
    self.version = self.get_version()
                   ^^^^^^^^^^^^^^^^^^
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 646, in
get_version
    config = get_config().split()
             ^^^^^^^^^^^^^^^^^^
AttributeError: 'NoneType' object has no attribute 'split'
```

Out[37]:

```
array([[False,  True]])
```

И прогноз правильный! Цифра 5 на самом деле небольшая и нечетная.

Существует много способов оценки многозначеного классификатора и выбор надлежйшей метрики зависит от проекта. Один из подходов заключается в том, чтобы оперделить для каждой отдельной метки меру F1 (или любую другую метрику двоичных классификаторов, которые обсуждались ранее) и затем просто подсчитать среднюю сумму очков. Следующий код вычисляет F-меру по всем меткам:

In [38]:

```
y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3)
f1_score(y_multilabel, y_train_knn_pred, average='macro')
```

```
Exception ignored on calling ctypes callback function: <function _ThreadpoolInfo._find_modules_with_dl_iter
ate_phdr.<locals>.match_module_callback at 0x7efc996d0540>
Traceback (most recent call last):
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 400, in
match_module_callback
    self._make_module_from_path(filepath)
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 515, in
_make_module_from_path
    module = module_class(filepath, prefix, user_api, internal_api)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 606, in
__init__
    self.version = self.get_version()
                   ^^^^^^^^^^^^^^^^^^
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 646, in
get_version
    config = get_config().split()
             ^^^^^^^^^^^^^^^^^^
AttributeError: 'NoneType' object has no attribute 'split'
Exception ignored on calling ctypes callback function: <function _ThreadpoolInfo._find_modules_with_dl_iter
ate_phdr.<locals>.match_module_callback at 0x7efc996d0360>
Traceback (most recent call last):
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 400, in
match_module_callback
    self._make_module_from_path(filepath)
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 515, in
_make_module_from_path
    module = module_class(filepath, prefix, user_api, internal_api)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 606, in
__init__
    self.version = self.get_version()
                   ^^^^^^^^^^^^^^^^^^
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 646, in
get_version
    config = get_config().split()
             ^^^^^^^^^^^^^^^^^^
AttributeError: 'NoneType' object has no attribute 'split'
Exception ignored on calling ctypes callback function: <function _ThreadpoolInfo._find_modules_with_dl_iter
ate_phdr.<locals>.match_module_callback at 0x7efc996d0400>
Traceback (most recent call last):
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 400, in
match_module_callback
    self._make_module_from_path(filepath)
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 515, in
_make_module_from_path
    module = module_class(filepath, prefix, user_api, internal_api)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 606, in
__init__
    self.version = self.get_version()
                   ^^^^^^^^^^^^^^^^^^
  File "/home/agat.local/s.bulganin/anaconda3/lib/python3.11/site-packages/threadpoolctl.py", line 646, in
get_version
    config = get_config().split()
             ^^^^^^^^^^^^^^^^^^
AttributeError: 'NoneType' object has no attribute 'split'
```

Out[38]:

```
0.976410265560605
```

Однако здесь предполагается равная степень важности всех меток, что может быть и не так. В частности, если фотографий Алисы намного больше, чем Боба и Чарли, тогда вы можете поставить больший вес оценке классификатора на фотографиях Алисы. Простой вариант -- назначить каждой метке вес, равный её поддержке (т. е. количеству образцов с такой целевой меткой), для чего в предыдущем коде необходимо просто установить average='weighted'.

Упражнения

1. Попробуйте построить классификатор для набора данных MNIST, который достигает 97%-ной правильности (accuracy) на испытательном наборе. Подсказка: с этой задачей неплохо справится KNeighborsClassifier ; вам просто нужно найти подходящие значения гиперпараметров (воспользуйтесь решетчатым поиском для гиперпараметров weights и n_neighbors).
   Класс GridSearchCV (решетчатый поиск) (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html);
   Класс KNeighborsClassifier (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html).
2. Напишите функцию, которая может смещать изображение MNIST в любом направлении (влево, вправо, вверх или вниз) на один пиксель. Для этого вы можете применить функцию **shift()** из модуля **scipy.ndimage.interpolation**. Затем для каждого изображения в обучающем наборе создайте четыре смещенных копии (по одной на каждое направление) и добавьте их в обучающий набор. Наконец, обучите на этом расширенном обучающем наборе свою наилучшую модель и измерьте её правильность (accuracy) на испытательном наборе. Вы должны заметить, что модель стала работать лучше. Такой приём искусственного увеличения данных называется дополнением данных (*data augmentation*) или расширением обучающего набора.

Дополнительные материалы:

- http://www.machinelearning.ru/wiki/index.php?title=Метод_ближайших_соседей (http://www.machinelearning.ru/wiki/index.php?title=%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%B1%D0%BB%D0%B8%D0%B6%D0%B0%D0%B9%D1%88%D0%B8%D1%85_%D1%81%D0%BE
- http://scikit-learn.org/stable/modules/neighbors.html (http://scikit-learn.org/stable/modules/neighbors.html)
- http://bigdata-madesimple.com/k-nearest-neighbors-curse-dimensionality-python-scikit-learn/ (http://bigdata-madesimple.com/k-nearest-neighbors-curse-dimensionality-python-scikit-learn/)
- https://www.kaggle.com/skalskip/iris-data-visualization-and-knn-classification (https://www.kaggle.com/skalskip/iris-data-visualization-and-knn-classification)