

Машинное обучение

Никита Трофимов

18 сентября 2022 г.

1 Возможность компьютеров обучаться

1.1 Три разных типа машинного обучения

Есть три разных типа м.о.:

- Supervised learning: labeled data, direct feedback, predict outcome/future
- Unsupervised learning: no labels/targets, no feedback, find hidden structure in data
- Reinforcement learning: decision process, reward system, learn series of actions

В качестве примера **supervised learning** можно привести разделение на спам/не спам сообщений. Если выход дискретный, то это **classification** task, если сигнал непрерывный, то это **regression** task. Разделение (граница) между классами называется **decision boundary**. Пример multiclass classification – это распознавание цифр, букв.

Пример **regression analysis** – балл студентов на экзамене в зависимости от времени. Нам даны features – получаем на выходе непрерывные target variables. Идея **linear regression** – провести линию так, чтобы минимизировать (чаще всего) средний квадрат расстояния от прямой до наших данных.

Теперь про **reinforcement learning**. Цель – разработать систему (агента), который улучшается на основании взаимодействий со средой. На выходе мы получаем **reward signal**, но это не классификация. В итоге мы максимизируем награду через **exploratory trial-and-error approach** (исследовательский метод проб и ошибок) или через **deliberative planning** (совещательное планирование). Пример: игра в шахматы, где среда – состояния доски, reward – lose or win.

Пару слов о **unsupervised learning**. Здесь мы имеем дело с данными без меток или с данными неопределённой структуры. Наша задача – выделить значимую информацию. **Clustering** (unsupervised classification) – специальная техника для разбиения данных на **clusters**, некоторые группы с общими свойствами. Ещё одно важное ответвление – **dimensionality reduction**. Часто мы имеем данные очень больших размерностей, что увеличивает время работы и занимает память. Этим алгоритмом мы сжимаем данные в меньшее подпространство, выделяя самое главное. То есть мы убираем шум, который мешает точно предсказывать. Это можно применить для визуализации в 2D и 3D.

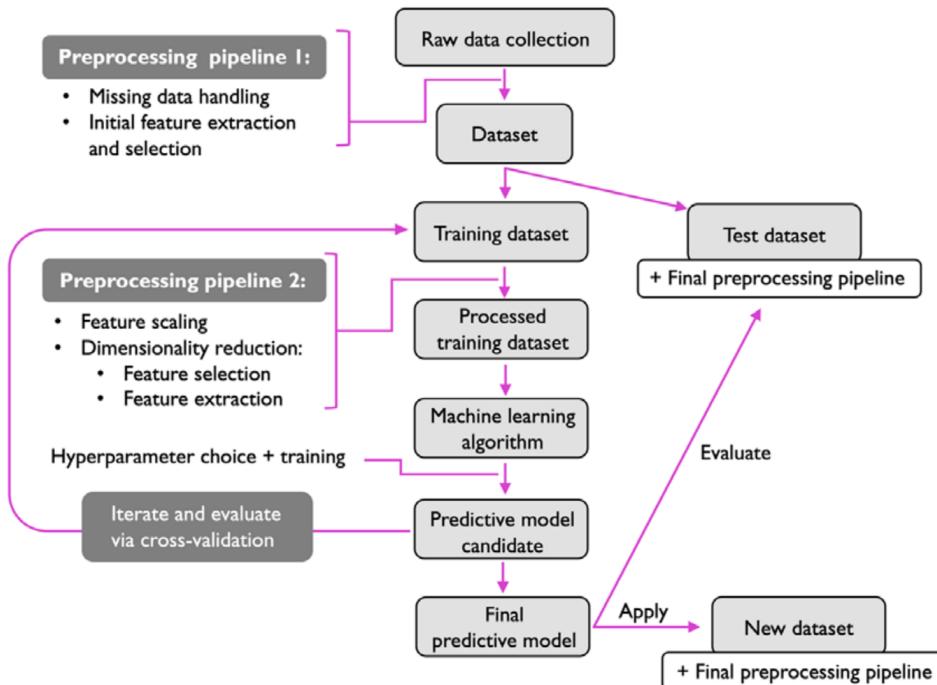
1.2 Базовая терминология и нотации

Датасет данных (design matrix):

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(n)} & x_2^{(n)} & x_3^{(n)} & x_4^{(n)} \end{bmatrix} \in \mathbb{R}^{n \times 4}$$

Важно, что мы будем считать *вектором* именно столбец. Важная терминология: training example, training, feature (x), target (y), loss function (иногда: loss - для одной строки, cost - среднее для всего датасета).

1.3 План обучения системы



Часто алгоритмам для хорошей работы нужно сначала обработать данные. Например: привести все фичи в отрезок $[0, 1]$ или сделать **standard normal distribution** с нулевым средним значением и единичной дисперсией. Если датасет имеет низкий signal-to-noise ratio (много шума, мало полезных данных), то нужно сделать dimensional reduction. Важно определить метрику, часто – это **measure performance**, процент правильных ответов. Есть техника **cross-validation**, когда мы делим данные на training и validation датасеты. Тренируемся на training, выбираем лучшую модель на validation (по метрике), итоговый результат по test data (например, в процентах). Также применяются hyperparameter optimization techniques, когда дефолтные значения алгоритма не подходят, например learning rate. **generalization error** – ошибка на тестовых данных, важно, что все feature scaling должны применяться и на будущих данных, правда их значения взяты с тренировочных (μ, σ, σ^2) .

1.4 Python для машинного обучения

Сейчас мы будем использовать scikit-learn library, дальше для нейронных сетей PyTorch library. Важно, что используется Python версии ≥ 3.9 . Для пакетов нужно использовать pip installer program. Также можно использовать Anaconda Python distribution. Ещё из библиотек мы будем использовать NumPy, а также pandas, что является надстройкой над NumPy. Matplotlib нужен нам для визуализации данных. Версию можно проверить так:

```
import numpy  
numpy.__version__
```

2 Простые алгоритмы классификации

2.1 Нейроны

Разберёмся что такое **artificial neuron** и как он работает. Определим decision function $\sigma(z)$, вектор весов w , тогда $z = w_1x_1 + \dots + w_mx_m$, где x – входные значения. Скоро мы будем рассматривать **perceptron algorithm**, поэтому напишем функцию для него. В нём используется вариант **unit step function**:

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

θ – удерживаемое значение. Это можно упростить, определив **bias unit** $b = -\theta$. Тогда: $z = w_1x_1 + \dots + w_mx_m + b = w^T x + b$. Теперь удерживаемое значение можно поставить на 0. Научимся обучать нейрон. Суть **perceptron algorithm** такая:

- Назначим веса и bias unit нулями или маленькими случайными числами
- Для каждого $x^{(i)}$ посчитаем выходное значение $\hat{y}^{(i)}$, обновим веса и bias unit ($\Delta b, \Delta w_j$)

Напишем формулу для $\Delta b, \Delta w_j$: $\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}, \Delta b = \eta(y^{(i)} - \hat{y}^{(i)})$. Важно обновлять значения одновременно – для всего вектора w сразу. η – **learning rate** – это как бы ширина шага алгоритма. Если нет чёткой линейной границы между двумя классами, то алгоритм сходиться не будет, поэтому, чтобы он не обучался вечно зададим кол-во эпох.

2.2 Реализация perceptron algorithm

Далее в реализации мы выбираем для начальных значений весов маленькие числа около нуля. Это делается потому что иначе η не будет влиять на направление вектора, лишь только на его длину, а также при изменении η не будет меняться decision boundary, что плохо. Наш perceptron algorithm – это бинарный классификатор, но он может быть обобщён на несколько меток с помощью **one-versus-all (OvA)** техники. Пусть у нас есть n меток. Давайте обучим n моделей, где i – *a*я метка считается 1, а остальные считаются 0. Тогда из n моделей выберем ту, которая больше всего уверена в себе.

Напишем код:

```
import numpy as np

# зададим наше API
class Perceptron:
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    # X: shape = [n_examples, n_features]
    # y: shape = [n_examples]
    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        # a normal (Gaussian) distribution
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.float_(0.)
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_ += update * xi
                self.b_ += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    # произойдёт умножение каждой строки на веса
    # как скалярное произведение
    def net_input(self, X):
        return np.dot(X, self.w_) + self.b_

    def predict(self, X):
        return np.where(self.net_input(X) >= 0.0, 1, 0)

    # пример того, почему нельзя просто присвоить 0 нашему вектору w_
    v1 = np.array([1, 2, 3])
    v2 = 0.5 * v1
    np.arccos(v1.dot(v2) / (np.linalg.norm(v1) * np.linalg.norm(v2)))
    # output: 0.0

    # теперь обучим
    import os
    import pandas as pd
    # вместо ссылки можно использовать
    # путь на компьютере
    s = 'https://archive.ics.uci.edu/ml/' \
        'machine-learning-databases/iris/iris.data'
    print('From URL:', s)
    # header=None позволяет первому наблюдению
    # не уезжать в названия колонок, можно указать
    # номер строки, который станет в качестве
    # названий
    df = pd.read_csv(s, header=None, encoding='utf-8')
```

```

df.tail()

import numpy as np
import matplotlib.pyplot as plt

# .values вернёт np.array
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', 0, 1)
X = df.iloc[0:100, [0, 2]].values

plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='Setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='s', label='Versicolor')
plt.xlabel('Sepal length [cm]')
plt.ylabel('Petal length [cm]')
plt.legend(loc='upper left')
plt.show()

ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, y)
plt.plot(range(1, len(ppn.errors_) + 1),
         ppn.errors_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')
plt.show()

from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

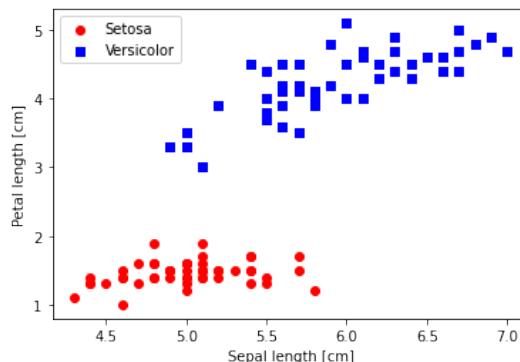
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    # делаем координатную сетку
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    # ravel делает один массив из нескольких
    # [[1, 2], [3, 4]] => [1, 2, 3, 4]
    # T - транспонирование
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

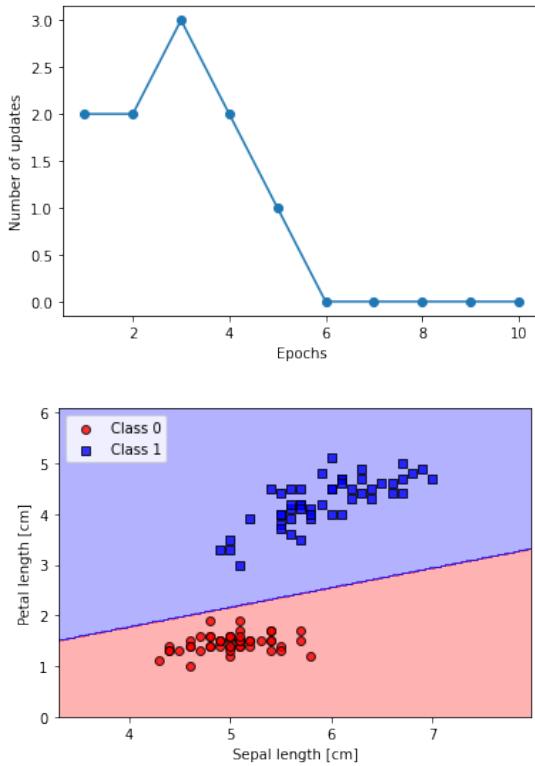
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8, # прозрачность
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}', # легенда
                    edgecolor='black') # цвет границы фигуры

plot_decision_regions(X, y, classifier=ppn)
plt.xlabel('Sepal length [cm]')
plt.ylabel('Petal length [cm]')
plt.legend(loc='upper left')
plt.show()

```

Теперь посмотрим на распределение наших данных, процесс тренировки и итоговую границу.





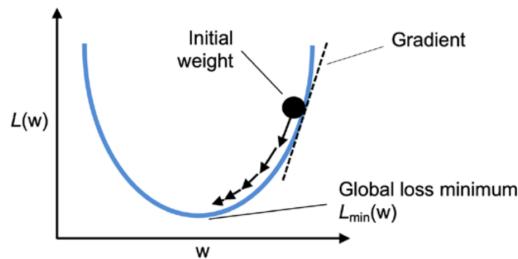
2.3 Adaptive linear neurons

В этой части мы посмотрим на другой тип однослойной **neural network** (NN): **ADaptive LInear NEuron (Adaline)**. Практически он интересен тем, что показывает как объявить и минимизировать непрерывную функцию потерь. Главное отличие в том, что здесь мы применяем линейную функцию активации, т.е. $\sigma(z) = z$ (вместо unit step function).

Будем минимизировать функцию потерь с помощью градиентного спуска. В adaline функция потерь это **mean squared error (MSE)**:

$$L(w, b) = \frac{1}{2n} \sum_{i=1}^n (y^{(i)} - \sigma(z^{(i)}))^2$$

$\frac{1}{2}$ добавлена для того, чтобы было удобнее дифференцировать, главная прелест линейной функции в том, что теперь можно взять производную от L по весам, также очень большой плюс в том, что loss function выпуклая, т.е. можно применить спуск.



Заметим, что в этом алгоритме мы обновляем наши значения на базе всего датасета, а не после каждого конкретного примера, поэтому ещё данный алгоритм называют **full batch** gradient descent. Немного математики:

$$\Delta w = -\eta \nabla_w L(w, b), \Delta b = -\eta \nabla_b L(w, b)$$

$$\frac{\partial L}{\partial w_j} = -\frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) x_j^{(i)}$$

$$\frac{\partial L}{\partial b} = -\frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)}))$$

Теперь реализуем алгоритм:

```

class AdalineGD:
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.float_(0.)
        self.losses_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            # у нас она линейная
            output = self.activation(net_input)
            errors = (y - output)
            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (errors**2).mean()
            self.losses_.append(loss)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_) + self.b_

    def activation(self, X):
        return X

    # у нас же функция любого 0 либо 1
    # линейная функция даёт значение от 0 до 1
    # поэтому зафиксируем в 0.5
    def predict(self, X):
        return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)

import os
import pandas as pd
s = 'https://archive.ics.uci.edu/ml/' \
    'machine-learning-databases/iris/iris.data'
print('From URL:', s)
df = pd.read_csv(s, header=None, encoding='utf-8')
df.tail()

import numpy as np
import matplotlib.pyplot as plt

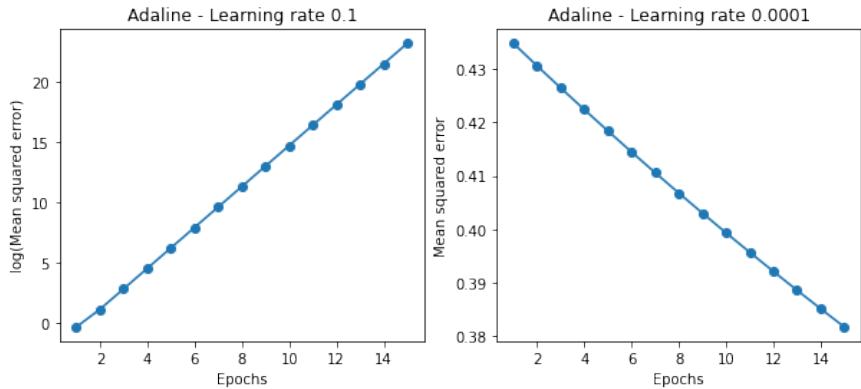
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', 0, 1)
X = df.iloc[0:100, [0, 2]].values

plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='Setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='s', label='Versicolor')
plt.xlabel('Sepal length [cm]')
plt.ylabel('Petal length [cm]')
plt.legend(loc='upper left')
plt.show()

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
ada1 = AdalineGD(n_iter=15, eta=0.1).fit(X, y)
ax[0].plot(range(1, len(ada1.losses_)+1),
            np.log10(ada1.losses_), marker='o')
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('log(Mean squared error)')
ax[0].set_title('Adaline - Learning rate 0.1')
ada2 = AdalineGD(n_iter=15, eta=0.0001).fit(X, y)
ax[1].plot(range(1, len(ada2.losses_)+1),
            ada2.losses_, marker='o')
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('Mean squared error')
ax[1].set_title('Adaline - Learning rate 0.0001')
plt.show()

```

Интересно посмотреть как мы обучались в зависимости от разных η , поэтому прикреплю картинку. Как мы видим, если выбрать маленькое значение, то мы очень долго сходимся, а если очень большое то даже расходимся, поэтому это значение довольно важно.



Теперь рассмотрим улучшение через **feature scaling**. Их очень много, но сейчас мы пройдём такой метод как **standardization**. Это позволяет алгоритму быстрее сходиться, но не делает датасет нормально распределённым. Мы делаем так, чтобы у всех фич среднее значение было равно 0, а $\text{std} = 1$. Например, можно сделать так:

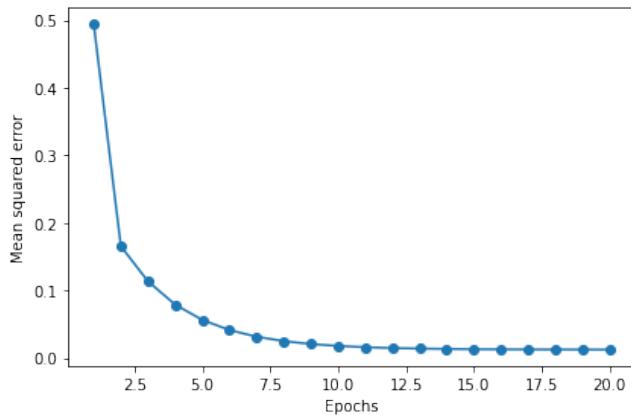
$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

Здесь j – это j -ая фича, а x – столбец из всех n примеров, μ – sample mean, σ – std. Теперь проще находить правильный η и bias. Если значения разбросаны, то найти гиперпараметры сразу для всех фич очень сложно. Реализуем то, что обсудили и посмотрим как это нам поможет на графике.

```
# feature scaling
X_std = np.copy(X)
X_std[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()
X_std[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()
ada_gd = AdalineGD(n_iter=20, eta=0.5)
ada_gd.fit(X_std, y)

# нарисуем
plot_decision_regions(X_std, y, classifier=ada_gd)
plt.title('Adaline - Gradient descent')
plt.xlabel('Sepal length [standardized]')
plt.ylabel('Petal length [standardized]')
plt.legend(loc='upper left')
# автоматически делает так, чтобы subplots
# выглядели хорошо на одном plot
plt.tight_layout()
plt.show()

plt.plot(range(1, len(ada_gd.losses_) + 1),
         ada_gd.losses_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Mean squared error')
plt.tight_layout()
plt.show()
```



Теперь поговорим про **large-scale machine learning** и **stochastic gradient descent**. Представим, что у нас огромный датасет, тогда для одного шага градиента нужно посмотреть на весь датасет, что дорого. Популярная альтернатива – это **stochastic gradient descent (SGD)**. Его иногда называют итеративным или online gradient descent. Делают вместо суммы так:

$$\Delta w_j = \eta (y^{(i)} - \sigma(z^{(i)})) x_j^{(i)}, \Delta b = \eta (y^{(i)} - \sigma(z^{(i)}))$$

То есть мы смотрим по очереди на каждый пример, а не сразу на среднее для всего сета. Это как бы приближение обычного спуска. Поверхность ошибки более грубая, поэтому гладкие локальные минимумы SGD не трогает. Нужно перемешать исходный датасет, допустим, если данные идут по порядку, а потом мешать после каждой эпохи, чтобы не было циклов. Часто в SGD lr заменяют на adaptive lr, который всё время уменьшается:

$$\frac{c_1}{[\text{number of iterations}] + c_2}$$

Также плюс в том, что данные можно подавать онлайн, то есть, например, если данных очень много, мало памяти или у нас web application. Но что, если нам нужно что-то среднее, тогда: **mini-batch gradient descent**. Идея в том, чтобы применять обычный GD, но к маленьким порциям, например, по 32 sample. То есть берём 32 примера, считаем сумму как в GD, применяем к w , берём другую группу и т.д. Во-первых сходится быстрее, чем GD, во-вторых можно заменить циклы на более быстрые векторные операции. Приступим к реализации sgd.

```
# shuffle=True/False — перемешивает данные каждую эпоху
class AdalineSGD:
    def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        self.random_state = random_state

    def fit(self, X, y):
        self._initialize_weights(X.shape[1])
        self.losses_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            losses = []
            for xi, target in zip(X, y):
                losses.append(self._update_weights(xi, target))
            avg_loss = np.mean(losses)
            self.losses_.append(avg_loss)
        return self

    def partial_fit(self, X, y):
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        # перемешанная последовательность индексов
        r = self.rgen.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        self.rgen = np.random.RandomState(self.random_state)
        self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=m)
        self.b_ = np.float_(0.)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        output = self.activation(self.net_input(xi))
        error = (target - output)
        self.w_ += self.eta * 2.0 * xi * error
        self.b_ += self.eta * 2.0 * error
        loss = error**2
        return loss

    def net_input(self, X):
        return np.dot(X, self.w_) + self.b_

    def activation(self, X):
        return X

    def predict(self, X):
        return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```

```

ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
ada_sgd.fit(X_std, y)
# пример partial_fit
ada_sgd.partial_fit(X_std[0, :], y[0])

```

3 Классификация с помощью Scikit-Learn

Для начала попробуем реализовать perceptron с помощью scikit-learn, чтобы почувствовать всё удобство. Кодировать метки нужно цифрами, хотя большинство библиотек отлично работают и со строками. **Overfitting** – проблема, при которой алгоритм очень хорошо обучился на тренировочных данных, а на тестовые данные обобщился очень плохо.

```

# небольшое изменение: теперь трен. данные отделяются от
# тест. с помощью кружочков
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):
    # ... этот код можно увидеть в главе 2

    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='none', edgecolor='black', alpha=1.0, # 0. — самое np.
                    linewidth=1, marker='o',
                    s=100, label='Test set')
        # s — marker size

    from sklearn import datasets
    import numpy as np
    iris = datasets.load_iris()
    X = iris.data[:, [2, 3]]
    y = iris.target
    print('Class labels:', np.unique(y))

    from sklearn.model_selection import train_test_split
    # train-test-split уже перемешивает данные внутри себя
    # stratify — этот параметр делает в тренировочных и тестовых данных
    # такое же соотношение меток, как и в исходном датасете
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=1, stratify=y
    )

    # проверим выше сказанное
    print('Labels counts in y:', np.bincount(y))
    print('Labels counts in y_train:', np.bincount(y_train))
    print('Labels counts in y_test:', np.bincount(y_test))

    from sklearn.preprocessing import StandardScaler
    sc = StandardScaler()
    # обязательный шаг, т.к. мы хотим чтобы тест.
    # и трен. данные были соотносимы
    sc.fit(X_train)
    X_train_std = sc.transform(X_train)
    X_test_std = sc.transform(X_test)

    from sklearn.linear_model import Perceptron
    ppn = Perceptron(eta0=0.1, random_state=1)
    ppn.fit(X_train_std, y_train)

    y_pred = ppn.predict(X_test_std)
    print('Misclassified examples: %d' % (y_test != y_pred).sum())

    from sklearn.metrics import accuracy_score
    print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))

    # можно скомбинировать predict & score
    print('Accuracy: %.3f' % ppn.score(X_test_std, y_test))

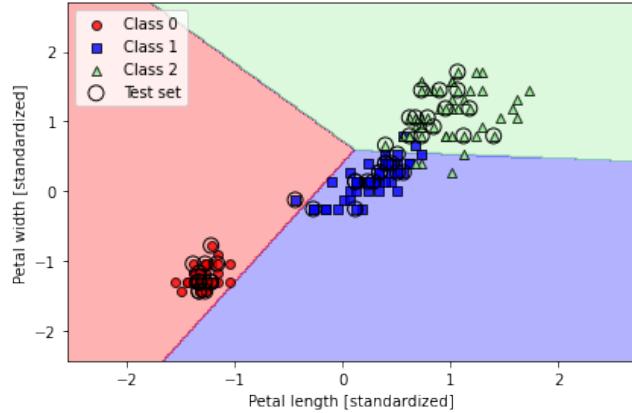
    # соединяет вертикально два массива
    X_combined_std = np.vstack((X_train_std, X_test_std))
    # также самое только горизонтально
    y_combined = np.hstack((y_train, y_test))
    plot_decision_regions(X=X_combined_std,
                          y=y_combined,

```

```

        classifier=ppn,
        test_idx=range(105, 150))
plt.xlabel('Petal length [standardized]')
plt.ylabel('Petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()

```



Как мы видим по картинке линейная граница не может чётко разделить уже хотя бы 3 класса. Вспомним, что, если классы чётко разделить нельзя, то ppn никогда не сойдётся, поэтому в таких случаях его не рекомендуют использовать.

3.1 Logistic regression

В предыдущем примере веса вечно обновляются, т.к. есть хотя бы один ложный пример, поэтому посмотрим на **logistic regression**. Multinomial logistic regression или softmax regression – это расширение бинарной лог. рег. на несколько классов. Пусть p – вероятность некоторого положительного события, $y = 1$, тогда введём такое понятие как **odds**:

$$\frac{p}{1-p}$$

Для лучшего понимания напишем так $p := p(y = 1|x)$. Теперь введём **logit function**, она получает на вход отрезок $[0, 1]$, а на выходе даёт всю вещ. прямую:

$$\text{logit}(p) = \log \frac{p}{1-p}$$

Под логистической моделью мы понимаем наличие линейной связи между взвешенной суммой x и нашей функцией logit:

$$\text{logit}(p) = w_1x_1 + \dots + w_mx_m + b = w^T x + b$$

Как раз нас интересует вероятность p принадлежности к классу. Давайте возьмём обратную функцию, которая даст нам $p \in [0, 1]$. Эта обратная функция называется **sigmoid function** или **logistic sigmoid function** и записывается так:

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

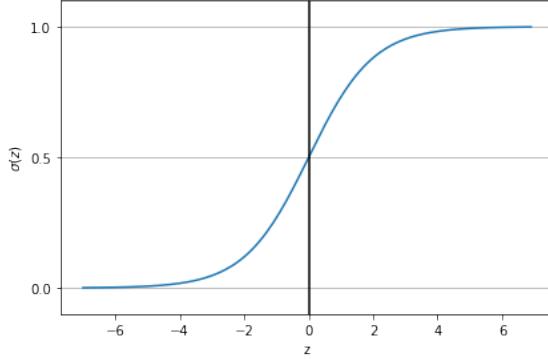
Теперь напишем немного кода и глянем на эту функцию:

```

import matplotlib.pyplot as plt
import numpy as np
def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))
z = np.arange(-7, 7, 0.1)
sigma_z = sigmoid(z)
plt.plot(z, sigma_z)
# вертикальная линия x=0.0
plt.axvline(0.0, color='k')
plt.ylim(-0.1, 1.1)
plt.xlabel('z')
plt.ylabel('$\sigma(z)$')
# засечки и полупрозрачные линии
plt.yticks([0.0, 0.5, 1.0])

```

```
# даёт нам оси x, y
ax = plt.gca()
# разрешаем сетку
ax.yaxis.grid(True)
plt.tight_layout()
plt.show()
```



Получается, что в итоге разница между Adaline и log. reg. – это функция активации, которая отождествляется с вероятностью к классу $y = 1$.

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases} \Leftrightarrow \hat{y} = \begin{cases} 1 & \text{if } z \geq 0.0 \\ 0 & \text{otherwise} \end{cases}$$

Теперь научимся обучать веса регрессии. Для начала введём похожую на L функцию \mathcal{L} , которую мы хотим максимизировать.

$$\mathcal{L}(w, b | x) = p(y|x; w, b) = \prod_{i=1}^n p(y^{(i)}|x^{(i)}; w, b) = \prod_{i=1}^n (\sigma(z^{(i)}))^{y^{(i)}} (1 - \sigma(z^{(i)}))^{1-y^{(i)}}$$

Первое равенство очевидно, второе – просто для каждого примера, третье – заметим, что наша σ даёт вероятность предсказания $y = 1$, поэтому нам нужны два множителя. На практике проще максимизировать $\log \mathcal{L}$:

$$l(w, b | x) = \log \mathcal{L}(w, b | x) = \sum_{i=1}^n \left[y^{(i)} \log(\sigma(z^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(z^{(i)})) \right]$$

Для чего нужен \log ? Во-первых, если вероятности малы, то не будет недооценки, во-вторых сложение проще дифферинцировать, чем умножение. Немного формальнее про третий переход первого равенства. Вероятность числа $1 - p$, поэтому – это **Bernoulli variable**, т.е. $y \sim Bern(p)$.

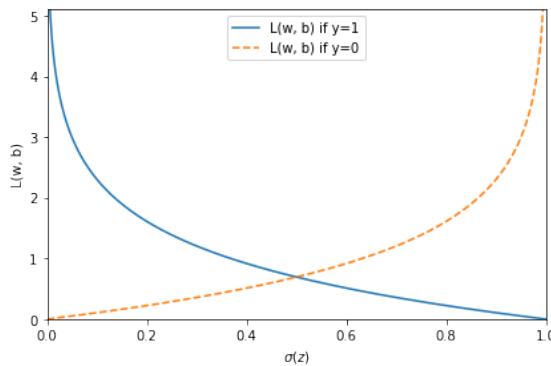
$$P(Y = 1 | X = x^{(i)}) = \sigma(z^{(i)}), P(Y = 0 | X = x^{(i)}) = 1 - \sigma(z^{(i)})$$

Осталось возвести в степень и умножить. Нам нужно сделать **gradient ascent**, но, если поставить ' $-$ ', то можно сделать обычный спуск. Перепишем в виде системы, чтобы изобразить:

$$L(\sigma(z), y; w, b) = \begin{cases} -\log(\sigma(z)) & \text{if } y = 1 \\ -\log(1 - \sigma(z)) & \text{if } y = 0 \end{cases}$$

Посмотрим на код и вывод:

```
def loss_1(z):
    return -np.log(sigmoid(z))
def loss_0(z):
    return -np.log(1 - sigmoid(z))
z = np.arange(-10, 10, 0.1)
sigma_z = sigmoid(z)
c1 = [loss_1(x) for x in z]
plt.plot(sigma_z, c1, label='L(w, b) if y=1')
c0 = [loss_0(x) for x in z]
plt.plot(sigma_z, c0, linestyle='--', label='L(w, b) if y=0')
plt.ylim(0.0, 5.1)
plt.xlim([-0, 1])
plt.xlabel('$\sigma(z)$')
plt.ylabel('L(w, b)')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```



Теперь полностью реализуем сам алгоритм:

```
# надо понимать, что работает только
# для бинарной классификации
class LogisticRegressionGD:
    # ...

    def fit(self, X, y):
        # ...

        for i in range(self.n_iter):
            # ...

            loss = (-y.dot(np.log(output))) - ((1 - y).dot(np.log(1 - output)))
            self.losses_.append(loss)
        return self

    # ...

    def activation(self, z):
        # np.clip: все значения < -250 делает равными -250,
        # все значения > 250 делает равными 250
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

    # ...

    # ...
X_train_01_subset = X_train_std[(y_train == 0) | (y_train == 1)]
y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
lrgd = LogisticRegressionGD(eta=0.3, n_iter=1000, random_state=1)
lrgd.fit(X_train_01_subset, y_train_01_subset)
plot_decision_regions(X=X_train_01_subset, y=y_train_01_subset,
                      classifier=lrgd)
# ...
```

Видно, что в этом коде, по сравнению с Adaline, поменялись только функция активации и формула для потерь. Очень интересный факт – это то, что градиенты для Adaline и lg совпали. Я не буду приводить выкладки по вычислению градиентов, но скажу, что их можно взять с помощью правила цепочки. Пару слов в сторону: mutually exclusive – каждое наблюдение принадлежит ровно к одному классу, в противовес – multilabel classification – несколько меток для того же наблюдения. Теперь реализуем тоже самое только из коробки.

```
from sklearn.linear_model import LogisticRegression
# присутствует загадочный параметр C:
# inverse of regularization strength
# это будет рассмотрено далее
lr = LogisticRegression(C=100.0, solver='lbfgs', multi_class='ovr')
lr.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined, classifier=lr)
plt.xlabel('Petal length [standardized]')
plt.ylabel('Petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

Заметим в коде такой solver как **lbfgs**. Для оптимизации выпуклых функций есть множество алгоритмов и для лог. рег. не рекомендуется использовать SGD. Из коробки доступны след. алгоритмы: 'newton-cg', 'lbfgs', 'liblinear', 'sag', и 'saga'.

```
lr.predict_proba(X_test_std[:, :])
# вывод такой:
# array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
```

```

#           [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
#           [8.48831425e-01, 1.51168575e-01, 2.62277619e-14]])

lr.predict_proba(X_test_std[:3, :]).sum(axis=1)
# по столбцам
# ожидается, что везде 1.

lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
# вывод:
# array([2, 0, 0])

lr.predict(X_test_std[:3, :])
# делает тоже самое, но более удобно записано

lr.predict(X_test_std[0, :].reshape(1, -1))
# если мы хотим предсказать только один пример,
# то нужно сделать двумерный массив из одномерного
# это удобно делать через reshape

```

Это пример работы с предсказаниями в этой библиотеке. Далее мы рассмотрим такие вещи как **overfitting** и **regularization**.

Overfitting – стандартная проблема для машинного обучения, когда алгоритм плохо обощается на тестовые данные, при этом хорошо работает на тренировочных. В этом случае говорят, что у алгоритма **high variance**. Это может произойти, если модель слишком сложна для простых данных. С другой стороны есть **underfitting (high bias)** – плохо работает даже на тренировочных данных. В случае variance можно сказать, что модель чувствительна к перестановке данных, в bias – наоборот нет. Один из способов настраивать сложность модели – это через **regularization**. Этот метод помогает фильтровать шум данных, не давать переобучаться и настраивает корреляцию между фичами. Самый популярный подвид – это **L2 regularization** (L2 shrinkage или weight decay):

$$\frac{\lambda}{2n} \|w\|^2 = \frac{\lambda}{2n} \sum_{j=1}^m w_j^2$$

Здесь λ – **regularization parameter**. λ нужна, чтобы она ушла при GD. λ добавлена, чтобы порядок величины был такой же как и у cost function. Для регуляризации важно, чтобы данные были стандартизованы, потому что нужен один порядок величин. Это слагаемое нужно добавить в L функцию. Но тогда нужно пересчитать и формулу для градиента, т.к. у нас в L сумма, то в градиенте надо просто прибавить

$$\frac{\lambda}{n} w_j$$

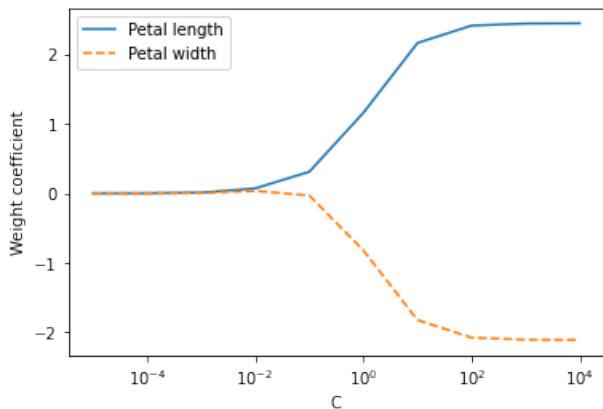
Теперь через λ можно контролировать обучение: увеличивая параметр мы увеличиваем regularization strength и наоборот. Также важно заметить, что bias unit не участвует в сумме. Вспомним параметр C – этот параметр следует из соглашения в svm (будет рассмотрено далее), он обратно пропорционален к λ . Следовательно, уменьшая inverse-regularization параметр C , мы увеличиваем силу регуляризации.

```

weights, params = [], []
for c in np.arange(-5, 5):
    lr = LogisticRegression(C=10.**c, multi_class='ovr')
    lr.fit(X_train_std, y_train)
    # lr.coef_ – массив весов (1, n_features)
    weights.append(lr.coef_[1])
    params.append(10.**c)
weights = np.array(weights)
plt.plot(params, weights[:, 0], label='Petal length')
plt.plot(params, weights[:, 1], linestyle='--', label='Petal width')
plt.ylabel('Weight coefficient')
plt.xlabel('C')
plt.legend(loc='upper left')
# иначе шкала со слишком большими числами
plt.xscale('log')
plt.show()

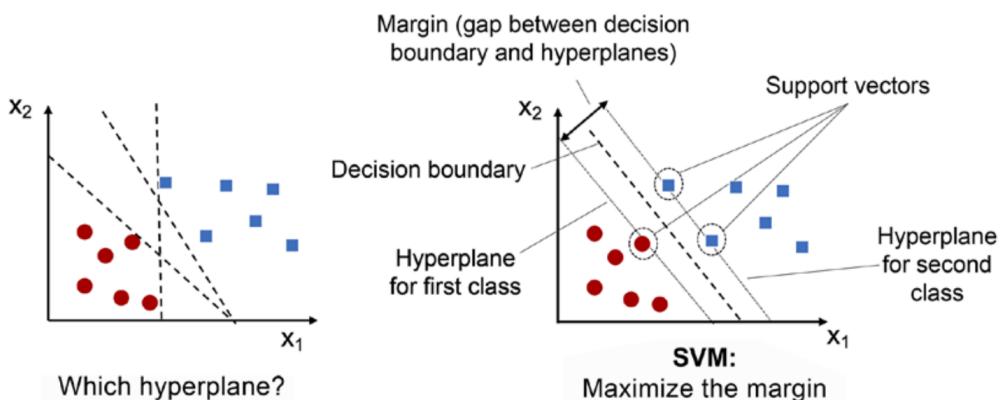
```

На картинке ниже видно, что, чем меньше параметр, тем меньше значения весов. Возникает вопрос, а почему не все алгоритмы сразу регуляризовать – потому что может произойти недообучение.



3.2 Maximum margin classification with SVM

Support vector machine можно рассматривать как расширение персептрона. Но здесь у нас немножко другая цель – не просто разделить, а так разделить, чтобы ещё расстояние было максимальное. См. картинку.



Модели с маленькими отступами более склонны к переобучению, а с большими – имеют меньшую ошибку обобщения, поэтому нам интересны svm. Рассмотрим работу с линейно не разделяемыми случаями с помощью **slack variables**, которые потом привели к soft-margin classification. Их было необходимо ввести, чтобы алгоритм сходился в любом случае. Большое значение C даёт сильное наказание, из-за чего будет учтён каждый выброс, маленькое же значение C даст более обобщённый результат. Логистическая регрессия более склонна к выпадам, но с ней проще работать и её проще реализовать.

```
from sklearn.svm import SVC
svm = SVC(kernel='linear', C=1.0, random_state=1)
svm.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined,
                      classifier=svm)
plt.xlabel('Petal length [standardized]')
plt.ylabel('Petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

Не очень понятно, что такое **kernel**, рассмотрим это позже. Рассмотрим ещё 4 строчки кода, с помощью которых можно сделать онлайн тренировку:

```
from sklearn.linear_model import SGDClassifier
ppn = SGDClassifier(loss='perceptron')
lr = SGDClassifier(loss='log')
svm = SGDClassifier(loss='hinge')
```

3.3 Нелинейные задачи с использованием kernel SVM

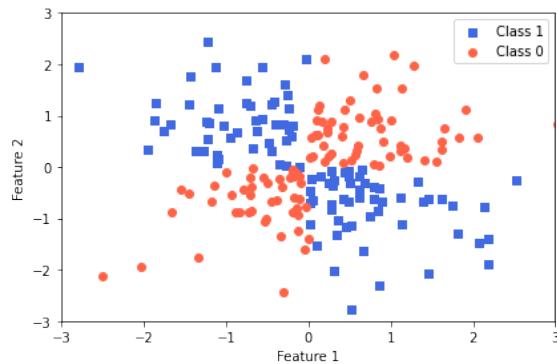
Для начала сделаем искусственный XOR dataset.

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

```

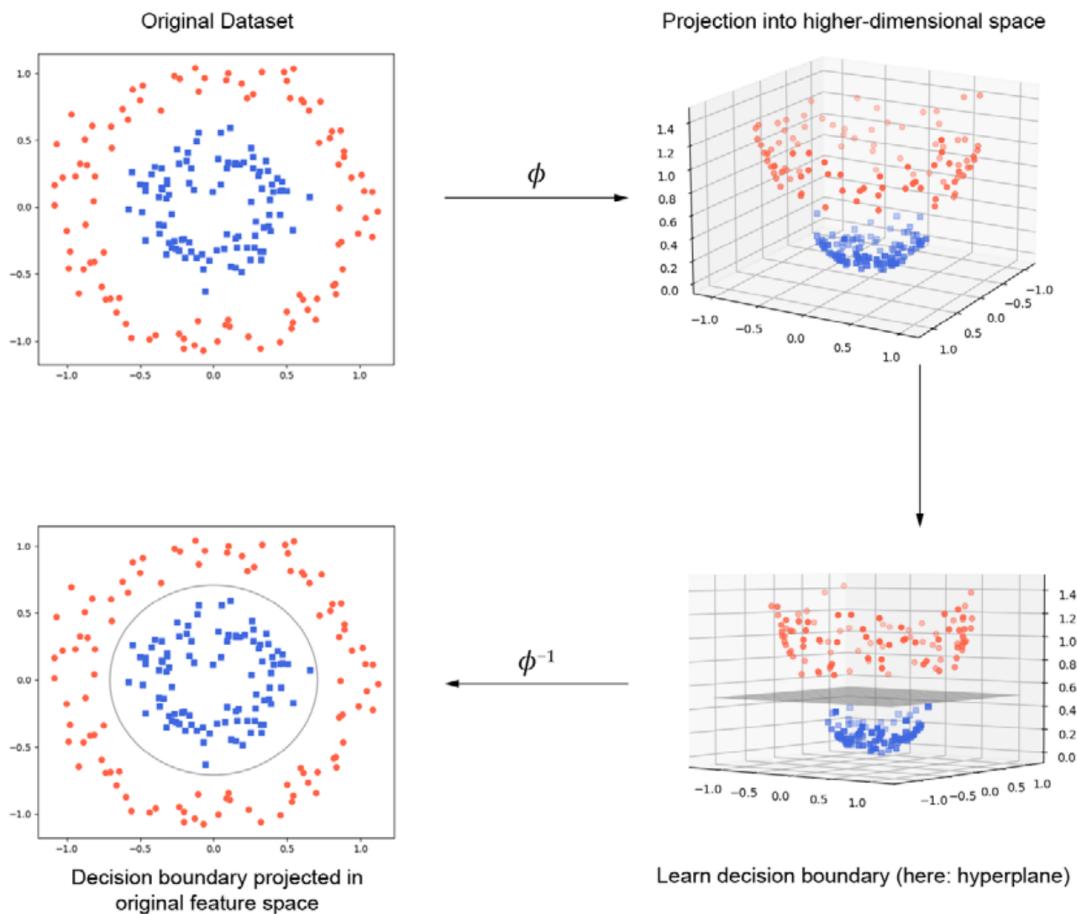
np.random.seed(1)
# случайный массив 200 на 2
X_xor = np.random.randn(200, 2)
y_xor = np.logical_xor(X_xor[:, 0] > 0, X_xor[:, 1] > 0)
y_xor = np.where(y_xor, 1, 0)
plt.scatter(X_xor[y_xor == 1, 0], X_xor[y_xor == 1, 1],
            c='royalblue', marker='s', label='Class 1')
plt.scatter(X_xor[y_xor == 0, 0], X_xor[y_xor == 0, 1],
            c='tomato', marker='o', label='Class 0')
plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(loc='best')
plt.tight_layout()
plt.show()

```



Безусловно, используя линейную гиперплоскость, разделить данные отлично невозможно. Основная идея **kernel methods** работы с такими случаями – это создать нелинейную комбинацию входных данных, чтобы спроцировать их на большую размерность, используя **mapping function** ϕ , где данные уже линейно разделимы. Посмотрим на следующую картинку, на ней

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$



Теперь будем использовать **kernel trick**, чтобы найти гиперплоскость в многомерном пространстве, потому что иначе поиск становится очень затратным по времени и памяти. На практике нам нужно заменить $x^{(i)T}x^{(j)}$ на $\phi(x^{(i)})^T\phi(x^{(j)})$. Чтобы избежать этого дорогое произведения, мы вводим **kernel function**:

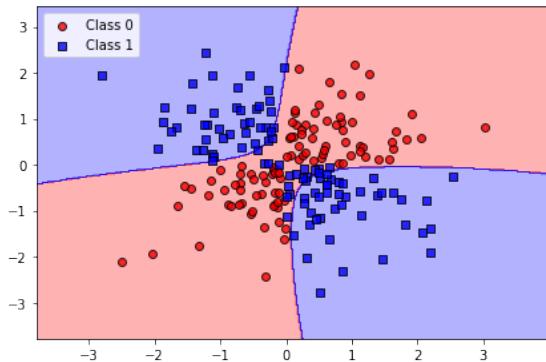
$$k(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T\phi(x^{(j)})$$

Одно из самых широко используемых ядер – это **radial basis function (RBF)** kernel (Gaussian kernel):

$$k(x^{(i)}, x^{(j)}) = \exp\left(-\frac{\|x^{(i)} - x^{(j)}\|^2}{2\sigma^2}\right) = \exp(-\gamma\|x^{(i)} - x^{(j)}\|^2)$$

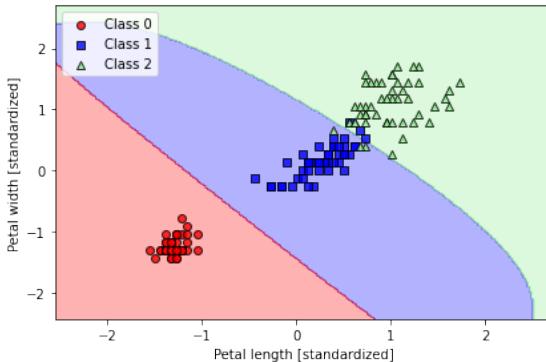
Теперь вспомним про линейное ядро – это просто скалярное произведение, что давало линейный результат. Наш параметр $\gamma = \frac{1}{2\sigma^2}$ свободен для оптимизации. Грубо говоря наше ядро – similarity function между парой примеров. В итоге 1 – очень похожие примеры, 0 – очень различающиеся. Теперь попробуем написать код и разделить наш искусственный датасет XOR.

```
svm = SVC(kernel='rbf', random_state=1, gamma=0.10, C=10.0)
svm.fit(X_xor, y_xor)
plot_decision_regions(X_xor, y_xor, classifier=svm)
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

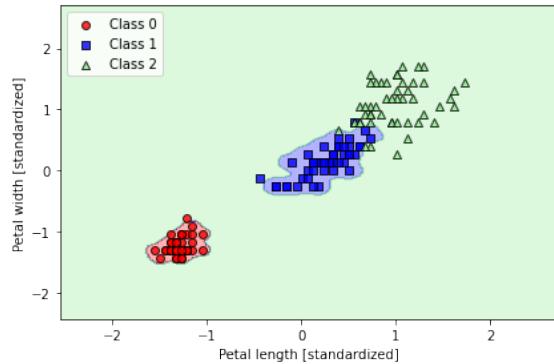


Параметр γ можно воспринимать как параметр отсечения для Гауссовой сферы. Если мы его увеличиваем, то мы лучше достигаем тренировочных примеров, тем самым получаем более обтягивающую границу. Чтобы это лучше увидеть, давайте применим ядро rbf для датасета Iris.

```
svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
svm.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined, classifier=svm)
plt.xlabel('Petal length [standardized]')
plt.ylabel('Petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```



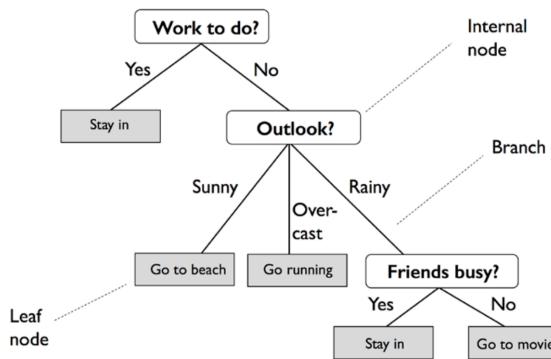
Как мы видим граница получилась довольно гладкая. Теперь давайте поставим $\gamma = 100.0$. Мы увидим, что граница прям обтягивает наши примеры.



Правда такой классификатор имеет большую ошибку обобщения.

3.4 Decision tree learning

Коротко дерево решений можно описать одной картинкой:



Эта картинка демонстрирует концепцию, основываясь на **categorical variables**. Когда мы выбираем следующую вышину, мы выбираем ту, у которой больше **information gain (IG)** (будет пояснено дальше). На практике может получиться очень глубокое дерево, то есть мы получим переобучение, поэтому нужно ограничить глубину. Нужно ввести целевую функцию, наша цель – каждый раз получить максимальный IG:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^m \frac{N_j}{N_p} I(D_j)$$

f – фича, по которой идёт разбиение, D_p, D_j – датасеты родителя и j -го ребёнка у вершины, I – это **impurity** measure, N – число элементов. Чтобы уменьшить комбинаторный поиск, обычно реализуют бинарное дерево, есть только два ребёнка: *left* и *right*.

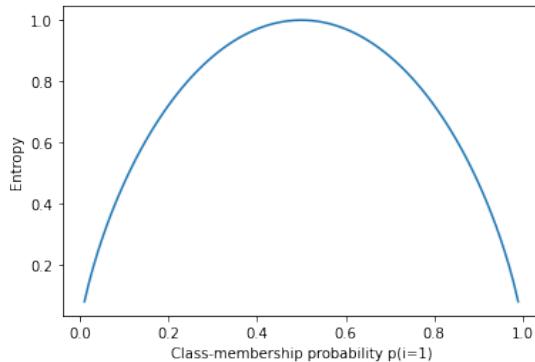
З самые популярные **impurity measures** или **splitting criteria**: Gini impurity (I_G), entropy (I_H), classification error (I_E). Начнём с определения **энтропии** для всех непустых классов ($p(i|t) \neq 0$):

$$I_H(t) = - \sum_{i=1}^c p(i|t) \log_2 p(i|t)$$

$p(i|t)$ – это пропорция количества примеров, которые принадлежат классу i в вершине t . Энтропия равна 0, если есть только один класс, энтропия максимальна, если у нас uniform class distribution. Мы пытаемся максимизировать взаимную информацию в дереве. Напишем некоторое количество строк кода и посмотрим на график.

```

def entropy(p):
    return -p * np.log2(p) - (1 - p) * np.log2(1 - p)
x = np.arange(0.0, 1.0, 0.01)
ent = [entropy(p) if p != 0 else None for p in x]
plt.ylabel('Entropy')
plt.xlabel('Class-membership probability p(i=1)')
plt.plot(x, ent)
plt.show()
  
```



Теперь про **Gini impurity** – критерий минимизации вероятности ошибки:

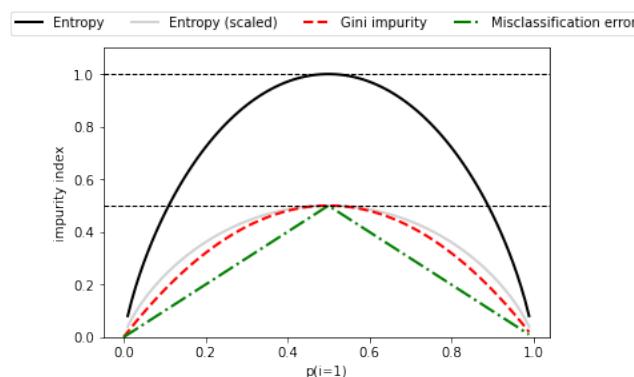
$$I_G(t) = \sum_{i=1}^c p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^c p(i|t)^2$$

Опять же максимум, если всё идеально перемешано. На самом деле эти два критерия на практике дают очень похожий результат. Теперь напишем что такое **classification error**:

$$I_E(t) = 1 - \max\{p(i|t)\}$$

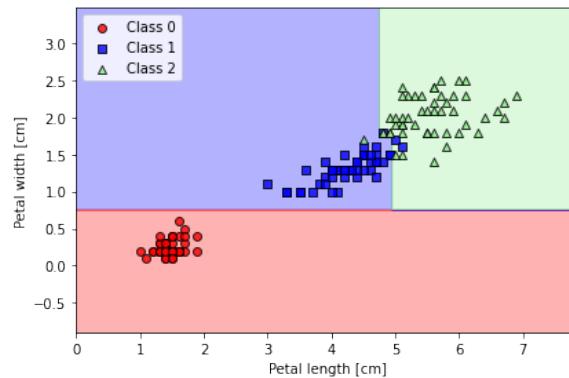
Этот критерий не рекомендуется для разрастающихся деревьев, т.к. он менее чувствителен к изменениям вероятностей в вершинах. Можно сделать даже пример из 3 вершин, где первые два критерия строго лучшего этого, этот даже может не различить примеры вовсе. Для лучшего визуального понимания давайте нарисуем все 3 критерия. Мы также добавим график entropy / 2, чтобы показать, что Gini находится по середине между двумя другими.

```
import matplotlib.pyplot as plt
import numpy as np
def gini(p):
    return p * (1 - p) + (1 - p) * (1 - (1 - p))
def entropy(p):
    return -p * np.log2(p) - (1 - p) * np.log2(1 - p)
def error(p):
    return 1 - np.max([p, 1 - p])
x = np.arange(0.0, 1.0, 0.01)
ent = [entropy(p) if p != 0 else None for p in x]
sc_ent = [e * 0.5 if e else None for e in ent]
err = [error(i) for i in x]
fig = plt.figure()
# эти три числа понимаются по отдельности, кол-во строк, столбцов, текущий индекс
ax = plt.subplot(111)
for i, lab, ls, c in zip([ent, sc_ent, gini(x), err],
                         ['Entropy', 'Entropy (scaled)', 'Gini impurity',
                          'Misclassification error'],
                         [':', '--', '-.', '-'],
                         ['black', 'lightgray', 'red', 'green']):
    line = ax.plot(x, i, label=lab, linestyle=ls, lw=2, color=c)
ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
          ncol=5, fancybox=True, shadow=False)
ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
plt.ylim([0, 1.1])
plt.xlabel('p(i=1)')
plt.ylabel('impurity index')
plt.show()
```



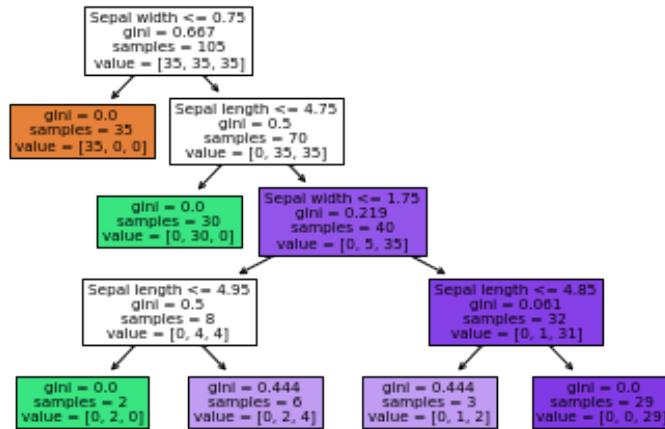
Теперь научимся строить и обучать такое дерево. Такое дерево может строить довольно сложные граници, разделяя всё на прямоугольники, но нужно быть осторожным из-за риска переобучения. Feature scaling можно применить для лучшей визуализации, но оно совершенно не обязательно для этого алгоритма.

```
from sklearn.tree import DecisionTreeClassifier
tree_model = DecisionTreeClassifier(criterion='gini', max_depth=4,
                                     random_state=1)
tree_model.fit(X_train, y_train)
X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(X_combined, y_combined, classifier=tree_model)
plt.xlabel('Petal length [cm]')
plt.ylabel('Petal width [cm]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```



Мы видим стандартную для дерева границу в виде параллельных прямоугольников. Полезная особенность нашей библиотеки – это то, что можно читабельно изобразить наше дерево.

```
from sklearn import tree
feature_names = ['Sepal length', 'Sepal width']
tree.plot_tree(tree_model, feature_names=feature_names,
               filled=True)
plt.show()
```



Также здесь окрашено так, что более тёмному цвету соответствует более приоритетная вершина. Влево – True, вправо – False. Теперь поговорим про **random forests**. Эти методы стали популярны ввиду их хороших показателей и устойчивости к переобучению. Random forests – это просто группа глубоких decision trees, которые по отдельности очень переобучены. Алгоритм можно описать в виде простых 4-х этапов:

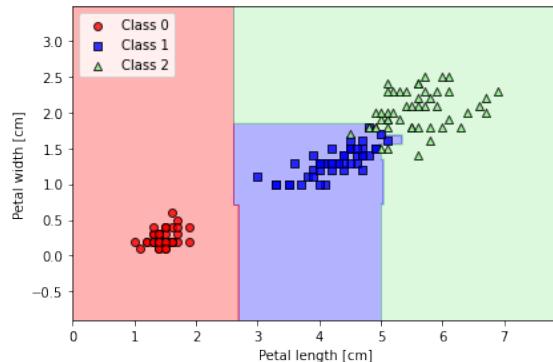
1. Возьмём случайный bootstrap sample размера n : случайно выберем n примеров из тренировочного датасета с **replacement**.
2. Вырастим дерево из этого bootstrap sample. В каждой вершине:

- (a) Случайно выберем d фич **без replacement**.
 - (b) Разобъём вершину по фиче, которая даёт наибольший IG (либо другая целевая функция).
3. Повторим пункты 1-2 k раз.
4. Соединим предсказания с помощью **majority vote** (будет подробно рассмотрено в главе 7).

Небольшая модификация пункта 2: при вычислении лучшей фичи будем смотреть только на случайное подмножество датасета. Разберёмся с тем, что такое **replacement**. Пусть у нас есть урна с числами. Есть два способа их доставать: **не** класть обратно выбранное число, или класть обратно. В первом случае второе число зависит от того, какое число вытянули первым. Во втором случае вероятность каждого числа всегда постоянна. В случае ‘with replacement’ все числа независимые и имеют нулевую **ковариацию** (мера зависимости 2-х случайных величин).

Главный плюс леса в том, что не нужно сильно думать насчёт гиперпараметров. Единственное, что нужно трогать – это параметр k , с его увеличением растёт accuracy, но сильно растёт время ожидания. Оптимизацию параметров n и d (случайные на каждом шаге) мы рассмотрим в главе 6. С помощью n мы регулируем bias-variance tradeoff, его уменьшение даёт большую случайность, что в свою очередь не даёт переобучаться, но при этом результат хуже. При этом d обычно такой: $d = \sqrt{m_{\text{features}}}$, n выбирают равным количеству тренировочных примеров. Теперь реализуем лес.

```
from sklearn.ensemble import RandomForestClassifier
# по умолчанию Gini impurity
# n_jobs позволяет использовать параллелизм
# в параметрах — кол-во ядер
forest = RandomForestClassifier(n_estimators=25,
                                random_state=1,
                                n_jobs=2)
forest.fit(X_train, y_train)
plot_decision_regions(X_combined, y_combined,
                      classifier=forest)
```



3.5 KNN

Последний алгоритм из типа supervised, который мы рассмотрим в этой главе – это **k-nearest neighbor (KNN)**, он интересен, потому что фундаментально отличается от тех, что мы рассмотрели раньше. Этот алгоритм – типичный пример **lazy learner**. Он ленивый, потому что он не обучает discriminative function, а запоминает тренировочный датасет.

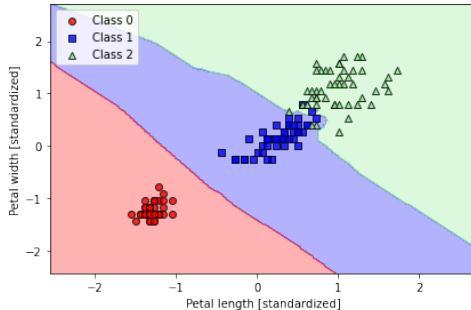
Модели с параметрами и без – на такие две группы можно поделить все алгоритмы. В первом случае мы обучаем параметры функции, чтобы потом что-то предсказать, при этом тренировочный датасет нам больше не нужен. Во-втором случае количество параметров не фиксированное, оно меняется в зависимости от количества данных. Два примера таких алгоритмов мы уже видели – kernel (но **не** linear) SVM и decision tree classifier/random forest. KNN принадлежит подгруппе обучения без параметров – instance-based learning – запоминание датасета. А ленивое обучение говорит о том, что нам нужно 0 времени на обучение.

Опишем алгоритм в 3 шага:

1. Нужно выбрать число k и метрику расстояния.
2. Найдём k ближайших соседей к той точке, которую мы предсказываем.
3. Присвоим ответ с помощью majority vote.

Посмотрим на плюсы и минусы memory-based подхода. Хорошо, что он быстро обрабатывает новые данные, с другой стороны время на запрос растёт линейно в зависимости от размера данных. С другой стороны, если количество фич очень маленькое, то можно реализовать быструю структуру данных: k-d tree или ball tree. Но если данных немного, то алгоритм прилично работает. Реализуем, используя Euclidean distance metric.

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5, p=2,
                           metric='minkowski')
knn.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined,
                      classifier=knn)
```



В случае равенства алгоритм выберет ближайшую точку. Так как мы используем метрику, то важно стандартизировать данные. Мы использовали minkowski distance – это просто обобщение евклидова и манх. расстояний:

$$d(x^{(i)}, x^{(j)}) = \left(\sum_k |x_k^{(i)} - x_k^{(j)}|^p \right)^{\frac{1}{p}}$$

Есть такой термин как проклятие размерностей – чем больше размерность пространства – тем как бы дальше находятся точки, даже если они соседние. В этом алгоритме и деревьях нельзя сделать регуляризацию, поэтому применяют feature selection и dimensionality reduction (следующие 2 главы). Можно использовать NVIDIA's CUDA library, RAPIDS ecosystem и её библиотеку cuML, чтобы обучать быстрее.

4 Data Preprocessing

4.1 Dealing with missing data

В реальном мире нормально, если часть данных отсутствует: измерения могут быть неприменимы, часть данных просто пустая, ошибка в сборе данных. Пропущенные данные: NULL, NaN, пустота. Большинству алгоритмов нужен полный датасет. Рассмотрим поиск пропусков в табличных данных. Создадим **comma-separated values (CSV)** file и посмотрим на него. scikit-learn умеет работать с pandas, но изначально всё было на numpy, поэтому рекомендуется брать из pandas dataframe numpy array.

```
import pandas as pd
from io import StringIO
csv_data = \
'''A,B,C,D
1.0,2.0,3.0,4.0
5.0,,8.0
10.0,11.0,12.0,''''
df = pd.read_csv(StringIO(csv_data))
print(df)
print(df.isnull().sum()) # по умолчанию axis=0
print(df.values)
```

| | A | B | C | D |
|---|------|------|------|-----|
| 0 | 1.0 | 2.0 | 3.0 | 4.0 |
| 1 | 5.0 | 6.0 | NaN | 8.0 |
| 2 | 10.0 | 11.0 | 12.0 | NaN |
| A | 0 | | | |
| B | 0 | | | |
| C | 1 | | | |
| D | 1 | | | |

dtype: int64
[[1. 2. 3. 4.]
 [5. 6. nan 8.]
 [10. 11. 12. nan]]

Самый простой способ справиться – **удалить** повреждённые строки или столбцы. У удаления есть большой минус – становится слишком мало данных, также мы можем потерять ценную информацию. Далее мы посмотрим на альтернативу: interpolation techniques.

```
# лишний = хотя бы один nan
print(df.dropna(axis=0)) # уберём лишние строки
print(df.dropna(axis=1)) # уберём лишние столбцы
# удалит только полностью пустые строки:
print(df.dropna(how='all'))
# удалит строки, где меньше 4 реальных значений
print(df.dropna(thresh=4))
# удалит только те строки, где nan на позиции C
print(df.dropna(subset=['C']))
```

```
A   B   C   D
0  1.0  2.0  3.0  4.0
      A   B
0  1.0  2.0
1  5.0  6.0
2 10.0 11.0
      A   B   C   D
0  1.0  2.0  3.0  4.0
1  5.0  6.0  NaN  8.0
2 10.0 11.0 12.0  NaN
      A   B   C   D
0  1.0  2.0  3.0  4.0
      A   B   C   D
0  1.0  2.0  3.0  4.0
2 10.0 11.0 12.0  NaN
```

Теперь попробуем **вставить** пропущенные данные в таблицу. Чтобы это решить, нужно применять различные interpolation techniques для оценки пропущенных значений. Самая популярная – mean imputation, ставим просто среднее значение всей колонки, ещё есть KNNImputer.

```
from sklearn.impute import SimpleImputer
import numpy as np
# median, most_frequent – виды стратегий
imr = SimpleImputer(missing_values=np.nan, strategy='mean')
imr = imr.fit(df.values)
imputed_data = imr.transform(df.values)
print(imputed_data)
# это всё полезно, если у нас категориальные данные,
# например, названия цветов
print(df.fillna(df.mean()))
```

```
[[ 1.   2.   3.   4. ]
 [ 5.   6.   7.5  8. ]
 [10.  11.  12.   6. ]]
      A   B   C   D
0  1.0  2.0  3.0  4.0
1  5.0  6.0  7.5  8.0
2 10.0 11.0 12.0  6.0
```

Теперь попытаемся понять scikit-learn estimator API. Мы использовали SimpleImputer, который является частью **transformer** API. При этом трансформер для тестовых данных тот же, мы не делаем заново fit на тестовых данных. Кстати говоря, алгоритмы из главы 3 относят к estimators.

4.2 Handling categorical data

Важно понимать разницу между ordinal и nominal фичами. Ordinal – фичи, которые можно отсортировать, например, размер футболки. Nominal – соответственно наоборот, например, цвет футболки. Сначала создадим датасет:

```
import pandas as pd
df = pd.DataFrame([
    ['green', 'M', 10.1, 'class2'],
    ['red', 'L', 13.5, 'class1'],
    ['blue', 'XL', 15.3, 'class2']])
df.columns = ['color', 'size', 'price', 'classlabel']
print(df)
```

Теперь соотнесём ordinal фичи с цифрами:

```
size_mapping = {'XL': 3, 'L': 2, 'M': 1}
df['size'] = df['size'].map(size_mapping)
print(df)
inv_size_mapping = {v: k for k, v in size_mapping.items()}
print(df['size'].map(inv_size_mapping))
```

Поговорим про то, как закодировать class labels. Большинство estimators внутри конвертируют всё в цифры, но всё же лучше сделать это самим.

```
import numpy as np
class_mapping = {label: idx for idx, label in
    enumerate(np.unique(df['classlabel']))}
print(class_mapping)
df['classlabel'] = df['classlabel'].map(class_mapping)
print(df)
inv_class_mapping = {v: k for k, v in class_mapping.items()}
df['classlabel'] = df['classlabel'].map(inv_class_mapping)
print(df)
```

Напишем, используя библиотеку:

```
from sklearn.preprocessing import LabelEncoder
class_le = LabelEncoder()
# Это shortcut κ fit, a common transform
y = class_le.fit_transform(df['classlabel'].values)
print(y)
print(class_le.inverse_transform(y))
```

Теперь сделаем one-hot encoding на nominal фичах:

```
X = df[['color', 'size', 'price']].values
color_le = LabelEncoder()
X[:, 0] = color_le.fit_transform(X[:, 0])
print(X)
```

Но если мы здесь остановимся, то сделаем самую распространённую ошибку. Мы закодировали цвета числами от 0 до 2, но для любого алгоритма $2 > 1 > 0$, а это значит, что результат будет не оптимальен, так как есть порядок. Для этого используют технику **one-hot encoding**. Идея в том, чтобы создать 3 новые фичи на 3 цвета, тогда любой цвет можно закодировать вектором из одной 1 и двух 0. Напишем это:

```
from sklearn.preprocessing import OneHotEncoder
X = df[['color', 'size', 'price']].values
color_ohe = OneHotEncoder()
print(color_ohe.fit_transform(X[:, 0].reshape(-1, 1)).toarray())
# Вывод:
# [[0.  1.  0.]
#  [0.  0.  1.]
#  [1.  0.  0.]]
```

Заметим, что мы сделали сложный reshape, чтобы модифицировать только одну колонку, но можно сделать проще:

```
from sklearn.compose import ColumnTransformer
X = df[['color', 'size', 'price']].values
# 'passthrough' показывает, что мы не хотим ничего делать
c_transf = ColumnTransformer([
    ('onehot', OneHotEncoder(), [0]),
    ('nothing', 'passthrough', [1, 2])])
print(c_transf.fit_transform(X).astype(float))
# Вывод:
# [[ 0.    1.    0.    1.   10.1]
#  [ 0.    0.    1.    2.   13.5]
#  [ 1.    0.    0.    3.   15.3]]
```

Вообще нужно быть осторожным с таким кодированием, так как мы создаём мультиколлинеарность, что может быть опасно для методов, где нужно, например, брать обратную матрицу. Чтобы как-то спасти ситуацию, можно убрать колонку color_blue, так как комбинация 00 уже говорит нам, что это синий цвет.

Можно сделать ещё проще. Метод get_dummies делает кодирование только строковых данных, а остальные не трогает:

```
print(pd.get_dummies(df[['price', 'color', 'size']]))

# price size color_blue color_green color_red
# 0 10.1 1 0 1 0
# 1 13.5 2 0 0 1
# 2 15.3 3 1 0 0

# Categories (unique values) per feature:
# auto : Determine categories automatically from the training data.
color_ohe = OneHotEncoder(categories='auto', drop='first')
c_transf = ColumnTransformer([
    ('onehot', color_ohe, [0]),
    ('nothing', 'passthrough', [1, 2])])
print(c_transf.fit_transform(X).astype(float))
```

Есть альтернативы one-hot encoding. Эти способы могут быть полезны, если у нас очень много уникальных категориальных меток.

- Бинарное кодирование: пусть K – количество уникальных категорий, тогда нам нужно $\log_2(K)$ колонок. Сначала каждое число кодируется в бинарный вид, а потом каждый разряд в отдельную колонку.
- Подсчёт частоты встречаемости.

В итоге у нас получается появилсялся ещё один гиперпараметр. Пару слов про кодирование ordinal фич. Если мы не уверены в численной разнице между ordinal фичами или разница между двумя фичами не определена, то можно закодировать их обычными 0 и 1. Посмотрим на код:

```
df = pd.DataFrame([['green', 'M', 10.1, 'class2'],
                   ['red', 'L', 13.5, 'class1'],
                   ['blue', 'XL', 15.3, 'class2']])
df.columns = ['color', 'size', 'price', 'classlabel']
print(df)

df['x > M'] = df['size'].apply(
    lambda x: 1 if x in {'L', 'XL'} else 0)
df['x > L'] = df['size'].apply(
    lambda x: 1 if x == 'XL' else 0)
del df['size']
print(df)
# Вывод:
#      color  size   price  classlabel
# 0    green     M    10.1      class2
# 1     red      L    13.5      class1
# 2    blue     XL    15.3      class2
#      color   price  classlabel  x > M  x > L
# 0    green    10.1      class2      0      0
# 1     red     13.5      class1      1      0
# 2    blue     15.3      class2      1      1
```

4.3 Ещё о предобработке

Рассмотрим разделение датасета на тренировочный и тестовый. Будем работать теперь с Wine датасетом. Тестовый датасет важен так как показывает объективную оценку. Wine датасет имеет 178 примеров по 13 фич.

```
df_wine = pd.read_csv('https://archive.ics.uci.edu/',
                      'ml/machine-learning-databases/',
                      'wine/wine.data', header=None)
df_wine.columns = ['Class label', 'Alcohol',
                   'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium',
                   'Total phenols', 'Flavanoids',
                   'Nonflavanoid phenols',
                   'Proanthocyanins',
                   'Color intensity', 'Hue',
                   'OD280/OD315 of diluted wines',
                   'Proline']
print('Class labels', np.unique(df_wine['Class label']))
df_wine.head()

from sklearn.model_selection import train_test_split
X, y = df_wine.iloc[:, 1:], df_wine.iloc[:, 0].values
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.3, random_state=0,
                     stratify=y)
```

Иногда делают так: обучим модель, потом протестируем, и если результат хороший, то можно обучить модель на всех данных сразу. Теперь поговорим о feature scaling. Есть два основных способа: **normalization** и **standardization**. Под первым обычно понимают **min-max scaling**:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}$$

В scikit-learn это выглядит так:

```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
X_train_norm = mms.fit_transform(X_train)
X_test_norm = mms.transform(X_test)
```

Этот метод нужен, когда мы хотим вогнать значения в какой-то отрезок, но всё таки стандартизация применяется чаще. Причина в том, что начальные веса имеют значения близкие к нулю, а тогда мы хотим, чтобы и фичи были той же структуры. Однако мы не меняем форму распределения и не делаем ненормально распределённые данные нормально распределёнными. И по сравнению с min-max scaling стандартизация делает алгоритм менее чувствительным к выбросам. Формула:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Здесь μ_x – mean, σ_x – std. Теперь реализуем это в коде:

```
ex = np.array([0, 1, 2, 3, 4, 5])
print('standardized:', (ex - ex.mean()) / ex.std())
# standardized:
# [-1.46385011 -0.87831007 -0.29277002  0.29277002  0.87831007  1.46385011]
print('normalized:', (ex - ex.min()) / (ex.max() - ex.min()))
# normalized: [0.  0.2 0.4 0.6 0.8 1.]
```

```
from sklearn.preprocessing import StandardScaler
stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.transform(X_test)
```

Важно, что мы делаем fit только один раз. Ещё один scaler – **RobustScaler** – он очень полезен, если у нас мало данных, а количество выбросов большое; также он хорош, когда мы переобучаемся. Он удаляет медианное значение и масштабирует значения между 1 и 3 квартилями данных, поэтому выбросы становятся менее выраженным.

4.4 Selecting meaningful features

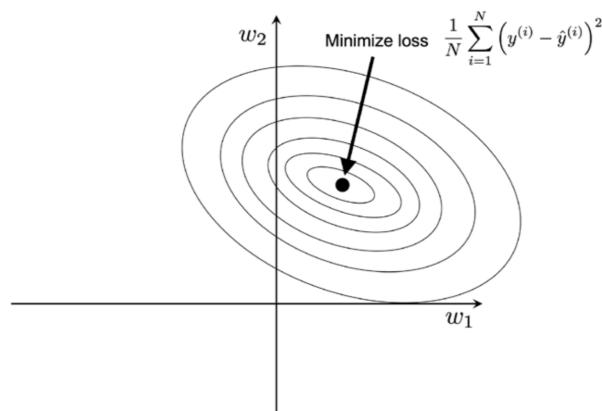
Если мы видим, что на тестовых данных результат хуже, чем на тренировочных, то это явный сигнал переобучения. Это значит, что мы очень близко учимся к конкретным данным, но обобщаем плохо. Причина – модель слишком сложная для наших данных, мы имеем high variance. Способы уменьшить generalization error: собрать больше данных, использовать регуляризацию, выбрать модель попроще, сократить размерность данных.

Сначала рассмотрим **L1** и **L2 regularization**:

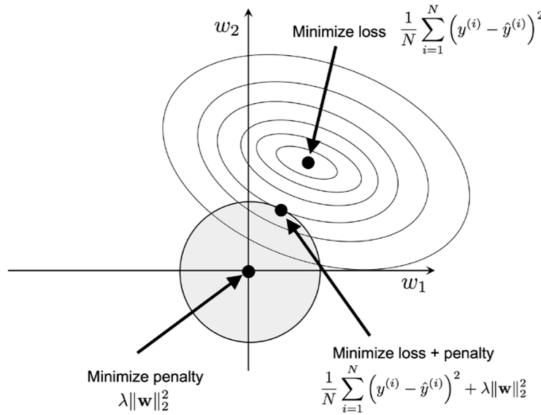
$$L2 : \|w\|_2^2 = \sum_{j=1}^m w_j^2$$

$$L1 : \|w\|_1 = \sum_{j=1}^m |w_j|$$

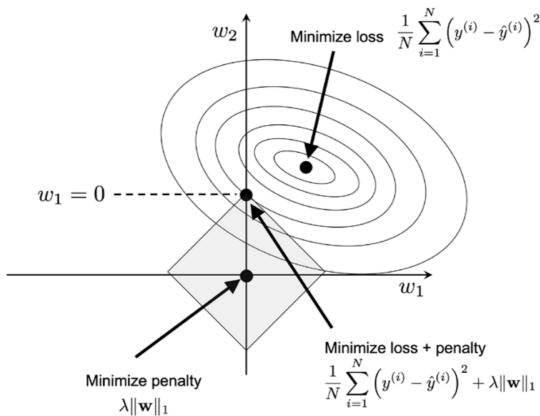
$L1$ регуляризация приводит к очень разреженным векторам, где большая часть значений равна 0. Это особенно полезно, если у нас многомерные данные, где большая часть данных не относится к делу, поэтому её можно воспринимать как технику **feature selection**. Теперь посмотрим на геометрическую интерпретацию $L2$ регуляризации. Давайте нарисуем контурный график для выпуклой функции потерь для двух весов w_1 и w_2 . Так как MSE сферическая, то её проще нарисовать, чем функцию для логистической регрессии, хотя возможно.



Увеличивая λ , мы уменьшаем зависимость модели от тренировочных данных.



Наша регуляризация – как раз затенённый шарик, за него мы не можем выйти. Теперь поговорим про разреженные решения с регуляризацией $L1$. Общая идея такая:



Теперь наши пересечения будут лежать на осях – как раз разреженность. Взглянем на использование этого в коде. Уменьшил параметр C в следующем коде, мы будем добиваться большего количества нулей.

```
from sklearn.linear_model import LogisticRegression
# lbfgs не поддерживает l1 рег., поэтому мы
# используем liblinear
lr = LogisticRegression(penalty='l1',
                        solver='liblinear',
                        multi_class='ovr',
                        C=1.0)

lr.fit(X_train_std, y_train)
print('Training accuracy:', lr.score(X_train_std, y_train))
print('Test accuracy:', lr.score(X_test_std, y_test))
# Training accuracy: 1.0
# Test accuracy: 1.0

print(lr.intercept_)
# У нас 3 intercept (aka bias), потому что
# мы используем ovr:
# [-1.26329805 -1.21585775 -2.37021278]

print(lr.coef_)
# [[ 1.2458383  0.18016963  0.74626522 -1.16410631  0.          0.
#   1.15983577  0.          0.          0.          0.          0.55673342
#   2.5085062 ]
#  [-1.53961776 -0.38547842 -0.9962278   0.36421952 -0.05865063  0.
#   0.66693913  0.          0.          -1.9314525   1.23772498  0.
#   -2.23249442]
#  [ 0.13554393  0.16876127  0.35724241  0.          0.          0.
#   -2.4375721   0.          0.          1.56337501 -0.81815889 -0.49315182
#   0.          ]]
# Видим, что на многих местах стоят нули.
```

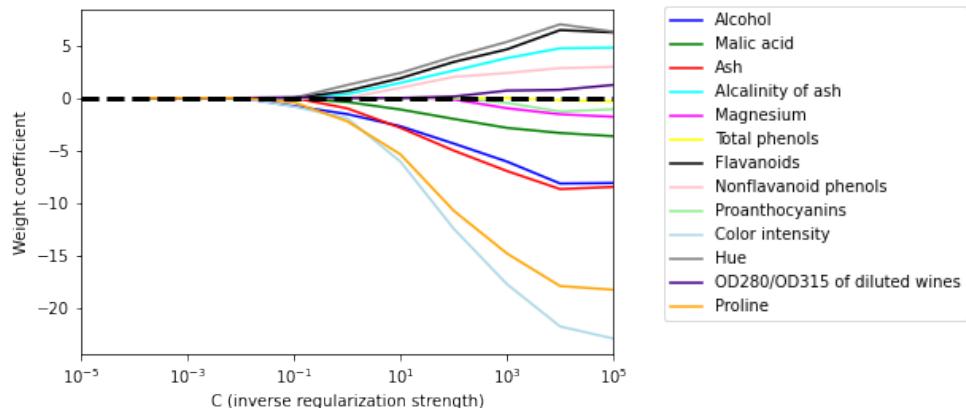
Напишем ещё немного кода и посмотрим на график зависимости w от C при $L1$.

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.subplot(111)
colors = ['blue', 'green', 'red', 'cyan',
```

```

'magenta', 'yellow', 'black',
'pink', 'lightgreen', 'lightblue',
'gray', 'indigo', 'orange']
weights, params = [], []
for c in np.arange(-4., 6.):
    lr = LogisticRegression(penalty='l1', C=10.**c, solver='liblinear',
                            multi_class='ovr', random_state=0)
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[1])
    params.append(10.**c)
weights = np.array(weights)
for column, color in zip(range(weights.shape[1]), colors):
    plt.plot(params, weights[:, column],
              label=df_wine.columns[column + 1],
              color=color)
plt.axhline(0, color='black', linestyle='--', linewidth=3)
plt.xlim([10**(-5), 10**5])
plt.ylabel('Weight coefficient')
plt.xlabel('C (inverse regularization strength)')
plt.xscale('log')
ax.legend(loc='upper center',
          bbox_to_anchor=(1.38, 1.03),
          ncol=1, fancybox=True)
plt.show()

```



Теперь поговорим про **sequential feature selection algorithms**. Ещё один способ избежать переобучения – это dimensionality reduction через feature selection, что очень полезно для нерегуляризируемых моделей. Есть две главные техники – feature selection и feature extraction. Соответственно мы либо выбираем нужные фичи, либо убираем ненужные. Sequential feature selection algorithms – это семейство жадных алгоритмов, которые сокращают d -мерное пространство на k -мерное. Задача – уменьшить overfitting и время работы. В этой группе есть классический алгоритм – **sequential backward selection (SBS)** алгоритм. Его главная цель – повышение вычислительной эффективности с минимальным снижением производительности; также помогает избежать overfitting. Этот **жадный** алгоритм делает серию локальных оптимальных выборов, но не гарантирует самого оптимального глобального выбора. Идея очень простая – мы убираем последовательно фичи пока не достигнем нужного размера, на каждом шаге мы выбираем фичу так, чтобы минимизировать некую функцию J . Функция – разность производительности до и после удаления данной фичи, в итоге убираем ту фичу, у которой эта разность самая большая (\sim самый маленький loss). Итого 4 простых шага:

1. Изначально $k = d$, где X_d – изначальное пространство.
2. Найдём фичу \bar{x} , которая максимизирует критерий: $\bar{x} = \text{argmax} J(X_k - x)$, где $x \in X_k$.
3. Убираем фичу: $X_{k-1} = X_k - \bar{x}$ и $k = k - 1$.
4. Продолжаем пока $k >$ мы хотим фич.

Теперь напишем код SBS:

```

from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

```

```

class SBS:
    def __init__(self, estimator, k_features,
                 scoring=accuracy_score,
                 test_size=0.25, random_state=1):
        self.scoring = scoring
        # Construct a new unfitted estimator with
        # the same parameters. (deep copy)
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state

    def fit(self, X, y):
        # делаем validation dataset
        X_train, X_test, y_train, y_test = \
            train_test_split(X, y, test_size=self.test_size,
                             random_state=self.random_state)

        dim = X_train.shape[1]
        # непосредственно кортежи
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                                 X_test, y_test, self.indices_)
        self.scores_ = [score]
        while dim > self.k_features:
            scores = []
            subsets = []

            # все возможные неупорядоченные комбинации
            # из этих индексов, где мы выбрали ровно r
            # индексов
            for p in combinations(self.indices_, r=dim - 1):
                score = self._calc_score(X_train, y_train,
                                         X_test, y_test, p)
                scores.append(score)
                subsets.append(p)

            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1

            self.scores_.append(scores[best])

        self.k_score_ = self.scores_[-1]
        return self

    def transform(self, X):
        return X[:, self.indices_]

    def _calc_score(self, X_train, y_train, X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score

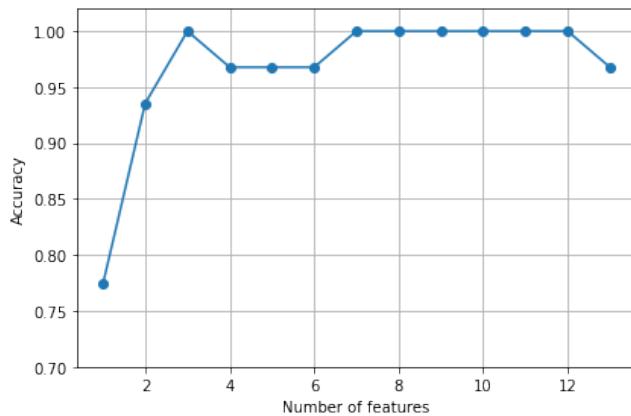
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)
sbs = SBS(knn, k_features=1)
sbs.fit(X_train_std, y_train)

# теперь посмотрим на график результативности
# в зависимости от кол-ва фич
k_feat = [len(k) for k in sbs.subsets_]
plt.plot(k_feat, sbs.scores_, marker='o')
plt.ylim([0.7, 1.02])
plt.ylabel('Accuracy')
plt.xlabel('Number of features')
plt.grid()
plt.tight_layout()
plt.show()

# например, для knn мы по сути
# решили curse of dimensionality

```

На следующем графике мы увидим, что максимальный показатель уже достигается при $k = 3$:



Попишем ещё немного кода и узнаем немного про такой маленький датасет:

```
k3 = list(sbs.subsets[-3])
print(df_wine.columns[1:][k3])
# Вот такие это фичи:
# Index(['Alcohol', 'Malic acid', 'OD280/OD315 of diluted wines'],
#       dtype='object')

knn.fit(X_train_std, y_train)
print('Training accuracy:', knn.score(X_train_std, y_train))
print('Test accuracy:', knn.score(X_test_std, y_test))
# Training accuracy: 0.967741935483871
# Test accuracy: 0.9629629629629629

knn.fit(X_train_std[:, k3], y_train)
print('Training accuracy:', knn.score(X_train_std[:, k3], y_train))
print('Test accuracy:', knn.score(X_test_std[:, k3], y_test))
# Training accuracy: 0.9516129032258065
# Test accuracy: 0.9259259259259259
```

Судя по тому, что мы видим – результаты стали немного хуже, но они всё равно очень хорошие; всё таки не стоит забывать, что мы оставили меньше $\frac{1}{4}$ данных. Можно использовать уже встроенный алгоритм, который можно найти в библиотеке **mlxtend**. Ещё есть множество других похожих алгоритмов: recursive backward elimination based on feature weights, tree-based methods to select features by importance, и univariate statistical tests.

4.5 Оценка с помощью random forests

Научимся оценивать важность фич с помощью леса. Используя random forest, мы можем измерить важность фич как среднее значение уменьшения impurity по всем деревьям в лесу, не делая никаких предположений о том, что данные линейно разделимы или нет. Также вспомним, что данные никак не нужно стандартизировать. Очень удобно, что RandomForestClassifier уже содержит всю необходимую нам информацию. Напишем немного кода:

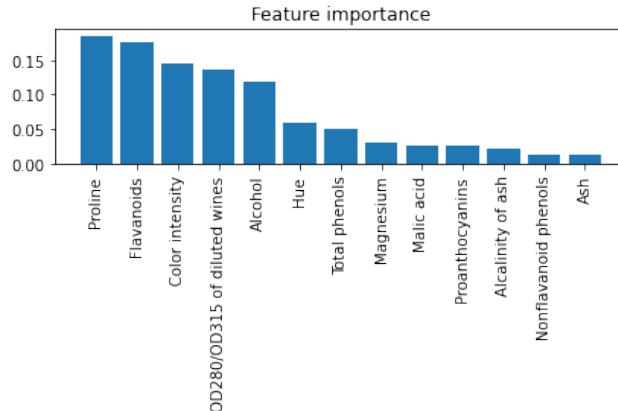
```
from sklearn.ensemble import RandomForestClassifier
feat_labels = df_wine.columns[1:]
forest = RandomForestClassifier(n_estimators=500,
                                 random_state=1)

forest.fit(X_train, y_train)
importances = forest.feature_importances_
# argsort - returns the indices that would sort an array
# [:-1] - reverse массива
indices = np.argsort(importances)[::-1]
# интересный способ форматирования:
# %30s - пробелы будут с начала
# %-30s - пробелы будут с конца
# число можно заменить на * и передать параметром
for f in range(X_train.shape[1]):
    print("%2d) %-*s %f" %
          (f + 1, 30, feat_labels[indices[f]], importances[indices[f]]))
# заметим, что сумма всех важностей даст 1
plt.title('Feature importance')
# center - bar полоски посередине штрихов на цифрах
plt.bar(range(X_train.shape[1]),
        importances[indices],
        align='center')
# rotation - чтобы названия шли под 90 градусов
plt.xticks(range(X_train.shape[1]),
```

```

feat_labels[indices], rotation=90)
plt.xlim([-1, X_train.shape[1]])
plt.tight_layout()
plt.show()

```



Однако есть небольшая ошибка, о которой нужно упомянуть. Если есть несколько фич с похожими значениями, то одна из них может быть высоко оценена, пока остальные останутся менее важными. Однако, если нас интересует только производительность, а не реальная важность фич, то всё хорошо. В заключение, есть такой объект в нашей библиотеке **SelectFromModel**, который выбирает фичи, используя заданный threshold и важности.

```

from sklearn.feature_selection import SelectFromModel
# prefilt - уже обучен
# выбрав 0.1, мы возьмём только первые 5 фич
sfm = SelectFromModel(forest, threshold=0.1, prefit=True)
X_selected = sfm.transform(X_train)
print('Number of features that meet this threshold',
      'criterion:', X_selected.shape[1])
# Number of features that meet this threshold criterion: 5
for f in range(X_selected.shape[1]):
    print("%2d %-*s %f" %
          (f + 1, 30, feat_labels[indices[f]], importances[indices[f]]))

```

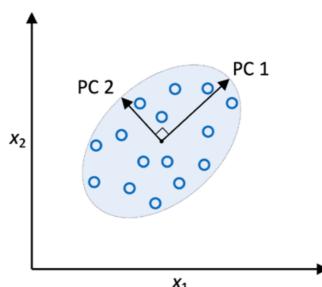
В следующей части мы изучим ещё один подход к dimensionality reduction – **feature extraction**, это позволяет нам сжать наше пространство на меньшее подпространство, а не просто убрать фичи как сейчас.

5 Compressing Data via Dimensionality Reduction

Мы уже прошли feature selection, а теперь пройдём feature extraction.

5.1 Unsupervised dimensionality reduction via principal component analysis

На практике эти разные алгоритмы могут помочь не только считать эффективнее, но и лучше предсказывать, преодолевая проклятие размерностей. Посмотрим на главные шаги в **principal component analysis (PCA)**. PCA – техника линейной трансформации, применяют также для шумоподавления сигнала, фондовых рынках, биоинформатике. Он выявляет шаблоны, основываясь на корреляции фич. PCA находит основные направления максимального variance, а дальше делает проекцию на пространство, размерность которого меньше либо равна нашего. Ортогональные оси (главные компоненты) нового подпространства можно интерпретировать как направления максимального variance, учитывая, что они ортогональны.



Когда мы работаем с PCA, мы строим **transformation matrix** W размерности $d \times k$, которая позволяет мэпить вектора в новое пространство. Немного формул: $x = [x_1, x_2, \dots, x_d] \in \mathbb{R}^d \Rightarrow xW = z \Rightarrow z = [z_1, z_2, \dots, z_k] \in \mathbb{R}^k$. После преобразования первая главная компонента будет иметь максимальный variance, последующие компоненты тоже будут иметь максимальный variance, но учитывая, что всё должно быть ортогонально (даже, если входные данные коррелированы, то выход некоррелированный). Важно сделать feature scaling. Выделим главные 7 шагов:

1. Стандартизуем d – мерное пространство.
2. Сделаем covariance matrix, она симметричная. ($A = A^T$)
3. Разложим ковариационную матрицу на ее собственные векторы и собственные числа.
4. Сортируем с.ч. в порядке убывания, чтобы ранжировать вектора.
5. Выбираем k первых векторов.
6. Сделаем матрицу проектирования W из этих k векторов.
7. Трансформируем пространства.

После разложения ковариационной матрицы мы получим вещественные с.ч., а все вектора будут взаимо перпендикулярны, при этом и с.ч. и с.в. идут парами. С.в. с самыми большими с.ч. ассоциируются с самым большим variance в датасете. Это направление – линейная трансформация колонки этой фичи. Симметричная матрица размера $d \times d$ (covariance matrix) хранит попарные ковариации между разными фичами x_j и x_k :

$$\sigma_{jk} = \frac{1}{n-1} \sum_{i=1}^n (x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k)$$

В формуле μ – средние значения, но если мы сделали стандартизацию, то эти значения равны 0. Если ковариация > 0 , то эти значения усиливают друг друга, в противном случае они действуют в разных направлениях. Например, для трёх фич выглядят так:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

В итоге с.в. будут давать главные компоненты, а с.ч. показывать магнитуду. Из курса лин. алгебры мы знаем, что $\Sigma v = \lambda v$. Напишем код:

```
import numpy as np
cov_mat = np.cov(X_train_std.T)
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
print('Eigenvalues:\n', eigen_vals)
print('Eigenvectors:\n', eigen_vecs)
# Eigenvalues:
# [4.84274532 2.41602459 1.54845825 0.96120438 0.84166161 0.6620634
# 0.51828472 0.34650377 0.3131368 0.10754642 0.21357215 0.15362835
# 0.1808613 ]
```

Вообще в питоне есть две функции: eig и eigh, первая может работать с любыми матрицами, выдаёт она комплексные числа, вторая функция сделана для декомпозиции hermitian матриц, чтобы более стабильно работать для симметричных матриц нужно использовать именно её – функция всегда даётвещ. числа. Теперь поговорим про **total** и **explained variance**. Самые длинные с.в. дают нам самый большой variance, а это в свою очередь информация. Построим график variance explained ratios, это в свою очередь просто отношение текущего с.ч. к сумме остальных:

$$\text{Explained variance ratio} = \frac{\lambda_j}{\sum_{j=1}^d \lambda_j}$$

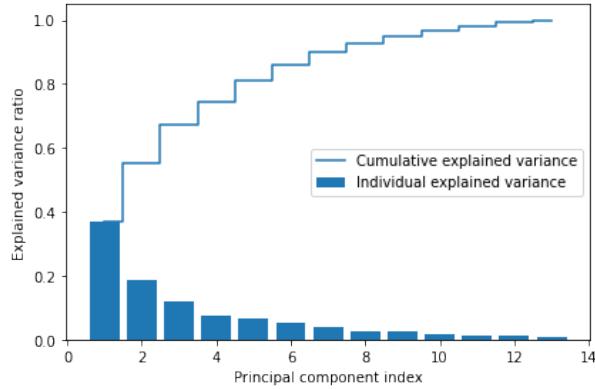
Построим ступенчатый график cumulative сумм (префиксных сумм) этих коэффициентов:

```
tot = sum(eigen_vals)
var_exp = [(i / tot) for i in sorted(eigen_vals, reverse=True)]
# префиксная сумма:
cum_var_exp = np.cumsum(var_exp)
import matplotlib.pyplot as plt
plt.bar(range(1, 14), var_exp, align='center',
        label='Individual explained variance')
# кусочная функция, mid – посередине:
```

```

plt.step(range(1, 14), cum_var_exp, where='mid',
         label='Cumulative explained variance')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal component index')
plt.legend(loc='best')
plt.tight_layout()
plt.show()

```



Необходимо помнить, что этот алгоритм в отличие от того, что из главы 4, игнорирует значения методов. Дисперсия измеряет разброс значений фич по оси. Следующий шаг – **feature transformation**. Напишем код для последних 3 пунктов:

```

eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
               for i in range(len(eigen_vals))]
eigen_pairs.sort(key=lambda k: k[0], reverse=True)
# далле мы выберем 2 фичи, это только для иллюстрации
# на самом деле – это tradeoff между качеством и
# производительностью

# np.newaxis – это псевдоним None, используется, чтобы
# генерировать новую размерность: в нашем случае это
# столбец, но если убрать :, то это будет строка:
w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
                eigen_pairs[1][1][:, np.newaxis]))
print('Matrix W:\n', w)
# Matrix W:
# [[-0.13724218  0.50303478]
# ...
# [-0.29669651  0.38022942]]

```

В зависимости от версий библиотек можно получить либо матрицу W , либо её же, но с обратными знаками – это не проблема. Пусть у нас $\Sigma v = \lambda v$, тогда $-v$ – тоже с.в. Покажем это: $\alpha \Sigma v = \alpha \lambda v \Leftrightarrow \Sigma(\alpha v) = \lambda(\alpha v)$, а это и значит, что это с.в. для того же с.ч. при $\alpha = \pm 1$. Сделаем переходы $x' = xW$ и $X' = XW$ в коде:

```

X_train_std[0].dot(w)
# array([2.38299011, 0.45458499])
X_train_pca = X_train_std.dot(w)

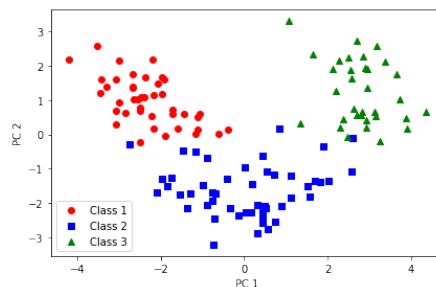
```

Теперь визуализируем:

```

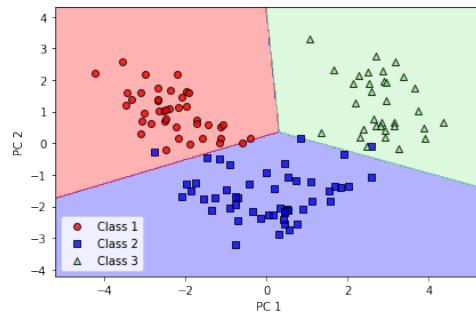
colors = ['r', 'b', 'g']
markers = ['o', 's', '^']
for l, c, m in zip(np.unique(y_train), colors, markers):
    plt.scatter(X_train_pca[y_train==l, 0],
                X_train_pca[y_train==l, 1],
                c=c, label=f'Class {l}', marker=m)
plt.xlabel('PC 1')
plt.ylabel('PC 2')

```



Можно также заметить, что наши данные более растянуты по оси x . Реализуем тоже самое из коробки:

```
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
lr = LogisticRegression(multi_class='ovr',
                        random_state=1,
                        solver='lbfgs')
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)
lr.fit(X_train_pca, y_train)
plot_decision_regions(X_train_pca, y_train, classifier=lr)
```

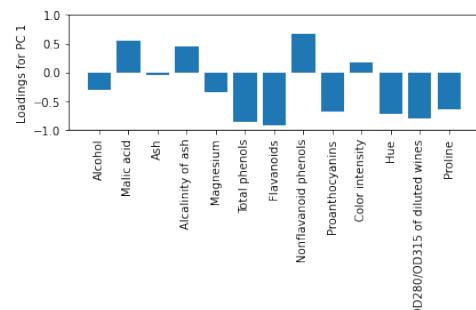


Сравнивая PCA проекции разных решений данной задачи, мы можем заметить, что графики являются зеркальными отражениями друг друга, потому что в зависимости от алгоритма поиска с.ч. с.в. могут быть либо больше либо меньше нуля. Чтобы это исправить, нужно умножить данные на -1 ; также заметим, что обычно с.в. нормированы. Если мы выведем границы на тестовых данных, то результат тоже будет очень хороший. Как можно узнать explained variance ratios – поставим кол-во компонент в None, чтобы сохранить все фичи:

```
pca = PCA(n_components=None)
X_train_pca = pca.fit_transform(X_train_std)
pca.explained_variance_ratio_
# array([0.36951469, 0.18434927, 0.11815159, 0.07334252, 0.06422108,
#        0.05051724, 0.03954654, 0.02643918, 0.02389319, 0.01629614,
#        0.01380021, 0.01172226, 0.00820609])
```

То есть вместо сжатия мы вывели все pc в отсортированном порядке. Теперь посмотрим на вклад изначальных фич в pc. Как мы уже знаем, мы создаём pc, которые представляют линейные комбинации исходных фич. Иногда хочется знать вклад этих фич – это называется **loadings**. Их можно посчитать, умножив с.в. на корень из с.ч. Напишем код, в котором посчитаем матрицу размера 13×13 , а потом нарисуем loadings для первого pc – первая колонка матрицы:

```
loadings = eigen_vecs * np.sqrt(eigen_vals)
fig, ax = plt.subplots()
# первая колонка:
ax.bar(range(13), loadings[:, 0], align='center')
ax.set_ylabel('Loadings for PC 1')
ax.set_xticks(range(13))
ax.set_xticklabels(df_wine.columns[1:], rotation=90)
plt.ylim([-1, 1])
plt.tight_layout()
plt.show()
# Alcohol, например, имеет отрицательную корреляцию
# с первым pc, а вот Malic acid – положительную
```



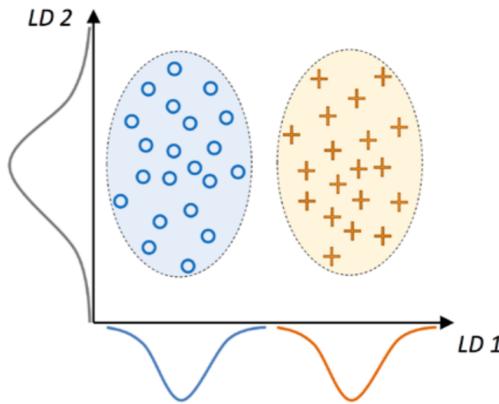
Теперь тоже самое только из библиотечного pca:

```
# explained_variance_ - с.ч.
# components_ - с.в.
sklearn_loadings = pca.components_.T * np.sqrt(pca.explained_variance_)
```

Немного объяснений: после составления матрицы ковариаций мы получаем с.ч. и с.в. Каждый с.в. соответствует каждой фиче, а координаты с.в. есть корреляция относительно исходных фич. Дальше мы пройдём **linear discriminant analysis (LDA)** – техника линейной трансформации, которая уже учитывает метки.

5.2 Supervised data compression via LDA

LDA – техника feature extraction для повышения вычислительной эффективности и для уменьшения переобучения. Общая идея похожа на PCA, но главная цель LDA – найти подпространство фич, которое оптимизирует **class separability**. Теперь обсудим такой топик: PCA vs LDA. Казалось бы, что так как LDA – supervised, то он лучше, но, например, при работе с изображениями, где для каждого класса мало примеров, PCA работает лучше. Кстати, иногда LDA называют ещё Fisher LDA, который его и придумал. Общая идея LDA такая:



Второй линейный discriminant имеет очень большую вариацию по сравнению с первым. Важно, что данные нормально распределённые, тренировочные данные статистически независимы и что классы имеют идентичные ковариационные матрицы. Но, как показывает практика, при небольшом отклонении от требований всё будет хорошо. Опять выделим главные 7 пунктов:

1. Стандартизуем d – мерный датасет.
2. Для каждого класса посчитаем d – мерный mean vector.
3. Сделаем **between-class** scatter matrix S_B , а также **within-class** scatter matrix S_W .
4. Разложим на с.ч. и с.в. матрицу $S_W^{-1}S_B$.
5. Отсортируем в обратном порядке с.ч.
6. Возьмём k самых больших с.в., сделаем transformation matrix размера $d \times k$; с.в. – столбики матрицы.
7. Спроектируем на новое пространство.

Как и обещалось в пункте 2 мы используем значения меток. Узнаем как считать **scatter matrices**. Каждый mean vector m_i хранит среднее значение фичи μ_m для класса i :

$$m_i = \frac{1}{n_i} \sum_{x \in D_i} x_m$$

Что даёт нам три таких вектора:

$$m_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix}, i \in \{1, 2, 3\}$$

Сделаем это в коде:

```

np.set_printoptions(precision=4)
mean_vecs = []
for label in range(1, 4):
    mean_vecs.append(np.mean(
        X_train_std[y_train==label], axis=0))
print(f'MV {label}: {mean_vecs[label-1]}\n')
# MV 1: [ 0.9066 -0.3497  0.3201 -0.7189  0.5056  0.8807  0.9589 -0.5516  0.5416
#      0.2338  0.5897  0.6563  1.2075]
# ...

```

Теперь посчитаем within-class scatter matrix S_W :

$$S_W = \sum_{i=1}^c S_i$$

Это сумма individual scatter matrices:

$$S_i = \sum_{x \in D_i} (x - m_i)(x - m_i)^T$$

Напишем код:

```

# кол-во фич:
d = 13
S_W = np.zeros((d, d))
for label, mv in zip(range(1, 4), mean_vecs):
    class_scatter = np.zeros((d, d))
    for row in X_train_std[y_train == label]:
        # делаем столбики:
        row, mv = row.reshape(d, 1), mv.reshape(d, 1)
        # пару слов про dot - это как раз умножение по
        # правилу строки на столбец, а * даёт как бы
        # по элементное
        class_scatter += (row - mv).dot((row - mv).T)
    S_W += class_scatter
print('Within-class scatter matrix: ',
      f'{S_W.shape[0]}x{S_W.shape[1]}')
# Within-class scatter matrix: 13x13

```

Когда мы считаем эту матрицу, мы делаем предположение, что данные нормально распределённые, но как мы видим это нарушается:

```

print('Class label distribution: ',
      np.bincount(y_train)[1:])
# Class label distribution: [41 50 33]

```

Поэтому мы хотим нормировать индивидуальные матрицы перед сложением с главной матрицей. Но, когда мы делим scatter matrices на n_i , то можно увидеть, что Σ_i ковариационная матрица – нормализованная версия scatter matrix:

$$\Sigma_i = \frac{1}{n_i} S_i = \frac{1}{n_i} \sum_{x \in D_i} (x - m_i)(x - m_i)^T$$

Напишем код для нормализованных матриц:

```

d = 13
S_W = np.zeros((d, d))
for label, mv in zip(range(1, 4), mean_vecs):
    class_scatter = np.cov(X_train_std[y_train==label].T)
    S_W += class_scatter

```

Теперь давайте посчитаем between-class scatter matrix S_B :

$$S_B = \sum_{i=1}^c n_i(m_i - m)(m_i - m)^T$$

Здесь m – общее среднее для всех классов c . Напишем код и для неё:

```

mean_overall = np.mean(X_train_std, axis=0)
mean_overall = mean_overall.reshape(d, 1)

d = 13
S_B = np.zeros((d, d))
for i, mean_vec in enumerate(mean_vecs):
    # это кол-во примеров в каждом классе c

```

```

n = X_train_std[y_train == i + 1, :].shape[0]
mean_vec = mean_vec.reshape(d, 1) # column vector
S_B += n * (mean_vec - mean_overall).dot(
    (mean_vec - mean_overall).T)
print('Between-class scatter matrix: ',
f'{S_B.shape[0]}x{S_B.shape[1]}')

```

Теперь поговорим про выбор linear discriminants для нового подпространства фич. Остальные шаги похожи на PCA, но здесь мы ищем с.ч. у матрицы $S_W^{-1}S_B$:

```

eigen_vals, eigen_vecs =\
    np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
    for i in range(len(eigen_vals))]
eigen_pairs = sorted(eigen_pairs,
    key=lambda k: k[0], reverse=True)
print('Eigenvalues in descending order:\n')
for eigen_val in eigen_pairs:
    print(eigen_val[0])
# Eigenvalues in descending order:

# 349.61780890599397
# 172.7615221897938
# 3.3428382148413644e-14
# 2.842170943040401e-14
# 2.5545786180111397e-14
# 1.753393918073425e-14
# 1.753393918073425e-14
# 1.657919399596089e-14
# 1.657919399596089e-14
# 8.242524002707208e-15
# 8.242524002707208e-15
# 6.36835506006027e-15
# 2.97463437554573e-15

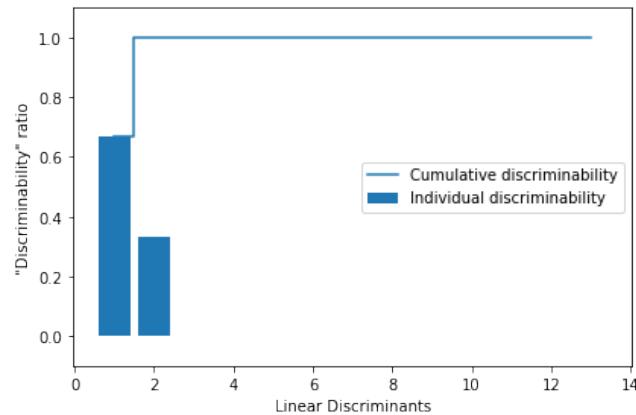
```

В LDA число linear discriminants $\leq c-1$, т.к. матрица S_B – сумма с матриц ранга 1 или меньше. Вот в нашем примере у нас лишь 2 числа не равны 0, остальные должны быть равны 0, но есть ошибка из-за float чисел. Лишь в редком случае, когда все точки на одной прямой, ков. матрица будет иметь ранг 1, а это значит, что будет лишь 1 ненулевое с.ч. Давайте посмотрим сколько информации взяли linear discriminants (eigenvectors) по аналогии с графиком из прошлой секции:

```

tot = sum(eigen_vals.real)
discr = [(i / tot) for i in sorted(eigen_vals.real,
    reverse=True)]
cum_discr = np.cumsum(discr)
plt.bar(range(1, 14), discr, align='center',
    label='Individual discriminability')
plt.step(range(1, 14), cum_discr, where='mid',
    label='Cumulative discriminability')

```



Как мы видим – первые два линейных дискриминанта дают 100% полезной информации. Теперь соберём матрицу W, спроектируем и посмотрим на картинку:

```

w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
    eigen_pairs[1][1][:, np.newaxis].real))
print('Matrix W:\n', w)

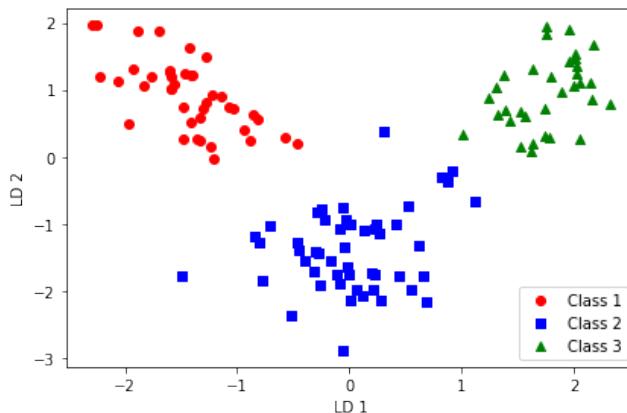
X_train_lda = X_train_std.dot(w)
colors = ['r', 'b', 'g']

```

```

markers = [ 'o' , 's' , '^' ]
for l , c , m in zip(np.unique(y_train) , colors , markers):
    plt.scatter(X_train_lda[y_train==l , 0] ,
                X_train_lda[y_train==l , 1] * (-1) ,
                c=c , label=f'Class {l}' , marker=m)

```



Видно, что здесь всё идеально разделяется линейной границей. Теперь сделаем тоже самое только из коробки:

```

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
lda = LDA(n_components=2)
X_train_lda = lda.fit_transform(X_train_std , y_train)
lr = LogisticRegression(multi_class='ovr' , random_state=1,
                        solver='lbfgs')

```

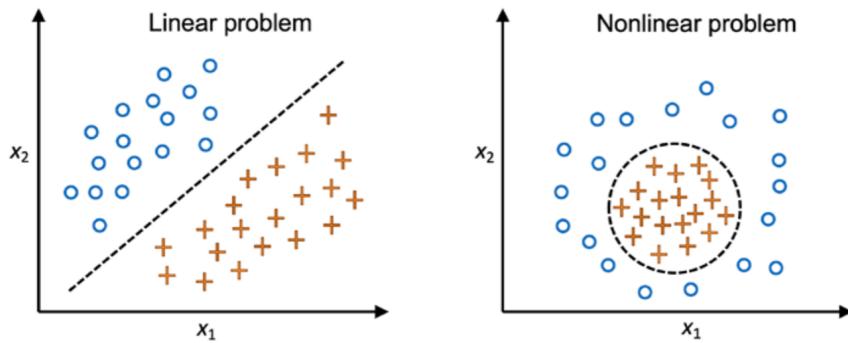
Можно попробовать регулировать результат на тренировочных данных с помощью регуляризации. Теперь запустим на тестовых данных:

```
X_test_lda = lda.transform(X_test_std)
```

Если вывести границы, то можно увидеть, что мы разделим 100% тестовых данных.

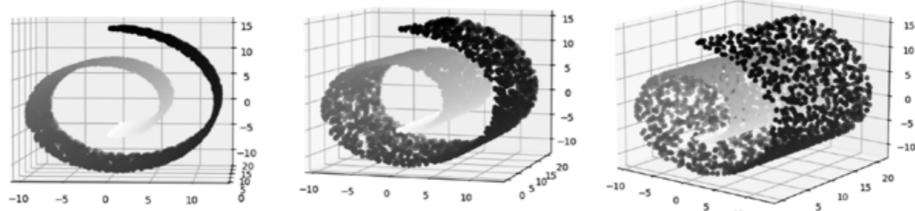
5.3 Nonlinear dimensionality reduction and visualization

T-distributed stochastic neighbor embedding (t-SNE) – одна из нелинейных техник, которая заслуживает внимания, т.к. она часто используется для визуализации многомерных данных на 2-х или 3-х мерное пространство. Мы посмотрим как построить написанные от руки изображения на 2-мерное пр. Многие алгоритмы ml предполагают, что данные линейно разделимы, поэтому для нелинейных данных PCA или LDA могут не подойти:

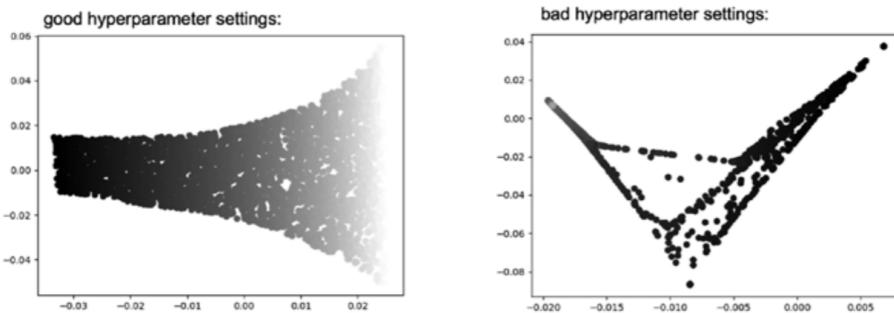


Такие алгоритмы уменьшения размерности относятся к **manifold learning**, они должны уметь выделить сложную структуру, чтобы потом хорошо спроектировать. Классический пример manifold learning – это 3-х мерный Swiss roll. С другой стороны эти алгоритмы крайне сложны в использовании, поэтому при неправильном подборе гиперпараметров мы можем сделать только хуже. Часто не хватает 2-3 фич, чтобы взять основные параметры из датасета, а тогда мы не можем нормально визуализировать, а следовательно и оценить результат.

Different views of a 3-dimensional Swiss roll:



Swiss roll projected onto a 2-dimensional feature space with ...



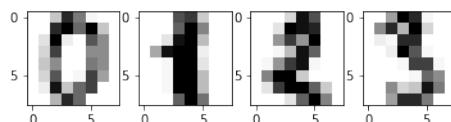
Теперь рассмотрим визуализацию через t-SNE. В двух словах t-SNE генерирует точки на основе попарных расстояний в оригинальном пространстве. Затем он находит распределение вероятностей попарных расстояний в новом пространстве, которое близко к исходному в исходном пространстве. Другими словами мы учимся встраивать точки так, чтобы сохранялись попарные расстояния. Это метод на самом деле для визуализации т.к. он требует весь набор. Так как он проецирует точки напрямую, то мы не можем добавить новых точек. Рассмотрим то, как можно его применить для рукописных цифр. Для начала получим данные:

```
from sklearn.datasets import load_digits
digits = load_digits()

fig, ax = plt.subplots(1, 4)
for i in range(4):
    # 8 на 8 pixels
    ax[i].imshow(digits.images[i], cmap='Greys')
plt.show()

# можно получить и табличную версию данных
# где строки - это примеры, а колонки - пиксели:
digits.data.shape

y_digits = digits.target
X_digits = digits.data
```



Теперь сделаем преобразование:

```
from sklearn.manifold import TSNE
# мы инициализируем t-SNE embedding
# с помощью PCA
tsne = TSNE(n_components=2, init='pca',
            random_state=123)
X_digits_tsne = tsne.fit_transform(X_digits)
```

Важно отметить, что есть еще есть два гиперпараметра: perplexity и learning rate. Теперь визуализируем:

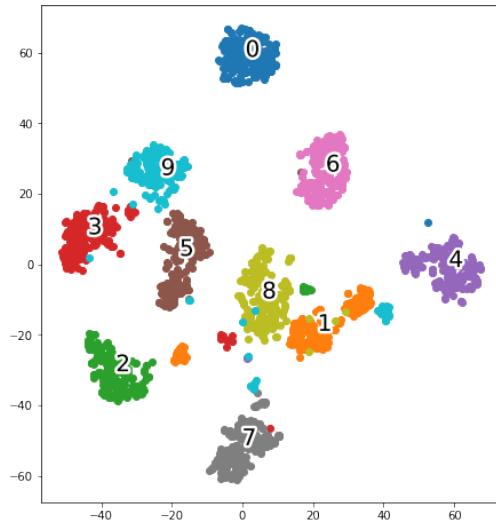
```
import matplotlib.path_effects as PathEffects
def plot_projection(x, colors):
    f = plt.figure(figsize=(8, 8))
    # одинаковые оси по размеру - aspect
    ax = plt.subplot(aspect='equal')
    for i in range(10):
        plt.scatter(x[colors == i, 0],
```

```

x[ colors == i , 1])

for i in range(10):
    xtext , ytext = np.median(x[ colors == i , : ] , axis=0)
    txt = ax.text(xtext , ytext , str(i) , fontsize=20)
    # Normal – слой обычного числа без эффектов
    # Stroke – ещё один слой
    txt.set_path_effects([
        PathEffects.Stroke(linewidth=5, foreground="w") ,
        PathEffects.Normal()])
plot_projection(X_digits_tsne , y_digits)
plt.show()

```



t-SNE – это unsupervised метод, метки нужны были только, чтобы вывести цифры. Другая популярная техника визуализации – это **uniform manifold approximation and projection (UMAP)**. Его плюс в том, что он быстрее, а также может быть использован, чтобы проецировать новые данные.

6 Model Evaluation and Hyperparameter Tuning

6.1 Streamlining workflows with pipelines

Рассмотрим оптимизацию рабочих процессов с помощью конвейеров. В предыдущих главах, когда мы делали стандартизацию и PCA – preprocessing techniques, мы узнали, что нужно переиспользовать параметры, полученные при fitting и т.д. Теперь мы пройдём Pipeline, который позволит обучать, вставляя произвольное кол-во шагов, а также делать предсказания о новых данных. Сначала загрузим новый датасет. Этот датасет о раке, в обозначениях B=benign (доброкачественная), M=malignant (злокачественная). Немного кода:

```

import pandas as pd
df = pd.read_csv('https://archive.ics.uci.edu/ml/'
                  'machine-learning-databases'
                  '/breast-cancer-wisconsin/wdbc.data',
                  header=None)

from sklearn.preprocessing import LabelEncoder
X = df.loc[:, 2: ].values
y = df.loc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
print('Classes:', le.classes_) # Classes: ['B' 'M']
print('Encoded:', le.transform(['M', 'B'])) # Encoded: [1 0]

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.20,
                     stratify=y, random_state=1)

```

Теперь скомбинируем transformers и estimators в pipeline. Безусловно мы хотим сжать наше пространство, потому что 30 размерностей явно много. Чтобы не делать 2 раза одни и те же вещи и для трен. и для тест. данных, соединим всё в pipeline:

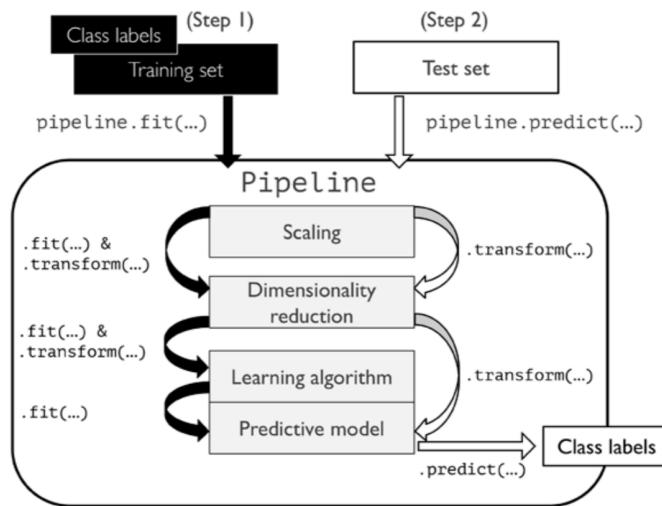
```

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

pipe_lr = make_pipeline(StandardScaler(),
                       PCA(n_components=2),
                       LogisticRegression())
pipe_lr.fit(X_train, y_train)
y_pred = pipe_lr.predict(X_test)
test_acc = pipe_lr.score(X_test, y_test)
# после : мы указали спецификацию
print(f'Test accuracy: {test_acc:.3f}')
# Test accuracy: 0.956

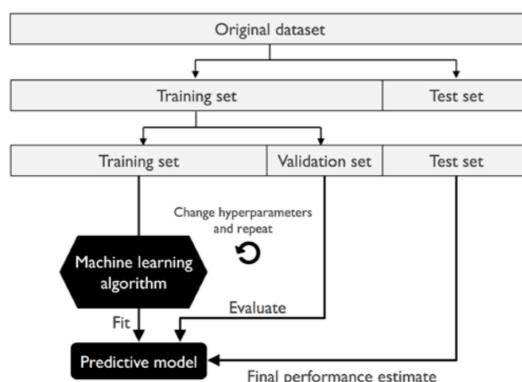
```

Функция `make_pipeline` принимает произвольное кол-во transformers (объект, у которого есть `fit` и `transform`), после которых идёт estimator (тоже с `fit` и `predict`). Мы можем думать о pipeline как о meta-estimator или wrapper. Теперь картинка для пояснения:



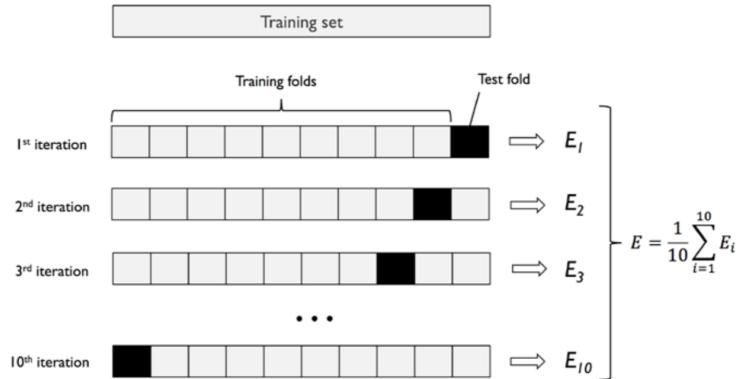
6.2 k-fold cross-validation

В этой части мы узнаем о двух cross-validation техниках: **holdout cross-validation** и **k-fold cross-validation**, которые помогут нам точно оценить качество обобщения на новые данные. Поговорим сначала о первом. Чаще всего мы делим данные на train и test, на втором мы оцениваем качество. Часто мы также заинтересованы в подборе разных параметров с целью улучшения качества – **model selection**. Однако, если мы будем использовать test датасет снова и снова, он всё равно станет частью train датасета. Поэтому лучше всего делить на 3 части: train, test, validation датасеты. Validation датасет используется как раз для подбора параметров. Минус в том, что оценка очень чувствительна к тому, как мы разделим train датасет на train и validation. Картинка для лучшего понимания:



Дальше мы рассмотрим k-fold cross-validation, где мы повторяем holdout метод k раз на k подмножествах тренировочных данных. В этом методе мы разбиваем train датасет на k folds без replacement (обратно не кладём). $k - 1$ folds – training folds, одна оставшаяся – test fold. Процедуру повторяем k

раз. Чаще всего используется для model tuning. Как только мы нашли нужные параметры – перетренируем на всём train сете, а оценку сделаем на test сете. Плюс в том, что на каждой итерации каждый пример используется ровно 1 раз. Картинка:



Эмпирические данные показывают, что хороший выбор – это $k = 10$. Получается хороший tradeoff между bias и variance. Если датасет маленький, то нужно увеличить k , а если большой, то можно уменьшить до $k = 5$. Увеличивая k , мы уменьшаем bias, но делаем больше variance (т.к. train folds становятся очень похожими). Отдельный случай – **leave-one-out cross-validation (LOOCV)**, где $k = n$ – для маленьких сетов. Маленькое улучшение – stratified k-fold cross-validation, где мы сохраняем все пропорции между классами в train сете, когда делим на folds. Напишем код:

```
import numpy as np
from sklearn.model_selection import StratifiedKFold
# вернёт индексы
kfold = StratifiedKFold(n_splits=10).split(X_train, y_train)
scores = []
for k, (train, test) in enumerate(kfold):
    pipe_lr.fit(X_train[train], y_train[train])
    score = pipe_lr.score(X_train[test], y_train[test])
    scores.append(score)
    print(f'Fold: {k+1:02d}, '
          f'Class distr.: {np.bincount(y_train[train])}, '
          f'Acc.: {score:.3f}')

mean_acc = np.mean(scores)
std_acc = np.std(scores)
print(f'\nCV accuracy: {mean_acc:.3f} +/- {std_acc:.3f}\n')

# Fold: 01, Class distr.: [256 153], Acc.: 0.935
# ...
# Fold: 10, Class distr.: [257 153], Acc.: 0.956
# CV accuracy: 0.950 +/- 0.014
```

Важно, что в класс мы передали train данные. Код с похожим результатом можно написать короче:

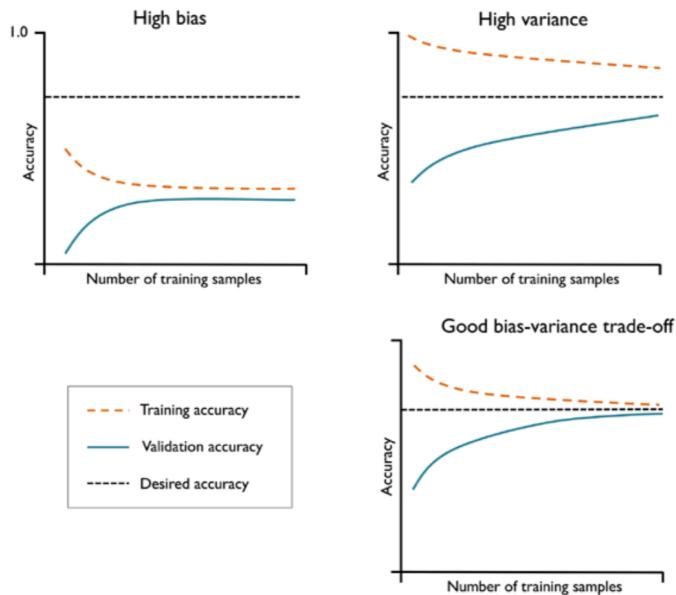
```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=pipe_lr,
                         X=X_train, y=y_train,
                         cv=10, n_jobs=1)
print(f'CV accuracy scores: {scores}')
print(f'CV accuracy: {np.mean(scores):.3f} +/- {np.std(scores):.3f}')
# CV accuracy scores: [0.93478261 ... 0.95555556]
# CV accuracy: 0.950 +/- 0.014
```

Главный плюс этой реализации в том, что можно разбить вычисления на несколько central processing units (CPUs). Если поставить $n_jobs = -1$, то это задействует все имеющиеся CPU.

6.3 Learning and validation curves

В этой части мы рассмотрим два популярных diagnostic tools: **learning curves** и **validation curves**. Сначала мы посмотрим на то, как learning curves помогают узнать о high bias или variance, а далее глянем на validation curves, которые показывают различные типичные проблемы в обучении.

Диагностика bias и variance с помощью learning curves. Построив график зависимости train и validation accuracies в зависимости от размера train сете, можно узнать, где алгоритм страдает от high bias, а где от high variance. Это помогает понять – нужно собирать ещё данных или нет. Теперь посмотрим на картинку, на ней видны 3 возможных сценария:



Левый верхний график: плохие показатели и на train и val. сетах; можно решить, увеличив кол-во параметров (собрать или сконструировать) или уменьшив регуляризацию (например, в svm или lr). График правее: большой разрыв между кривыми; можно решить, собрав новые данные, уменьшив сложность модели или увеличив регуляризацию. Для нерегуляризируемых моделей можно уменьшить размерность датасета. Новые данные обычно уменьшают overfitting, но не всегда: бывает, что в данных много шума или модель уже и так оптимальная. Напишем код:

```

import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve
# max_iter=10000:
# по умолчанию 1000, мы изменили, чтобы избежать
# проблем со сходимостью для меньших размеров сета
# или при экстремальных знач. параметров
# регуляризации (см. след. часть)
pipe_lr = make_pipeline(StandardScaler(),
                        LogisticRegression(
                            penalty='l2',
                            max_iter=10000))
# параметр train_size – когда генерировать
# новую точку кривой

# по умолчанию learning_curve использует
# stratified k-fold cv, чтобы оценить точность

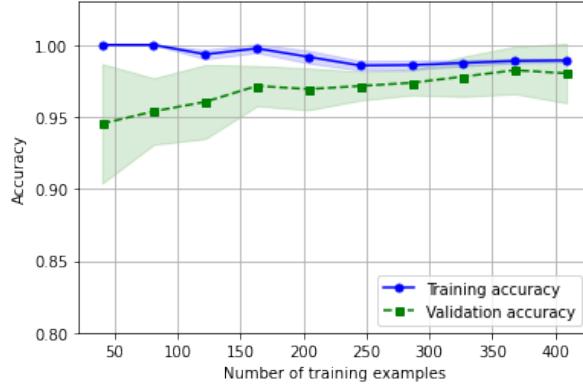
# train_sizes – позиция новых точек л.с.
train_sizes, train_scores, test_scores = \
    learning_curve(
        estimator=pipe_lr,
        X=X_train, y=y_train,
        train_sizes=np.linspace(0.1, 1.0, 10),
        cv=10, n_jobs=1)
# мы разделили на 10 folds, в каждом разделении
# получили точность:
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
plt.plot(train_sizes, train_mean,
         color='blue', marker='o',
         markersize=5, label='Training accuracy')
# чтобы обозначить variance оценки
plt.fill_between(train_sizes,
                 train_mean + train_std,
                 train_mean - train_std,
                 alpha=0.15, color='blue')
plt.plot(train_sizes, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='Validation accuracy')
plt.fill_between(train_sizes,
                 test_mean + test_std,
                 test_mean - test_std,

```

```

alpha=0.15, color='green')
plt.grid()
plt.xlabel('Number of training examples')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim([0.8, 1.03])
plt.show()

```

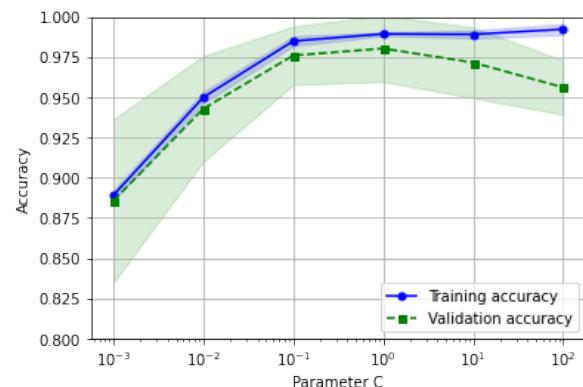


Теперь поговорим про over и underfitting в рамках validation curves. Эта кривая очень похожа на learning curve, но в ней мы меняем не размер сета, а различные параметры модели, например, C . Напишем код:

```

# A validation curve plot for the SVM hyperparameter C:
from sklearn.model_selection import validation_curve
param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
# наш классификатор - LogisticRegression
# наш параметр - C
# => мы получили param_name
train_scores, test_scores = validation_curve(
    estimator=pipe_lr,
    X=X_train, y=y_train,
    param_name='logisticregression__C',
    param_range=param_range, cv=10)
# ...
plt.plot(param_range, ...)
plt.fill_between(param_range, ...)
plt.plot(param_range, ...)
plt.fill_between(param_range, ...)
plt.grid()
plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('Parameter C')
plt.ylabel('Accuracy')
plt.ylim([0.8, 1.0])
plt.show()

```



6.4 Grid search

Параметры, которые мы можем менять называются tuning parameters или hyperparameters, это, например, C или глубина дерева решений. В этой части мы посмотрим на hyperparameter optimization

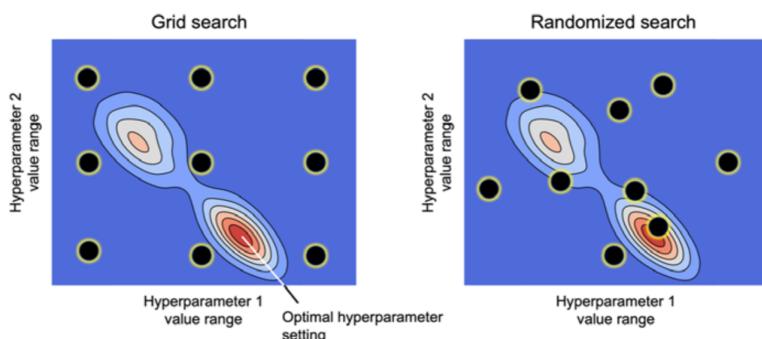
technique – **grid search**, которая ищет оптимальную комбинацию гиперпараметров. Идея очень простая: мы задаём списки значений для каждого параметра, алгоритм перебирает все комбинации и выбирает лучшую модель. Напишем код:

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
pipe_svc = make_pipeline(StandardScaler(),
                        SVC(random_state=1))
param_range = [float(10**i) for i in range(-4, 4)]
param_grid = [
    {'svc__C': param_range,
     'svc__kernel': ['linear']},
    {'svc__C': param_range,
     # в rbf ядре параметр gamma:
     'svc__gamma': param_range,
     'svc__kernel': ['rbf']}]
# параметр refit:
# refit an estimator using the best
# found parameters on the whole dataset

# n_jobs=None для одного ядра
gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=10, refit=True,
                  n_jobs=-1)
# gs использует k-fold cross-validation
# среди 10 folds он считает итоговое
# значение, используя scoring
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
# Выход:
# 0.9846859903381642
# {'svc__C': 100.0, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}

clf = gs.best_estimator_
# fit делать необязательно, refit=True
clf.fit(X_train, y_train)
print(f'Test accuracy: {clf.score(X_test, y_test):.3f}')
# Выход:
# Test accuracy: 0.974
```

Теперь поговорим про **randomized search**. Grid search гарантирует оптимальный estimator, т.к. перебирает все варианты, но это вычислительно очень дорого. Поэтому мы применяем randomized search:



Минус обычного поиска ещё в том, что он перебирает только дискретный набор значений, пропуская оптимальные конфигурации. Опять давайте протестируем на SVM, в scikit-learn есть похожий на обычный grid search класс, но есть отличия: можно указать кол-во конфигураций, которое нужно вычислить, а также distributions:

```
import scipy
param_range = [0.0001, 0.001, 0.01, 0.1,
               1.0, 10.0, 100.0, 1000.0]
# но можно заменить его на distribution
# из SciPy:
param_range = scipy.stats.loguniform(0.0001, 1000.0)
# плюс loguniform distribution по сравнению с обычным
# в том, что он гарантирует в случае большого кол-ва
# испытаний одинаковое кол-во примеров как из [0.0001, 0.001]
# так и из [10.0, 100.0]
```

```

np.random.seed(1)
print(param_range.rvs(10))
# [8.30145146e-02 1.10222804e+01 1.00184520e-04 1.30715777e-02
# 1.06485687e-03 4.42965766e-04 2.01289666e-03 2.62376594e-02
# 5.98924832e-02 5.91176467e-01]

# RandomizedSearchCV поддерживает любые
# распределения, где есть метод rvs

```

Теперь напишем уже сам код:

```

from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVC

pipe_svc = make_pipeline(StandardScaler(),
                        SVC(random_state=1))

param_range = scipy.stats.loguniform(0.0001, 1000.0)
param_grid = [
    {'svc__C': param_range,
     'svc__kernel': ['linear']},
    {'svc__C': param_range,
     'svc__gamma': param_range,
     'svc__kernel': ['rbf']}]

rs = RandomizedSearchCV(estimator=pipe_svc,
                        param_distributions=param_grid,
                        scoring='accuracy', refit=True,
                        n_iter=20, cv=10, n_jobs=-1,
                        random_state=1)

rs = rs.fit(X_train, y_train)
print(rs.best_score_)
print(rs.best_params_)

# Вывод:
# 0.9780676328502416
# {'svc__C': 0.05971247755848464, 'svc__kernel': 'linear'}

```

А теперь рассмотрим более эффективный поиск с последовательным разделением пополам или **hyperparameter search with successive halving**. В библиотеке уже есть вариант successive halving – HalvingRandomSearchCV. Имея большой сет конфигураций, алгоритм последовательно выкидывает плохие конфигурации пока не останется ровна одна. Выпишем основные шаги:

1. Сгенерируем большой набор возможных конфигураций с помощью случайной выборки.
2. Тренируем модель с ограниченными ресурсами, например, на маленьком подмножестве train датасета.
3. Выкинем 50% самых плохих конфигураций.
4. Перейдём к пункту 2 с **увеличенными** ресурсами.

Алгоритм продолжается пока не останется ровно одна конфигурация. Заметим, что также есть halving implementation для grid search – HalvingGridSearchCV, где в 1 пункте используются все данные нам наборы. Теперь напишем код:

```

# т.к. в версии 1.0.2 это ещё экспериментальная
# версия, то нужно явно проставить разрешение:
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import HalvingRandomSearchCV

# resource='n_samples': мы рассматриваем training set size
# как ресурс, который мы меняем между раундами
# factor=1.5: такое кол-во останется в каждом раунде
# т.е. пройдут дальше 100%/1.5 ~ 66%
# n_candidates='exhaust': вместо кол-ва итераций
# параметр показывает, что мы выберем столько
# итераций, что в последнем раунде использованы
# все ресурсы
hs = HalvingRandomSearchCV(pipe_svc,
                            param_distributions=param_grid,
                            n_candidates='exhaust',
                            resource='n_samples',
                            factor=1.5,
                            random_state=1,
                            n_jobs=-1)

```

```

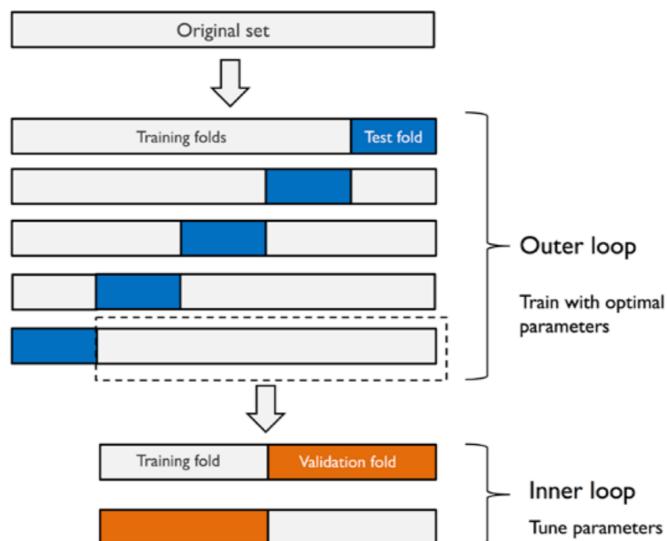
hs = hs.fit(X_train, y_train)
print(hs.best_score_)
print(hs.best_params_)
# 0.9617647058823529
# {'svc__C': 4.934834261073341, 'svc__kernel': 'linear'}

clf = hs.best_estimator_
print(f'Test accuracy: {hs.score(X_test, y_test):.3f}')
# Test accuracy: 0.982

```

Если мы сравним результативность с двумя предыдущими примерами, то увидим, что этот алгоритм показывает себя чуть чуть лучше на 1%. Другая популярная библиотека для таких оптимизаций – это **hyperopt**, которая реализует ещё парочку методов: randomized search и Tree-structured Parzen Estimators (TPE). **TPE** – это Bayesian optimization метод, основанный на вероятностной модели, которая непрерывно обновляется на предыдущих вычислениях гиперпараметров и их точностях, а не как мы – мы считаем, что эти события независимые. Также есть библиотека **hyperopt-sklearn**, которая даёт больше удобств в использовании.

Последний параграф в этой части – выбор алгоритма с **nested cross-validation**. Использование k-fold cross-validation в комбинации с grid search (или randomized) – хороший способ варьировать параметры – это прошлые части. Если мы хотим выбрать среди нескольких ml алгоритмов, то хороший подход – это **nested cross-validation**. Истинная ошибка модели практически не смешена по отношению к тестовым данным, если мы использовали этот алгоритм (в интернете даже можно найти статью). В nested cross-validation мы имеем внешний цикл k-fold cross-validation, чтобы разбить данные на тренировочные и тестовые, а также внутренний цикл, который используется, чтобы выбрать модель с помощью k-fold cross-validation на тренировочных данных. После выбора модели итоговая оценка делается на тестовых данных. Следующая картинка поясняет **5 × 2 cross-validation**:



Этот способ крайне полезен на огромных датасетах, где важна эффективность вычислений. Напишем код nested cross-validation с помощью grid search:

```

from sklearn.model_selection import GridSearchCV
param_range = [float(10**i) for i in range(-4, 4)]
param_grid = [ ... ]
gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=2)
scores = cross_val_score(gs, X_train, y_train,
                        scoring='accuracy', cv=5)
print(f'CV accuracy: {np.mean(scores):.3f} ± {np.std(scores):.3f}')

```

Попробуем сделать тоже самое только для decision tree:

```

from sklearn.tree import DecisionTreeClassifier
gs = GridSearchCV(
    estimator=DecisionTreeClassifier(random_state=0),
    param_grid=[{'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
    scoring='accuracy', cv=2
)
scores = cross_val_score(gs, X_train,

```

```

scoring='accuracy', cv=5)
print(f'CV accuracy: {np.mean(scores):.3f} ;'
      f' +/- {np.std(scores):.3f}')
# CV accuracy: 0.934 +/- 0.016

```

6.5 Performance evaluation metrics

В предыдущих главах мы использовали для оценки prediction accuracy, с помощью которой мы количественно оценивали производительность. Но есть множество других: **precision**, **recall**, **F1 score**, Matthews correlation coefficient (**MCC**).

Для начала рассмотрим **confusion matrix**, которая оценивает производительность. Это просто квадратная матрица, которая содержит следующие данные: true positive (TP), true negative (TN), false positive (FP) и false negative (FN). См. картинку:

| | | Predicted class | |
|--------------|---|----------------------|----------------------|
| | | P | N |
| Actual class | P | True positives (TP) | False negatives (FN) |
| | N | False positives (FP) | True negatives (TN) |

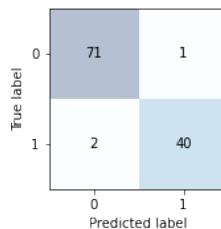
Безусловно эти значения можно вычислить руками, но есть удобная функция:

```

from sklearn.metrics import confusion_matrix
pipe_svc.fit(X_train, y_train)
y_pred = pipe_svc.predict(X_test)
confmat = confusion_matrix(y_true=y_test,
                           y_pred=y_pred)
print(confmat)
# [[71  1]
# [ 2 40]]

# теперь выведем это через matshow:
fig, ax = plt.subplots(figsize=(2.5, 2.5))
# plt.cm.Blues - colormap
ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confmat.shape[0]):
    for j in range(confmat.shape[1]):
        ax.text(x=j, y=i, s=confmat[i, j],
                va='center', ha='center')
# чтобы надпись не налезала
# на цифры 0 и 1
ax.xaxis.set_ticks_position('bottom')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.show()

```



Дальше мы будем использовать эту информацию, чтобы считать различные error metrics. Теперь рассмотрим оптимизацию precision и recall. И **error (ERR)**, и **accuracy (ACC)** дают общую информацию о кол-ве ошибок:

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

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

Метрики, которые особенно полезны для задач с несбалансированными классами – это **true positive rate (TPR)** и **false positive rate (FPR)**:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

Рассмотрим опухоли, наша задача определить malignant tumors, также важно уменьшить кол-во benign tumors, которые классифицированы как malignant (FP). По сравнению с FPR, TPR даёт полезную информацию о кол-ве правильно определённых malignant tumors. Метрики **precision (PRE)** и **recall (REC)** относятся к TP и TN, на самом деле REC – синоним TPR:

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

Другими словами он показывает какое кол-во Р исходов мы отнесли к ТР. А вот precision определяет отношение ТР исходов к исходам, которые мы вообще определили как Р:

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

Вернёмся к нашему примеру. Оптимизация recall – минимизация шансов не определить malignant tumor, с другой стороны это влияет на рост FP. Тогда оптимизация recall помогает как раз поддерживать корректность предсказаний, но тогда растёт число FN. Чтобы балансировать между этими + и – в оптимизации PRE и REC, используют их гармоническое среднее – **F1 score**:

$$F1 = 2 \cdot \frac{PRE \times REC}{PRE + REC}$$

Наконец, метрика, которая обобщает confusion matrix – это **MCC**, которая очень популярна в рамках исследований в биологии:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

По сравнению с PRE, REC и F1, MCC выдаёт отрезок $[-1; 1]$, а также учитывает все элементы confmat, например, F1 не учитывает TN. Хотя MCC сложнее интерпретировать, он считается лучше, чем F1. Напишем код:

```
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score, f1_score
from sklearn.metrics import matthews_corrcoef

pre_val = precision_score(y_true=y_test, y_pred=y_pred)
print(f'Precision: {pre_val:.3f}') # Precision: 0.976

rec_val = recall_score(y_true=y_test, y_pred=y_pred)
print(f'Recall: {rec_val:.3f}') # Recall: 0.952

f1_val = f1_score(y_true=y_test, y_pred=y_pred)
print(f'F1: {f1_val:.3f}') # F1: 0.964

mcc_val = matthews_corrcoef(y_true=y_test, y_pred=y_pred)
print(f'MCC: {mcc_val:.3f}') # MCC: 0.943
```

Мы можем использовать другие метрики в GridSearchCV, а не accuracy. Важно помнить, что positive class в scikit-learn – класс помеченный цифрой 1. Если мы хотим другую positive label, то это можно сделать через make_scorer. См. код:

```
from sklearn.metrics import make_scorer
c_gamma_range = [0.01, 0.1, 1.0, 10.0]
param_grid = [ тоже самое только с c_gamma_range ]
scorer = make_scorer(f1_score, pos_label=0)
gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring=scorer, cv=10)

gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
# Выход:
# 0.9861994953378878
# {'svc__C': 10.0, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}
```

В этом параграфе мы будем строить график **receiver operating characteristic (ROC)**. Графики ROC полезны, чтобы выбрать модель на основании FPR и TPR, которые считаются путём сдвига decision threshold. Диагональ ROC-графика может быть интерпретирована как random guessing, а модели классификации, которые находятся ниже диагонали, считаются хуже, чем random guessing.

Идеальный классификатор должен быть в левом верхнем углу графика с $TPR = 1$ и $FPR = 0$, потом можно посчитать площадь под графиком – характеристика модели – **ROC area under the curve (ROC AUC)**. По аналогии есть precision-recall curves. Посмотрим на код ниже, на нём уменьшено число фич и число fold, чтобы ROC-кривая была интереснее:

```
from sklearn.metrics import roc_curve, auc
from numpy import interp

pipe_lr = make_pipeline(
    StandardScaler(),
    PCA(n_components=2),
    LogisticRegression(penalty='l2', random_state=1,
                        solver='lbfgs', C=100.0))

# выбираем случайные 2 фичи
# попробуйте подать вместо этого просто X_train
# результат будет заметно лучше
X_train2 = X_train[:, [4, 14]]
cv = list(StratifiedKFold(n_splits=3).split(X_train, y_train))

fig = plt.figure(figsize=(7, 5))
mean_tpr = 0.0
mean_fpr = np.linspace(0, 1, 100)

for i, (train, test) in enumerate(cv):
    # вероятности принадлежности
    # к каждому из 2x классов
    probas = pipe_lr.fit(
        X_train2[train],
        y_train[train]
    ).predict_proba(X_train2[test])

    # на основании разных thresholds
    # вычисляются разные fpr и tpr
    fpr, tpr, thresholds = roc_curve(y_train[test],
                                      probas[:, 1],
                                      pos_label=1)
    # слева на графике очень большой threshold,
    # справа он очень маленький

    # это интерполяция, у нас дан дискретный набор
    # точек (fpr[i], tpr[i]), после мы вычисляем значения
    # в точках mean_fpr[i]
    mean_tpr += interp(mean_fpr, fpr, tpr)

    # вычисление площади под графиком
    roc_auc = auc(fpr, tpr)

    plt.plot(fpr, tpr,
              label=f'ROC fold {i+1} (area = {roc_auc:.2f})')

# это кривая случайной выборки
plt.plot([0, 1], [0, 1],
          linestyle='--',
          color=(0.6, 0.6, 0.6),
          label='Random guessing (area=0.5)')

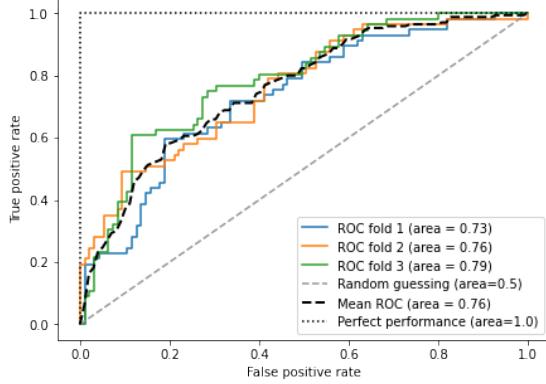
# считаем среднюю ROC кривую
mean_tpr /= len(cv)
mean_tpr[0] = 0.0
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)

plt.plot(mean_fpr, mean_tpr, 'k--',
          label=f'Mean ROC (area = {mean_auc:.2f})', lw=2)

# идеальная кривая
plt.plot([0, 0, 1], [0, 1, 1],
          linestyle=':', color='black',
          label='Perfect performance (area=1.0)')

plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.legend(loc='lower right')
plt.show()
```

```
# можно сделать проще, если нам нужно просто число:
from sklearn.metrics import roc_auc_score
pipe_lr.fit(X_train[:, [4, 14]], y_train)
probas = pipe_lr.predict_proba(X_test[:, [4, 14]])[:, 1]
print(f'ROC score: {roc_auc_score(y_true=y_test, y_score=probas):.2f}')
# Выход:
# ROC score: 0.74
```



Небольшое пояснение: выбираются различные threshold, слева очень большие, справа очень маленькие, для этих threshold считаются fpr и tpr, на основании этих точек и строится кривая. Площадь под графиком даёт численную оценку качества нашей модели. С помощью механизма линейной интерполяции строится средняя кривая. Этот метод оценки также хорошо работает для несбалансированных данных. Теперь немного поговорим про оценку для **multiclass classification**. Метрики, которые мы обсудили, специфичны для бинарной классификации. Однако scikit-learn реализует macro и micro averaging методы для расширения на несколько классов через OvA. Например, micro-average от precision в системе из k классов считают так:

$$PRE_{micro} = \frac{TP_1 + \dots + TP_k}{TP_1 + \dots + TP_k + FP_1 + \dots + FP_k}$$

А вот macro-average считается так:

$$PRE_{macro} = \frac{PRE_1 + \dots + PRE_k}{k}$$

Micro-averaging полезно, если мы хотим взвесить каждое предсказание одинаково, когда macro-averaging считает итоговую оценку в сторону самого частого класса. Когда мы считаем через scikit-learn, то по умолчанию используется нормализованный или взвешанный вариант macro-average, считается просто домножением каждого PRE_i на кол-во элементов в классе. Это очень полезно для несбалансированных классов. Напишем немного кода:

```
# greater_is_better: для loss function это False
# а вот в нашем случае, например, это True

# average по умолчанию взвешенное
pre_scoring = make_scoring(score_func=precision_score,
                           pos_label=1,
                           greater_is_better=True,
                           average='micro')
```

Рассмотрим работу с **class imbalance**. Есть много примеров, где такое происходит: spam filtering, fraud detection или screening for diseases. Например, если в датасете с раком 90% были бы здоровые люди, то можно сделать модель с точностью 90%, просто предсказывая всё время 0, что очень плохо. Сначала сделаем из нашего датасета несбалансированный (было 357B на 212M):

```
X_imb = np.vstack((X[y == 0], X[y == 1][:40]))
y_imb = np.hstack((y[y == 0], y[y == 1][:40]))
```

```
# плохой вариант:
y_pred = np.zeros(y_imb.shape[0])
print(f'Wrong acc: {np.mean(y_pred == y_imb) * 100}')
# Wrong acc: 89.92443324937027
```

Работая с такими данными нужно взять другие метрики – ROC curve, recall, precision. Например, если наш приоритет выявить всех больных для доп. обследования, то наш выбор – это recall; а вот если наша система не уверена, что сообщение – это спам, то лучше ничего не трогать – наш

выбор это precision. Большинство алгоритмов считают reward или loss, которые считаются суммой по всем примерами, следовательно decision rule будет смещаться к более частому классу. Один из вариантов решения – это делать более сильное наказание за ошибку на minority классе. Через scikit-learn это сделать очень просто – class_weight='balanced'. Другие варианты – уменьшение выборки большего класса, увеличение выборки меньшего класса, а также генерация синтетических данных. В библиотеке есть простая функция – resample, которая увеличивает выборку меньшего класса, выбирая данные из датасета with replacement. Напишем код:

```
from sklearn.utils import resample
print('Number of class 1 examples before:',
      X_imb[y_imb == 1].shape[0]) # 40
X_upsampled, y_upsampled = resample(
    X_imb[y_imb == 1],
    y_imb[y_imb == 1],
    replace=True,
    n_samples=X_imb[y_imb == 0].shape[0],
    random_state=123)
print('Number of class 1 examples after:',
      X_upsampled.shape[0]) # 357

X_bal = np.vstack((X[y == 0], X_upsampled))
y_bal = np.hstack((y[y == 0], y_upsampled))

y_pred = np.zeros(y_bal.shape[0])
print(np.mean(y_pred == y_bal) * 100.0) # 50.0

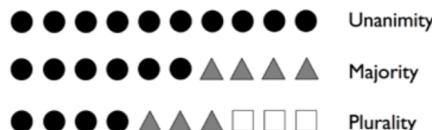
# аналогично можно сделать downsample
# у главного класса, для этого
# будем использовать тоже функцию resample
# только поменяем всегда 0 на 1, 1 на 0
X_exmp, y_exmp = resample(X_imb[y_imb == 0],
                           y_imb[y_imb == 0],
                           n_samples=X_imb[y_imb == 1].shape[0],
                           random_state=1)
print(f'Len={len(X_exmp)}') # Len=40
```

Другая техника – генерация новых синтетических примеров, самый популярный алгоритм – это **Synthetic Minority Over-sampling Technique (SMOTE)**. В следующей главе мы посмотрим на ensemble methods, которые позволяют комбинировать несколько моделей и алгоритмов, чтобы повысить качество предсказаний.

7 Ensemble learning

7.1 Learning with ensembles

Суть **ensemble methods** в соединении нескольких моделей с целью улучшения производительности по сравнению с каждым по отдельности. В этой части мы посмотрим на самую популярную часть – методы, которые используют **majority voting**. Идея в том, что ответ тот, который дали более 50% всех моделей, но строго говоря это применимо только для бинарной классификации, но можно расширить, тогда это называется **plurality voting**. Есть другие названия соответственно: **absolute** и **relative**. Пример на картинке:



Используя train dataset, мы тренируем m разных моделей C_1, \dots, C_m . В зависимости от техники мы можем построить ensemble из разных алгоритмов (svm + lr + tree), с другой стороны мы можем использовать один и тот же алгоритм на разных подмножествах данных. Кстати, у нас уже был пример ensemble – random forest. Чтобы предсказывать в plurality voting с помощью majority используем mode – в статистике – это событие с самой частой встречаемостью: $\hat{y} = \text{mode}\{C_1(x), C_2(x), \dots, C_m(x)\}$. Например, в бинарной классификации, где class1 = -1 и class2 = +1 можно записать majority vote так:

$$C(x) = \text{sign} \left[\sum_j^m C_j(x) \right] = \begin{cases} 1 & \text{if } \sum_j C_j(x) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Чтобы показать, что ensemble работает лучше, чем отдельный алгоритм применим комбинаторику. Некоторые соглашения для примера: пусть каждая из n моделей имеет одинаковую частоту ошибок ϵ (error rate), а также модели независимы. Вероятность ошибки ensemble:

$$P(y \geq k) = \sum_k^n \binom{n}{k} \epsilon^k (1 - \epsilon)^{n-k} = \epsilon_{\text{ensemble}}$$

Другими словами мы считаем вероятность, что ensemble ошибся. Более конкретный пример, где $n = 11$, $\epsilon = 0.25$:

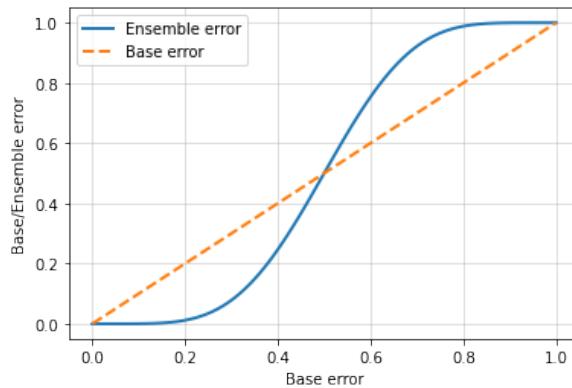
$$P(y \geq k) = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1 - 0.25)^{11-k} = 0.034$$

Пояснение: у нас 11 моделей, если ошиблось 6 или более, то ошибся и ensemble, поэтому сумма с 6. Чтобы сравнить этот ensemble с base classifier по разным error rates, напишем код probability mass функции и построим график:

```
from scipy.special import comb
import math
def ensemble_error(n_classifier, error):
    k_start = int(math.ceil(n_classifier / 2.0))
    probs = [comb(n_classifier, k) *
              error**k *
              (1 - error)**(n_classifier - k)
              for k in range(k_start, n_classifier + 1)]
    return sum(probs)

print(f'Ensemble error=\n\tf'{ensemble_error(n_classifier=11, error=0.25):.3f}\n')
# Выход
# Ensemble error=0.034

import numpy as np
import matplotlib.pyplot as plt
error_range = np.arange(0.0, 1.01, 0.01)
ens_errors = [ensemble_error(n_classifier=11, error=error)
              for error in error_range]
plt.plot(error_range, ens_errors,
         label='Ensemble error',
         linewidth=2)
plt.plot(error_range, error_range,
         linestyle='--', label='Base error',
         linewidth=2)
plt.xlabel('Base error')
plt.ylabel('Base/Ensemble error')
plt.legend(loc='upper left')
plt.grid(alpha=0.5)
plt.show()
```



Как мы видим ensemble лучше одной модели пока наш алгоритм лучше чем random guessing.

7.2 Combining classifiers via majority vote

Реализуем простой majority vote classifier. Алгоритм, который мы будем писать, разрешит нам комбинировать разные алгоритмы классификации с разными весами. Это позволит убрать слабости

каждого алгоритма. Строго говоря weighted majority vote – это:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j \chi_A(C_j(x) = i)$$

Здесь w_j – вес класса C_j , A – множество уникальных меток, χ_A – характеристическая (индикаторная) функция, которая возвращает 1, если выполнено условие. Маленький пример, у нас три модели с весами 0.2, 0.2 и 0.6, первые две дали 0, а последняя дала 1, тогда:

$$\hat{y} = \arg \max_i [0.2 \times i_0 + 0.2 \times i_0, 0.6 \times i_1] = 1$$

Напишем пару строчек кода:

```
import numpy as np
# предполагается, что метки
# начинаются с 0:
x = [0, 0, 1]
# кол-во просто умножится
# на весовой коэффи.
print(np.argmax(np.bincount(x,
                           weights=[0.2, 0.2, 0.6])))
# Выход: 1
```

Вспомним, что некоторые классы возвращают вероятности (например, lr) через predict_proba, тогда можно записать так:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j p_{ij}$$

Здесь p_{ij} – вероятность j – ой модели по i – my классу. Приведём пример:

$$C_1(x) \rightarrow [0.9, 0.1], C_2(x) \rightarrow [0.8, 0.2], C_3(x) \rightarrow [0.4, 0.6] \Rightarrow$$

$$p(i_0|x) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1|x) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42$$

Опять напишем пару строк кода:

```
ex = np.array([[0.9, 0.1],
               [0.8, 0.2],
               [0.4, 0.6]])
p = np.average(ex, axis=0, # no строкам
               weights=[0.2, 0.2, 0.6])
# Работаем так:
# avg = sum(a * weights) / sum(weights)
print(f'Probas: {p}') # Probas: [0.58 0.42]
print(f'Class: {np.argmax(p)}') # Class: 0
```

Теперь, собрав всё вместе, напишем MajorityVoteClassifier:

```
# get_params, set_params
from sklearn.base import BaseEstimator
# score
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator

# мы делаем наследование, чтобы получить базовый функционал
# за бесплатно: get_params, set_params методы, установка и
# возвращение параметров классификатора, score метод
class MajorityVoteClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, classifiers, vote='classlabel', weights=None):
        self.classifiers = classifiers
        # даёт названия:
        # print(_name_estimators(['a', 'a', 'b']))
        # [(‘a-1’, ‘a’), (‘a-2’, ‘a’), (‘b’, ‘b’)]
        self.named_classifiers = {
            key: value for key,
            value in _name_estimators(classifiers)}
    self.vote = vote
    self.weights = weights
```

```

def fit(self, X, y):
    if self.vote not in ('probability', 'classlabel'):
        raise ValueError(f"vote must be 'probability', "
                         f"or 'classlabel'"
                         f"; got (vote={self.vote})")

    if self.weights and \
        len(self.weights) != len(self.classifiers):
        raise ValueError(f'Number of classifiers and '
                         f'weights must be equal',
                         f'; got {len(self.weights)} weights, '
                         f'{len(self.classifiers)} classifiers')

    # нам он нужен, чтобы убедиться, что
    # метки идут с 0 – это важно для np.argmax
    self.lablenc_ = LabelEncoder()
    self.lablenc_.fit(y)
    self.classes_ = self.lablenc_.classes_
    self.classifiers_ = []

    for clf in self.classifiers:
        fitted_clf = clone(clf).fit(X,
                                     self.lablenc_.transform(y))
        self.classifiers_.append(fitted_clf)
    return self

# дальше будет написан метод predict_proba, который
# возвращает averaged probabilities, что очень полезно
# при подсчёте ROC AUC

def predict(self, X):
    if self.vote == 'probability':
        # no строкам
        maj_vote = np.argmax(self.predict_proba(X), axis=1)
    else:
        predictions = np.asarray([
            clf.predict(X) for clf in self.classifiers_
        ]).T
        # no строкам:
        maj_vote = np.apply_along_axis(
            lambda x: np.argmax(
                np.bincount(x, weights=self.weights)
            ),
            axis=1, arr=predictions
        )

    maj_vote = self.lablenc_.inverse_transform(maj_vote)
    return maj_vote

def predict_proba(self, X):
    probas = np.asarray([clf.predict_proba(X)
                        for clf in self.classifiers_])
    avg_proba = np.average(probas, axis=0,
                           weights=self.weights)
    return avg_proba

# If True, will return the parameters for this
# estimator and contained subobjects that are estimators.
def get_params(self, deep=True):
    if not deep:
        return super().get_params(deep=False)
    else:
        out = self.named_classifiers.copy()
        for name, step in self.named_classifiers.items():
            for key, value in step.get_params(
                deep=True).items():
                out[f'{name}__{key}'] = value
        return out

# Мы переопределили метод get_params, чтобы
# можно было доставать параметры каждого отдельного
# классификатора. Это нам пригодится дальше для
# grid search.

```

Эта реализация полезна для понимания, но в библиотеке она уже есть (даже более сложная) – sklearn.ensemble.VotingClassifier. Теперь научимся делать предсказания, из Iris dataset мы выберем ровно 2 фичи, чтобы сделать задачу классификации более сложной для целей иллюстрации. Наш

класс умеет работать с multiclass problems, но мы сделаем бинарную классификацию. Пару слов скажем о вероятностях классов для decision trees. ROC AUC score использует метод predict_proba, конечно если он есть. В decision trees вероятности считаются из frequency vector, который создаётся в каждой вершине во время тренировки. Вектор собирает частоты каждого класса, которые считаются из разбиения вершины, далее их нормализуют так, чтобы сумма была равна 1. Аналогично это работает и для KNN. Хотя всё это похоже на lr, но нужно понимать, что вероятности берутся не от probability mass functions. А вот и код:

```

from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
iris = datasets.load_iris()
X, y = iris.data[50:, [1, 2]], iris.target[50:]
le = LabelEncoder()
y = le.fit_transform(y)

X_train, X_test, y_train, y_test =\
    train_test_split(X, y, test_size=0.5,
                     random_state=1, stratify=y)

# дальше мы соединим lr, decision tree, KNN
# точность измерим с помощью 10-fold cross val.
# дальше ужес соединим в ensemble

from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
import numpy as np

clf1 = LogisticRegression(penalty='l2',
                          C=0.001,
                          solver='lbfgs',
                          random_state=1)
clf2 = DecisionTreeClassifier(max_depth=1,
                              criterion='entropy',
                              random_state=0)
clf3 = KNeighborsClassifier(n_neighbors=1,
                            p=2, metric='minkowski')

pipe1 = Pipeline([
    ['sc', StandardScaler()],
    ['clf', clf1]])
pipe3 = Pipeline([
    ['sc', StandardScaler()],
    ['clf', clf3]])

clf_labels = ['Logistic regression', 'Decision tree', 'KNN']
print('10-fold cross validation:\n')
for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
    scores = cross_val_score(estimator=clf,
                             X=X_train, y=y_train,
                             cv=10, scoring='roc_auc')
    print(f'ROC AUC: {scores.mean():.2f} '
          f'(+/- {scores.std():.2f}) [{label}]')

mv_clf = MajorityVoteClassifier(
    classifiers=[pipe1, clf2, pipe3])
clf_labels += ['Majority voting']
all_clf = [pipe1, clf2, pipe3, mv_clf]
for clf, label in zip(all_clf, clf_labels):
    scores = cross_val_score(estimator=clf,
                             X=X_train, y=y_train,
                             cv=10, scoring='roc_auc')
    print(f'ROC AUC: {scores.mean():.2f} '
          f'(+/- {scores.std():.2f}) [{label}]')

# Вывод:
# ROC AUC: 0.92 (+/- 0.15) [Logistic regression]
# ROC AUC: 0.87 (+/- 0.18) [Decision tree]
# ROC AUC: 0.85 (+/- 0.13) [KNN]
# ROC AUC: 0.98 (+/- 0.05) [Majority voting]

```

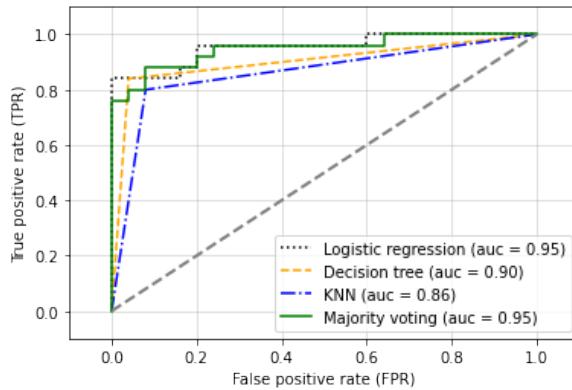
А теперь напишем код, в котором построим ROC кривые:

```
from sklearn.metrics import roc_curve
```

```

from sklearn.metrics import auc
colors = ['black', 'orange', 'blue', 'green']
linestyles = [':', '--', '-.', '-']
for clf, label, clr, ls \
    in zip(all_clf, clf_labels, colors, linestyles):
    y_pred = clf.fit(X_train, y_train).predict_proba(X_test)[:, 1]
    fpr, tpr, thresholds = roc_curve(y_true=y_test,
                                      y_score=y_pred)
    roc_auc = auc(x=fpr, y=tpr)
    plt.plot(fpr, tpr, color=clr,
              linestyle=ls,
              label=f'{label} (auc = {roc_auc:.2f})')
plt.legend(loc='lower right')
plt.plot([0, 1], [0, 1],
          linestyle='--',
          color='gray',
          linewidth=2)
plt.xlim([-0.1, 1.1])
plt.ylim([-0.1, 1.1])
plt.grid(alpha=0.5)
plt.xlabel('False positive rate (FPR)')
plt.ylabel('True positive rate (TPR)')
plt.show()

```



Мы видим, что lr показывает такую же точность, но это связано с high variance, т.е. в нашем случае с тем, как мы разбиваем датасет. Хотя делать стандартизацию здесь необязательно (уже есть в pipeline), мы её всё таки сделаем для целей визуализации (мы хотим, чтобы дерево имело такой же scale). Посмотрим на decision bounds:

```

sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
from itertools import product
x_min = X_train_std[:, 0].min() - 1
x_max = X_train_std[:, 0].max() + 1
y_min = X_train_std[:, 1].min() - 1
y_max = X_train_std[:, 1].max() + 1

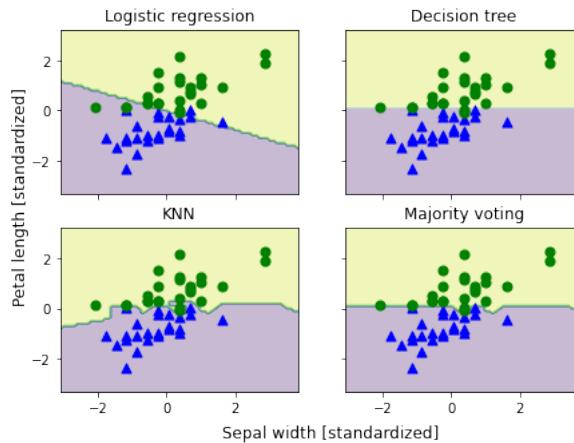
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                      np.arange(y_min, y_max, 0.1))
# про параметры share:
# мы контролируем для каждого подграфика его оси
# у каждого они свои или продолжаются как-то
f, axarr = plt.subplots(nrows=2, ncols=2,
                       sharex='col',
                       sharey='row',
                       figsize=(7, 5))
# product генерирует нам пары
# (0, 0), (0, 1), (1, 0), (1, 1)
for idx, clf, tt in zip(product([0, 1], [0, 1]),
                        all_clf, clf_labels):
    clf.fit(X_train_std, y_train)
    # np.c_[np.array([1, 2, 3]),
    #       np.array([4, 5, 6])]
    # Выход:
    # array([[1, 4],
    #        [2, 5],
    #        [3, 6]])
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)

```

```

axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
                               X_train_std[y_train==0, 1],
                               c='blue', marker='^', s=50)
axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
                               X_train_std[y_train==1, 1],
                               c='green', marker='o', s=50)
axarr[idx[0], idx[1]].set_title(tt)
plt.text(-3.5, -5.,
         s='Sepal width [standardized]',
         ha='center', va='center', fontsize=12)
plt.text(-12.5, 4.5,
         s='Petal length [standardized]',
         ha='center', va='center',
         fontsize=12, rotation=90)
plt.show()

```



Мы видим смешение границ у ensemble, в итоге граница и всё охватывает, и пытается быть максимально простой. Перед тем как мы настроим отдельные параметры каждого классификатора, вызовем метод `get_params`, чтобы понять как обращаться к индивидуальным параметрам внутри `GridSearchCV`. Я приведу одну строку кода и совсем немного вывода, он слишком большой, чтобы сюда его вставлять, попробуйте сами:

```

mv_clf.get_params()

# Вывод:
# {'decisiontreeclassifier': DecisionTreeClassifier(...),
# ...
# 'pipeline-1': Pipeline(steps=[('sc', StandardScaler()),
#                                ('clf', LogisticRegression(C=0.001, random_state=1))]),
# 'pipeline-1__clf': LogisticRegression(C=0.001, random_state=1),
# ...
# 'pipeline-2__verbose': False}

```

Теперь, основываясь на этом выводе, мы знаем как получить доступ к отдельным параметрам, напишем кода для демонстрации:

```

from sklearn.model_selection import GridSearchCV
params = {'decisiontreeclassifier__max_depth': [1, 2],
          'pipeline-1__clf__C': [0.001, 0.1, 100.0]}
grid = GridSearchCV(estimator=mv_clf,
                     param_grid=params,
                     cv=10,
                     scoring='roc_auc')
grid.fit(X_train, y_train)

# Теперь выведем разные комбинации параметров
# и их результатов через average ROC AUC scores
# via 10-fold cross-validation

for r, _ in enumerate(grid.cv_results_['mean_test_score']):
    mean_score = grid.cv_results_['mean_test_score'][r]
    std_dev = grid.cv_results_['std_test_score'][r]
    params = grid.cv_results_['params'][r]
    print(f'{mean_score:.3f} +/- {std_dev:.2f} {params}')
print(f'Best parameters: {grid.best_params_}')
print(f'ROC AUC: {grid.best_score_:.2f}')

# Вывод:

```

```

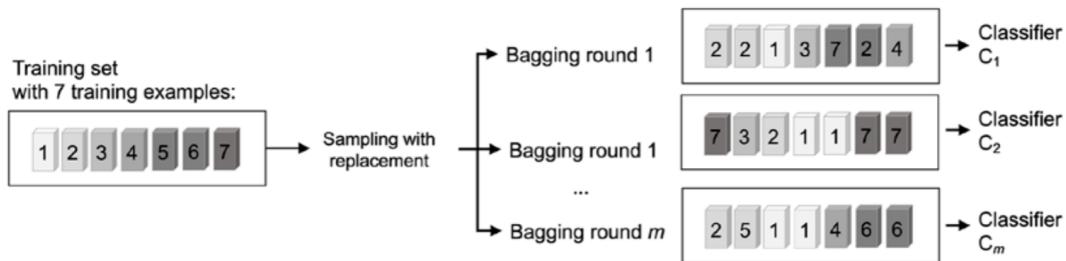
# 0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 0.001}
# 0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 0.1}
# 0.967 +/- 0.10 {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 100.0}
# 0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C': 0.001}
# 0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C': 0.1}
# 0.967 +/- 0.10 {'decisiontreeclassifier__max_depth': 2, 'pipeline-1__clf__C': 100.0}
# Best parameters: {'decisiontreeclassifier__max_depth': 1, 'pipeline-1__clf__C': 0.001}
# ROC AUC: 0.98

```

Далее мы рассмотрим альтернативный вариант ensemble _ learning – **bagging**. Есть ещё одна очень интересная техника – **stacking**. Этот алгоритм можно рассмотреть как двухуровневый ensemble, где на первом уровне есть разные классификаторы, а на втором уровне другой классификатор (обычно lr), который учится давать итоговое предсказание.

7.3 Bagging – ensemble from bootstrap samples

Техника очень похожа на ту, которую мы уже прошли, суть в том, что мы делаем bootstrap samples (random samples with replacement) из исходного датасета, поэтому этот алгоритм часто называют **bootstrap aggregating**. В качестве классификатора обычно выбирают неограниченное decision tree. См. картинку:



На самом деле, random forest – это особый случай bagging, где мы использовали random feature subsets. Bagging улучшает качество нестабильных моделей, а также уменьшает степень переобучения. Теперь применим bagging для классификации wine dataset, мы рассмотрим только классы 2, 3 и только 2 фичи:

```

import pandas as pd
df_wine = pd.read_csv( см. п. 4.3 )
df_wine.columns = [ см. п. 4.3 ]
df_wine = df_wine[df_wine['Class label'] != 1]
y = df_wine['Class label'].values
X = df_wine[['Alcohol',
              'OD280/OD315 of diluted wines']].values

from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
le = LabelEncoder()
y = le.fit_transform(y)
X_train, X_test, y_train, y_test = процент - 0.2

from sklearn.ensemble import BaggingClassifier
tree = DecisionTreeClassifier(criterion='entropy',
                               random_state=1,
                               # неограниченное
                               max_depth=None)

# max_samples, max_features – процент
# примеров и фич, которые нужно взять
# каждому base estimator
bag = BaggingClassifier(base_estimator=tree,
                        n_estimators=500,
                        max_samples=1.0,
                        max_features=1.0,
                        # примеры with replacement
                        bootstrap=True,
                        # фичи without replacement
                        bootstrap_features=False,
                        n_jobs=1,
                        random_state=1)

from sklearn.metrics import accuracy_score
tree = tree.fit(X_train, y_train)

```

```

y_train_pred = tree.predict(X_train)
y_test_pred = tree.predict(X_test)
tree_train = accuracy_score(y_train, y_train_pred)
tree_test = accuracy_score(y_test, y_test_pred)
print(f'Decision tree train/test accuracies',
      f'{tree_train:.3f}/{tree_test:.3f}')
# Decision tree train/test accuracies 1.000/0.833
# Видно, что одно дерево явно переобучается.

bag = bag.fit(X_train, y_train)
# ...
# также самое, что и выше, только bag, а не tree
print(f'Bagging train/test accuracies',
      f'{bag_train:.3f}/{bag_test:.3f}')
# Bagging train/test accuracies 1.000/0.917

```

Теперь посмотрим на границу между классами:

```

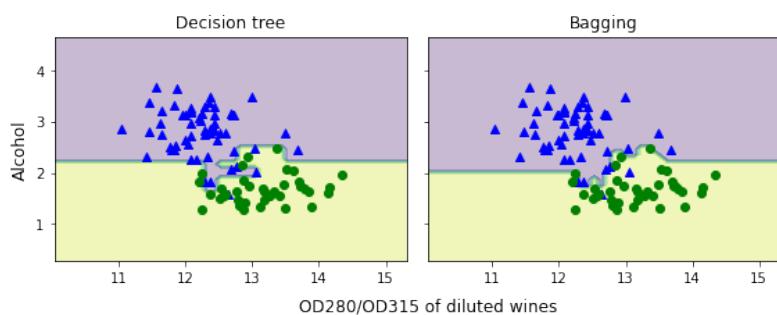
x_min = X_train[:, 0].min() - 1
# ...
y_max = X_train[:, 1].max() + 1

xx, yy = np.meshgrid( ... )

f, axarr = plt.subplots(nrows=1, ncols=2,
                       sharex='col', sharey='row',
                       figsize=(8, 3))

for idx, clf, tt in zip([0, 1], [tree, bag],
                       ['Decision tree', 'Bagging']):
    clf.fit(X_train, y_train)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    axarr[idx].contourf(xx, yy, Z, alpha=0.3)
    axarr[idx].scatter(X_train[y_train==0, 0],
                       X_train[y_train==0, 1],
                       c='blue', marker='^')
    axarr[idx].scatter(X_train[y_train==1, 0],
                       X_train[y_train==1, 1],
                       c='green', marker='o')
    axarr[idx].set_title(tt)
axarr[0].set_ylabel('Alcohol', fontsize=12)
plt.tight_layout()
# Мы ставим оси в обычное в математике положение
# в точку (0, 0) второго графика.
plt.text(0, -0.2,
         s='OD280/OD315 of diluted wines',
         ha='center', va='center', fontsize=12,
         transform=axarr[1].transAxes)
plt.show()

```



Как мы видим по картинке, кусочная граница от глубокого дерева сменилась гладкой, но и точной границей у bagging. Как мы видим, этот алгоритм очень хорошо помогает при overfitting, однако при high bias он крайне неэффективен, поэтому используются именно неограниченные деревья.

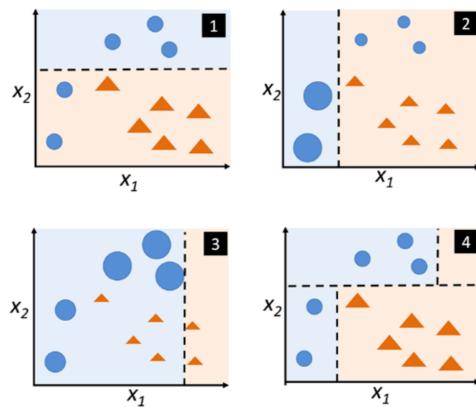
7.4 Adaptive boosting

В этой части мы будем обсуждать **boosting**, а точнее его самую частую реализацию – **Adaptive Boosting (AdaBoost)**. В boosting ensemble состоит из очень простых base classifiers, их называют **weak learners**, которые чуть лучше random guessing, например, tree stump. Ключевая идея в том, чтобы сфокусироваться на примерах, которые сложно классифицировать; weak learners

последовательно учатся на ошибках давать хороший результат. В отличие от bagging тут берутся случайные подмножества **without replacement**. Выделим 4 главные шага:

1. Выберем случайное подмножество примеров d_1 **without replacement** из train dataset D , чтобы обучить weak learner C_1 .
2. Таким же образом выберем множество d_2 и добавим 50% примеров, в которых мы ошиблись в C_1 , чтобы тренировать модель C_2 .
3. Найдём примеры d_3 из датасета D, в которых C_1 и C_2 расходятся во мнениях, чтобы обучить модель C_3 .
4. Соединим weak learners C_1 , C_2 и C_3 через majority voting.

По сравнению с bagging, boosting исправляет high bias. В отличие от первоначальной boosting процедуры, AdaBoost использует весь train dataset, где примеры перевзвешиваются на каждой итерации на основании ошибок сделанных раньше. Идея AdaBoost:



Сначала мы обучаем decision stump так, чтобы минимизировать функцию потерь; все примеры имеют одинаковый вес. Далее мы увеличиваем вес ошибочных примеров и понижаем вес правильных примеров. Учим второй learner, повышаем вес ошибочных примера, учим третий learner. В конце соединяя ответ с помощью взвешанного majority vote. Теперь напишем псевдо код. Соглашение: поэлементное умножение – \times , dot-product между векторами – \cdot . Код:

1. Присвоим вектору w uniform weights (одна константа): $\sum_i w_i = 1$.
2. Для j из m boosting rounds сделаем следующие шаги:
 - (a) Обучим weighted weak learner: $C_j = \text{train}(X, y, w)$.
 - (b) Предскажем: $\hat{y} = \text{predict}(C_j, X)$.
 - (c) Посчитаем weighted error rate: $\epsilon = w * (\hat{y} \neq y)$.
 - (d) Посчитаем коэффициент: $\alpha_j = 0.5 \log \frac{1-\epsilon}{\epsilon}$.
 - (e) Обновим веса: $w := w \times \exp(-\alpha_j \times \hat{y} \times y)$.
 - (f) Нормализуем веса так, чтобы сумма была равна 1: $w := w / \sum_i w_i$.
3. Итоговое предсказание: $\hat{y} = (\sum_{j=1}^m (\alpha_j \times \text{predict}(C_j, X)) > 0)$.

Возьмём такой dataset:

| Index | x | y | Weights | $\hat{y}(x \leq 3.0)?$ | Correct? | Updated weights |
|-------|------|----|---------|------------------------|----------|-----------------|
| 1 | 1.0 | 1 | 0.1 | 1 | Yes | 0.072 |
| 2 | 2.0 | 1 | 0.1 | 1 | Yes | 0.072 |
| 3 | 3.0 | 1 | 0.1 | 1 | Yes | 0.072 |
| 4 | 4.0 | -1 | 0.1 | -1 | Yes | 0.072 |
| 5 | 5.0 | -1 | 0.1 | -1 | Yes | 0.072 |
| 6 | 6.0 | -1 | 0.1 | -1 | Yes | 0.072 |
| 7 | 7.0 | 1 | 0.1 | -1 | No | 0.167 |
| 8 | 8.0 | 1 | 0.1 | -1 | No | 0.167 |
| 9 | 9.0 | 1 | 0.1 | -1 | No | 0.167 |
| 10 | 10.0 | -1 | 0.1 | -1 | Yes | 0.072 |

Теперь напишем код для вычисления новых весов:

```

y = np.array([1, 1, 1, -1, -1, -1, 1, 1, 1, -1])
yhat = np.array([1, 1, 1, -1, -1, -1, -1, -1, -1, -1])
correct = (y == yhat)

# просто массив констант
weights = np.full(10, 0.1)
print(weights) # [0.1 ... 0.1]

epsilon = np.mean(~correct)
print(epsilon) # 0.3
# но сумы это classification error

# это значение > 0 из предположения, что
# epsilon < 0.5
alpha_j = 0.5 * np.log((1-epsilon) / epsilon)
print(alpha_j) # 0.423

# теперь обновим вектор w
# в формуле  $y_{\text{hat}} * y$  надо понимать так:
# если предсказания совпали, то это значение > 0
# тем самым мы увеличим вес, в противном случае
# уменьшим:
update_if_correct = 0.1 * np.exp(-alpha_j * 1 * 1)
print(update_if_correct) # 0.065
update_if_wrong = 0.1 * np.exp(-alpha_j * 1 * -1)
print(update_if_wrong) # 0.152

weights = np.where(correct == 1,
                    update_if_correct,
                    update_if_wrong)
print(weights)
# [0.06546537 0.06546537 0.06546537 0.06546537 0.06546537 0.06546537
# 0.15275252 0.15275252 0.15275252 0.06546537]

normalized_weights = weights / np.sum(weights)
print(normalized_weights)
# [0.07142857 0.07142857 0.07142857 0.07142857 0.07142857 0.07142857
# 0.16666667 0.16666667 0.16666667 0.07142857]

```

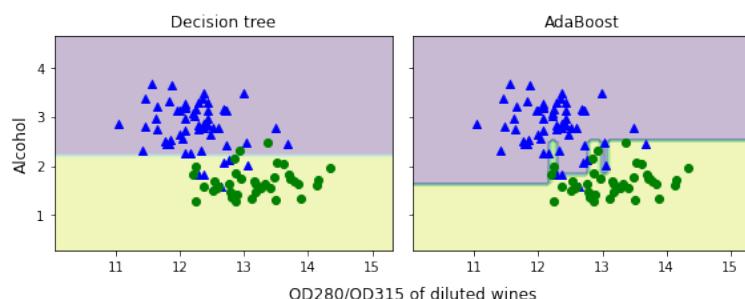
Теперь давайте напишем код полностью с помощью scikit-learn, будем использовать те данные, которые получили в прошлой части:

```

from sklearn.ensemble import AdaBoostClassifier
tree = DecisionTreeClassifier(criterion='entropy',
                             random_state=1,
                             max_depth=1) # tree stump
# learning_rate: weight applied to each classifier
# at each boosting iteration. A higher learning rate
# increases the contribution of each classifier.
ada = AdaBoostClassifier(base_estimator=tree,
                        n_estimators=500,
                        learning_rate=0.1,
                        random_state=1)
tree = tree.fit(X_train, y_train)
y_train_pred = tree.predict(X_train)
y_test_pred = tree.predict(X_test)
tree_train = accuracy_score(y_train, y_train_pred)
tree_test = accuracy_score(y_test, y_test_pred)
print(f'Decision tree train/test accuracies '
      f'{tree_train:.3f}/{tree_test:.3f}')
# Decision tree train/test accuracies 0.916/0.875
# аналогичный код, что и выше:
# AdaBoost tree train/test accuracies 1.000/0.917

```

Теперь посмотрим на decision boundary:



Код для вывода этой картинки можно найти в конце части 7.3, единственное изменения – это классификатор и название графика. Также заметим, что граница очень похожа на границу bagging classifier. Важно понимать, что ensemble сильно замедляет обучение и не всегда даёт большой прирост.

7.5 Gradient boosting – an ensemble based on loss gradients

Gradient boosting – другой вариант boosting концепции, который последовательно учит weak learners. Gradient boosting очень важная тема т.к. формирует базу для очень популярных алгоритмов, например, **XGBoost**. Немного сравним AdaBoost с gradient boosting. Gradient boosting также учит decision trees, используя prediction errors, однако глубина дерева обычно от 3 до 6. В отличие от AdaBoost, мы не используем prediction errors для sample weights, они используются сразу для создания target variable для след. дерева. Также, вместо весового множителя для каждого дерева, здесь мы имеем глобальный learning rate для всех деревьев сразу. Теперь опишем общий алгоритм gradient boosting, для простоты мы посмотрим на бинарную классификацию. Однако существует обобщение на несколько классов с помощью logistic loss. Также мы опустим gradient boosting для regression. Обновления между деревьями основаны на loss gradient. Напишем шаги general algorithm для gradient boosting:

1. Проинициализируем модель возвращать константную prediction value. Для этого будем использовать tree stump. Обозначим это значение как \hat{y} , найдём его, минимизируя дифференируемую loss function L , которую напишем позже:

$$F_0(x) = \arg \min_{\hat{y}} \sum_{i=1}^n L(y_i, \hat{y})$$

2. Для каждого дерева $m = 1, \dots, M$, где M – общее кол-во деревьев, заданное нами, выполним следующие шаги:

- (a) Посчитаем разность между предсказанным значением $F(x_i) = \hat{y}_i$ и меткой y_i . Это значение иногда называют **pseudo-response** или **pseudo-residual**. Более формально можно записать pseudo-residual как отрицание градиента loss функции:

$$r_{im} = - \left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)}, \text{ for } i = 1, \dots, n$$

Ещё раз обратим внимание, что $F(x)$ – предсказание на предыдущем дереве, так что на первом шаге – это та константа.

- (b) Обучим дерево на pseudo-residual r_{im} . Мы будем использовать обозначение R_{jm} , чтобы обозначить листовые вершины результирующего дерева на итерации m , $j = 1 \dots J_m$.
- (c) Для каждого листа R_{jm} мы считаем output value:

$$\gamma_{jm} = \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

Чуть дальше мы разберёмся как это значение γ_{jm} считается минимизацией loss функции.

- (d) Обновим модель добавлением output values γ_m к предыдущему дереву:

$$F_m(x) = F_{m-1}(x) + \eta \gamma_m$$

Learning rate η обычно лежит в прелах от 0.01 до 1. Эти маленькие шаги позволяют избежать переобучения.

Теперь, после общей структуры gradient boosting, адаптируем этот механизм на классификацию. Мы поговорим про детали реализации для binary classification. В этом контексте мы будем использовать logistic loss function (из главы 3):

$$L_i = -y_i \log p_i + (1 - y_i) \log (1 - p_i)$$

Также из главы 3 возьмём log (odds):

$$\hat{y} = \log (\text{odds}) = \log \left(\frac{p}{1-p} \right)$$

По причинам, которые мы поймём далее, будем использовать log (odds), чтобы переписать logistic loss function:

$$L_i = \log(1 + e^{\hat{y}_i}) - y_i \hat{y}_i$$

Теперь мы можем написать частные производные по отношению к \hat{y} :

$$\frac{\partial L_i}{\partial \hat{y}_i} = \frac{e^{\hat{y}_i}}{1 + e^{\hat{y}_i}} - y_i = p_i - y_i$$

Теперь, после этих выкладок, переформулируем шаги для бинарной классификации:

1. Создадим tree stump, который минимизирует logistic loss. Оказывается, что loss минимизируется, если root node возвращает log (odds), \hat{y} .
2. Для каждого дерева $m = 1, \dots, M$ сделаем следующие шаги:

- (a) Конвертируем log (odds) в вероятность (см. главу 3):

$$p = \frac{1}{1 + e^{-\hat{y}}}$$

Теперь посчитаем pseudo-residual:

$$-\frac{\partial L_i}{\partial \hat{y}_i} = y_i - p_i$$

- (b) Обучим новое дерево этим значениям.
- (c) Для каждого листа R_{jm} посчитаем значение γ_{jm} , которое минимизирует logistic loss function:

$$\gamma_{jm} = \arg \min_{\gamma} (\log(1 + e^{\hat{y}_i + \gamma}) - y_i(\hat{y}_i + \gamma))$$

В этом равенстве мы просто подставили loss функцию в общее равенство из прошлой схемы. Пропуская некоторые математические шаги, получаем:

$$\gamma_{jm} = \frac{\sum_i y_i - p_i}{\sum_i p_i(1 - p_i)}$$

Важно, что здесь суммирование только по примерам, которые относятся к листу R_{jm} , а не по всему training dataset.

- (d) Обновим модель:

$$F_m(x) = F_{m-1}(x) + \eta \gamma_m$$

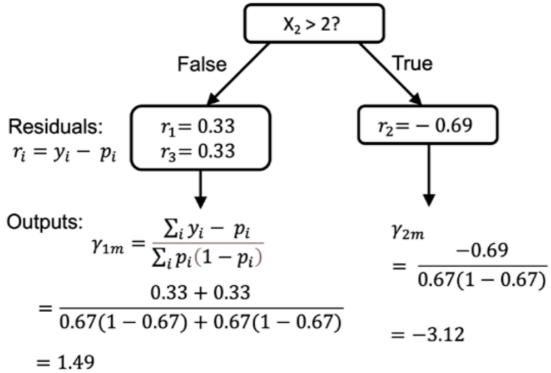
Возникает вопрос, почему деревья возвращают log (odds), а не просто вероятности. Если сложить просто значения вероятностей, то мы не сможем прийти к содержательному результату, поэтому, технически говоря, gradient boosting для классификации использует **regression trees**. Многие шаги до сих пор могут выглядеть довольно абстрактно, поэтому давайте их проиллюстрируем. Возьмём такой датасет:

| | Feature x_1 | Feature x_2 | Class label y |
|---|---------------|---------------|-----------------|
| 1 | 1.12 | 1.4 | 1 |
| 2 | 2.45 | 2.1 | 0 |
| 3 | 3.54 | 1.2 | 1 |

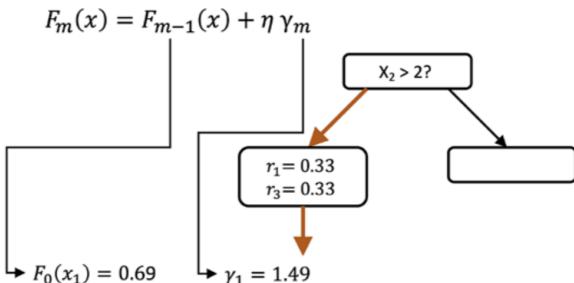
Сначала мы строим root node, считаем log (odds), потом конвертируем log (odds) в вероятности и считаем pseudo-residuals. Напомню, что odds можно посчитать как отношение положительных меток к отрицательным, в нашем примере это значение равно $\frac{2}{1} = 2$. Посмотрим на табличку:

| | Feature x_1 | Feature x_2 | Class label y | Step 1: $\hat{y} = \log(\text{odds})$ | Step 2a: $p = \frac{1}{1 + e^{-\hat{y}}}$ | Step 2a: $r = y - p$ |
|---|---------------|---------------|-----------------|--|--|-------------------------|
| 1 | 1.12 | 1.4 | 1 | 0.69 | 0.67 | 0.33 |
| 2 | 2.45 | 2.1 | 0 | 0.69 | 0.67 | -0.67 |
| 3 | 3.54 | 1.2 | 1 | 0.69 | 0.67 | 0.33 |

Теперь обучим новое дерево на pseudo-residuals r , а затем посчитаем output values γ , посмотрим на картинку дерева:



Пришло время обновить модель, всё можно увидеть на картинке:



Тогда в рамках первого тренировочного примера можно написать так:

$$F_1(x_1) = 0.69 + 0.1 \times 1.49 = 0.839$$

Аналогично можно сделать шаг $m = 2$, используя данные, полученные на шаге $m = 1$. Посмотрим на итоговую таблицу:

| x_1 | x_2 | y | Step 1: $F_0(x) = \hat{y}$ = log(odds) | Step 2a: $p = \frac{1}{1 + e^{-\hat{y}}}$ | Step 2a: $r = y - p$ | New log(odds) $\hat{y} = F_1(x)$ | Step 2a: p | Step 2a: r |
|-------|-------|-----|--|--|-------------------------|--|-----------------|-----------------|
| 1 | 1.12 | 1.4 | 1 | 0.69 | 0.67 | 0.33 | 0.839 | 0.698 |
| 2 | 2.45 | 2.1 | 0 | 0.69 | 0.67 | -0.67 | 0.378 | 0.593 |
| 3 | 3.54 | 1.2 | 1 | 0.69 | 0.67 | 0.33 | 0.839 | 0.698 |

Уже видно, что предсказанные значения выше для положительного класса и меньше для отрицательного. Следовательно, residuals тоже становятся меньше. Заметим, что этот процесс продолжается пока мы не получим M деревьев или пока residuals больше некоторого user-specified threshold. Когда алгоритм закончился, мы можем уже начать предсказывать, удерживая значение вероятности в точке 0.5 у модели $F_M(x)$. По сравнению с lr, здесь мы имеем несколько деревьев, а также мы можем получить нелинейную границу. Далее мы посмотрим на gradient boosting в действии. Рассмотрим использование **XGBoost**. В scikit-learn gradient boosting реализован как **GradientBoostingClassifier**. Важно, что gradient boosting – это последовательный процесс, из-за чего время обучения становится очень большим. Однако в последний год появилась более популярная реализация – XGBoost (extreme gradient boosting). Этот алгоритм делает некоторые трюки и приближения, что значительно ускоряет тренировку, в свою очередь эти приближения имеют очень хороший результат. Есть также и другие реализации: **LightGBM** и **CatBoost**. На основе первого scikit-learn сейчас пишет реализацию **HistGradientBoostingClassifier**. Но всё же XGBoost всё ещё сейчас самый популярный, а также выигрывал множество Kaggle competitions, поэтому будем использовать именно его. А вот и код:

```
# можно установить так:  
! pip install xgboost
```

```
# или так, указав версию:  
! pip install XGBoost==1.5.0
```

```
# к счастью XGBoosts XGBClassifier  
# работает так же как и scikit-learn API
```

```

import xgboost as xgb
model = xgb.XGBClassifier(n_estimators=1000, learning_rate=0.01,
                           max_depth=4, random_state=1,
                           use_label_encoder=False)
gbm = model.fit(X_train, y_train)
y_train_pred = gbm.predict(X_train)
y_test_pred = gbm.predict(X_test)
gbm_train = accuracy_score(y_train, y_train_pred)
gbm_test = accuracy_score(y_test, y_test_pred)
print(f'XGboost train/test accuracies :'
      f'{gbm_train:.3f}/{gbm_test:.3f}')
# XGboost train/test accuracies 0.937/0.917

```

Здесь мы обучили 1000 раундов, learning rate рекомендуют брать от 0.01 до 0.1, однако напомним, что он используется, чтобы масштабировать предсказания от индивидуальных раундов. Так что, интуитивно, чем меньше rate, тем больше раундов нужно, чтобы достичь хороших результатов. Высота каждого дерева должна быть от 2 до 6, однако большие значения также могут хорошо работать в зависимости от датасета. Параметр use_label_encoder=False убирает предупреждение, что алгоритм больше не конвертирует значения в цифры, так что это необходимо сделать самим, причём нумерация должна идти с 0. В следующей главе мы рассмотрим конкретное применение машинного обучения – sentiment analysis.

8 Sentiment Analysis

В современном мире мнение людей, reviews и рекомендации стали ценным ресурсом для политики и бизнеса. В этой главе мы окунёмся в подобласть **natural language processing (NLP)**, которая называется **sentiment analysis**, мы научимся использовать ml алгоритмы, чтобы классифицировать документы на основании их sentiment. Мы будем работать с датасетом Internet Movie Database (IMDb) – movie reviews и построим модель, которая будет делить отзывы на позитивные и негативные.

8.1 Preparing the IMDb movie review data for text processing

Sentiment analysis также иногда называют **opinion mining**, эта подобласть NLP занимается анализом sentiment документов. Популярная задача – классификация документов на основании выраженного мнения или эмоций в отношении конкретной темы. В этой главе мы будем работать с большим датасетом IMDb, который содержит более 50,000 reviews, помеченные как positive (≥ 6 stars) или negative (≤ 5 stars). Сначала мы загрузим датасет, далее сделаем его подходящим для ml алгоритмов и достанем значимую информацию. Получим датасет, формат – gzip-compressed tarball archive. Ссылка на скачивание архива и способ его скачать из Python будет ниже в коде. Открыть можно тремя способами: 7-Zip, tar -zxf aclImdb_v1.tar.gz или следующим кодом:

```

# скачивание архива:
import urllib.request
urllib.request.urlretrieve(
    'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz',
    'aclImdb_v1.tar.gz')

# распаковка архива:
import tarfile
# второй параметр:
# open for reading with gzip compression
with tarfile.open('aclImdb_v1.tar.gz', 'r:gz') as tar:
    tar.extractall()

```

Теперь сделаем предварительную обработку данных в более удобный формат. Соберём отдельные текстовые документы из архива в один CSV file. В следующем коде мы будем читать reviews в pandas DataFrame, что может занять до 10 минут. Чтобы визуализировать прогресс и оценить время до окончания, будем использовать **Python Progress Indicator (PyPrind)** пакет, его можно установить через команду pip install pyprind. А вот и код:

```

! pip install pyprind

import pyprind
import pandas as pd
import os

# путь до директории с распакованным
# архивом reviews, я поставил путь

```

```

# до текущего каталога, где мы сейчас
# находимся + название папки
basepath = os.getcwd() + '/aclImdb'

labels = { 'pos': 1, 'neg': 0}
# 1 - stdout, 2 - stderr
# 50000 - количество файлов
pbar = pyprind.ProgBar(50000, stream=1)
df = pd.DataFrame()
# папки для train и test сетов
for s in ('test', 'train'):
    # подпапки для 1 и 0
    for l in ('pos', 'neg'):
        # правильно соединяет компоненты пути:
        path = os.path.join(basepath, s, l)
        # все файлы и каталоги:
        for file in sorted(os.listdir(path)):
            with open(os.path.join(path, file),
                      'r', encoding='utf-8') as infile:
                txt = infile.read()
                # True - результирующие колонки будут
                # помечены так: 0, ..., n-1
                df = df.append([[txt, labels[l]]],
                               ignore_index=True)
    pbar.update()
df.columns = ['review', 'sentiment']
# Выглядеть это должно так:
# 0% #####] 100% / ETA: 00:00:31
# В конце будет такое:
# Total time elapsed: 00:01:51

```

Из np.random подмодуля мы возьмём функцию permutation, чтобы перемешать DataFrame, т.к. он у нас отсортированный; это будет полезно, когда мы будем делить датасет на train и test, подтягивая датасет уже из локальной директории. Сохраним датасет:

```

import numpy as np
np.random.seed(0)
df = df.reindex(np.random.permutation(df.index))
# index - писать или не писать названия
# рядов в файл
df.to_csv('movie_data.csv', index=False, encoding='utf-8')

```

Чтобы убедиться, что всё прошло хорошо, выведем первые 3 строки:

```

df = pd.read_csv('movie_data.csv', encoding='utf-8')
# может понадобиться для некоторых компьютеров:
df = df.rename(columns={"0": "review", "1": "sentiment"})
print(df.head(3))
# Выход:
#          review  sentiment
# 0  at a Saturday matinee in my home town. I went ...      0
# 1  I love this movie. It is the first film Master...      1
# 2  In the voice over which begins the film, Hugh...      1

# для самопроверки:
print(df.shape == (50000, 2)) # True

```

8.2 Bag-of-words model

Из главы 4 мы помним, что categorical data (слова, текст) нужно конвертировать в числа перед тем как подать ml алгоритму. В этой части мы рассмотрим **bag-of-words** модель, которая как раз позволяет представить текст в виде вектора из чисел. Алгоритм очень простой, напишем два пункта:

1. Делаем словарь unique tokens, например, множество слов из всех документов.
2. Делаем feature vector из каждого документа, который содержит встречаемость каждого слова из словаря.

Так как уникальные слова в каждом тексте занимают маленькое подмножество слов, то вектор будет состоять из большого кол-ва нулей, поэтому его называют **sparse**. Мы можем использовать CountVectorizer класс, он принимает текстовый массив, это могут быть предложения или документы. Числа, которые ставят в вектор фич называются **raw term frequencies**, обозначение: $tf(t, d)$ – частота термина t в документе d . Порядок слов в этом алгоритме не имеет значения, он определяется лишь порядком в словаре, он обычно алфавитный. См. код:

```

import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
count = CountVectorizer()
docs = np.array(['The sun is shining',
                'The weather is sweet',
                'The sun is shining, the weather is sweet,',
                'and one and one is two'])
bag = count.fit_transform(docs)
print(count.vocabulary_)
# Вывод:
# {'the': 6, 'sun': 4, 'is': 1, 'shining': 3, 'weather': 8,
# 'sweet': 5, 'and': 0, 'one': 2, 'two': 7}
# Каждому слову сопоставлен индекс, он как раз и используется
# для конструирования фич, на эту позицию ставится частота
# встречаемости.

print(bag.toarray())
# Получаем векторы фич:
# [[0 1 0 1 1 0 1 0 0]
# [0 1 0 0 0 1 1 0 1]
# [2 3 2 1 1 1 2 1 1]]

```

Немного поговорим про **N-gram models**. Последовательность объектов в bag-of-words model также называется 1-gram (или unigram) model – каждый токен в словаре представляет одно слово. В общем случае непрерывная последовательность элементов – слов, букв или символов – называется n-grams. Какое выбрать n зависит от задачи, например, размер 3 и 4 хорошо подходят для anti-spam filtering of email messages. Пример для предложения ‘the sun is shining’:

- 1-gram: ‘the’, ‘sun’, ‘is’, ‘shining’
- 2-gram: ‘the sun’, ‘sun is’, ‘is shining’

Класс CountVectorizer поддерживает выбор n с помощью параметра ngram_range. Например, 2-gram записывают так: ngram_range = (2, 2). Поговорим про оценку релевантности слов с помощью **term frequency-inverse document frequency (tf-idf)**. При анализе документов мы часто встречаемся со словами в разных документах из обоих классов. Эти часто встречающиеся слова не несут полезной информации, поэтому мы изучим технику tf-idf, которая понижает веса у этих частых слов в векторе фич. Значение tf-idf считается так:

$$tf\text{-}idf = tf(t, d) \times idf(t, d)$$

, где $idf(t, d)$ – inverse document frequency, который считается так:

$$idf(t, d) = \log \frac{n_d}{1 + df(d, t)}$$

, где n_d – общее кол-во документов, $df(d, t)$ – кол-во документов d , которые содержат слово t . Заметим, что 1 опциональна и служит для того, чтобы термины, которые не встречаются вообще, имели значение $\neq 0$. log используется, чтобы редкие термины не получали очень большие значения. В библиотеке есть класс TfidfTransformer, который принимает raw term frequencies из CountVectorizer и возвращает tf-idfs:

```

from sklearn.feature_extraction.text import TfidfTransformer
tfidf = TfidfTransformer(use_idf=True,
                         norm='l2',
                         smooth_idf=True)
np.set_printoptions(precision=2)
print(tfidf.fit_transform(count.fit_transform(docs)))
.toarray()
# [[0.        0.43     0.      0.56    0.56   0.        0.43    0.        0.      ]
# [0.        0.43     0.      0.        0.56    0.43    0.        0.56    0.      ]
# [0.5      0.45     0.5     0.19    0.19    0.19    0.3      0.25    0.19  ]]

```

Слово ‘is’ в третьем документе встречается очень часто, но так как оно есть и в первом и во втором, то значение 0.45 мало отличается от 0.43, с другой стороны посмотрите на первую колонку. Однако, если мы посчитаем значения руками, то увидим отличия, в scikit-learn формулы чуть-чуть другие:

$$idf(t, d) = \log \frac{1 + n_d}{1 + df(d, t)} \text{ и } tf\text{-}idf = tf(t, d) \times (idf(t, d) + 1)$$

Параметр smooth_idf=True как раз прибавляет +1 в первом равенстве, это полезно для присвоения 0 тем словами, которые есть во всех документах: $idf(t, d) = \log(1) = 0$. Также часто нормализуют

raw term frequencies перед тем, как считать *tf-idfs*, однако библиотечный класс нормализует сразу весь *tf-idfs*, по умолчанию (*norm='l2'*) применяется L2-normalization:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}} = \frac{v}{(\sum_{i=1}^n v_i^2)^{1/2}}$$

Остался последний параметр, *use_idf=True*, он как раз активирует наши формулы, в противном случае $idf(t) = 1$. Теперь можно двинуться дальше и применить эту концепцию ко всему датасету. Однако есть ещё один важный шаг – **cleaning text data** – убрать все нежелательные символы. Чтобы показать почему это важно, давайте выведем последние 50 символов первого документа в перемешанном датасете:

```
print(df.loc[0, 'review'][-50:])
# Вывод:
# is seven.<br /><br />Title (Brazil): Not Available
```

Мы можем видеть, что в тексте помимо обычных букв есть знаки пунктуации, HTML markup и другие символы. В то время как HTML markup не содержит полезной информации, пунктуация может сказать очень много в частных NLP контекстах. Однако для простоты мы уберём всю пунктуацию кроме символов эмоций, например, :). Чтобы это сделать, будем использовать библиотеку **re** – regular expression (regex):

```
import re
def preprocessor(text):
    # убираем все HTML markup символы
    # однако, часто рекомендуют парсить
    # документ вместо использования
    # regex, но здесь хватит и его
    text = re.sub('<[^>]*>', '', text)
    # найдём все смайлики
    emoticons = re.findall('(:|;|=)(:-)?(:\\)|\\(|D|P)', text)
    # убираем все non-word символы
    # join соединяет через пробел
    text = (re.sub('[\\W]+', ' ', text.lower()) +
            ''.join(emoticons).replace('-', ''))
    return text

# для желающих использовать Python's HTML parser module:
# from html.parser import HTMLParser
```

Понимать регулярные выражения очень полезно, поэтому поясню то, что написано в коде. Первое выражение: '<' – ждём угловую скобку, внутри 0 или более раз находится любой символ кроме '>', ждём угловую скобку '>'. Второе выражение: '?:' соответствует упрощённой форме записи без скобок, первая группа в скобках – ждём ':', ';' или '='; вторая группа в скобках – ждём '-', вопрос после скобки говорит, что ждём последнюю группу 0 или 1 раз, третья группа скобок – ждём ')', или '(', или 'D', или 'P'. В контексте анализа, большие буквы в начале предложения или собственных имён не несут важной информации, опять же это лишь упрощающее предположение. Конечно, добавлять в конец эмоции – не самый хороший способ, но вспомним, что у нас one-word tokens. Проверим, что функция работает и применим ко всему датасету:

```
print(preprocessor(df.loc[0, 'review'][-50:]))
# is seven title brazil not available
print(preprocessor('</a>This :) is :( a test :-)!!'))
# this is a test :) :( :)
```

```
df['review'] = df['review'].apply(preprocessor)
```

Теперь поговорим про обработку документов в токены. После подготовки датасета нужно думать о том, как разбить текст в отдельные элементы. Один из способов **tokenize** документ – разбить в отдельные слова по пробельному символу:

```
def tokenizer(text):
    return text.split()
print(tokenizer('runners like running and thus they run'))
# ['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

В контексте токенизации есть ещё одна полезная техника – **word stemming**, которая состоит в выделении корня слова, это позволяет отнести однокоренные слова к одному корню. Есть ещё одно название в честь создателя – **Porter stemmer algorithm**. **Natural Language Toolkit (NLTK)** для Python реализует этот алгоритм, установить можно так: *pip install nltk*. См. код:

```
from nltk.stem.porter import PorterStemmer
porter = PorterStemmer()
```

```

def tokenizer_porter(text):
    return [porter.stem(word) for word in text.split()]
print(tokenizer_porter('runners like running and thus they run'))
# ['runner', 'like', 'run', 'and', 'thu', 'they', 'run']

```

Буквально чуть-чуть поговорим про stemming algorithms. Porter stemming algorithm – один из самых старых и простых алгоритмов. Другие популярные алгоритмы: **Snowball stemmer** (Porter2 или English stemmer) и **Lancaster stemmer** (Paice/Husk stemmer). Оба алгоритма есть в пакете nltk, также они оба быстрее porter алгоритма, однако Lancaster stemmer более агрессивный, т.е. слова будут короче и менее ясные. Stemming может создавать нереальные слова ('thu' из 'thus'), техника **lemmatization** получает каноническую (грамматически корректную) форму идивидуальных слов – *lemmas*. Однако эта техника вычислительно более сложная и на практике было показано, что и stemming и lemmatization имеют слабое влияние на качество text classification. Перед тем, как перейти в следующую часть, где мы обучим ml модель, используя bag-of-words модель, давайте кратко поговорим о ещё одном топике – **stop word removal**. Stop words – очень общие слова, которые всегда есть во всех видах текстов, это полезно, если мы работаем с raw или normalized term frequencies, а не с *tf-idfs*, который уже сам уменьшает веса. Чтобы убрать stop words, мы будем использовать множество 127 слов, которые доступны из NLTK библиотеки, которые можно получить так:

```

import nltk
nltk.download('stopwords')

```

После загрузки слов, мы можем уже применить эти слова:

```

from nltk.corpus import stopwords
stop = stopwords.words('english')
x = {w for w in tokenizer_porter('a runner likes'
                                 ' running and runs a lot')
      if w not in stop}
print(x)
# {'run', 'runner', 'like', 'lot'}

```

8.3 Logistic regression for document classification

В этой части мы напишем lr, чтобы классифицировать reviews на positive и negative, в основе будет bag-of-words model. Сначала мы разделим DataFrame из cleaned text documents пополам для тренировки и тестирования:

```

X_train = df.loc[:25000, 'review'].values
y_train = df.loc[:25000, 'sentiment'].values
X_test = df.loc[25000:, 'review'].values
y_test = df.loc[25000:, 'sentiment'].values

```

Далее мы будем использовать GridSearchCV, чтобы найти оптимальный набор параметров для lr, используя 5-fold stratified cross-validation:

```

from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer

# lowercase – приводить или нет к маленьким буквам
# preprocessor – можно добавить шаг препроцессирования
# strip_accents – убирает ударения и делает некоторые
# другие шаги нормализации символов
tfidf = TfidfVectorizer(strip_accents=None,
                       lowercase=False,
                       preprocessor=None)

small_param_grid = [
    {
        'vect__ngram_range': [(1, 1)],
        'vect__stop_words': [None],
        'vect__tokenizer': [tokenizer, tokenizer_porter],
        'clf__penalty': ['l2'],
        'clf__C': [1.0, 10.0]
    },
    {
        'vect__ngram_range': [(1, 1)],
        'vect__stop_words': [stop, None],
        'vect__tokenizer': [tokenizer],
        'vect__use_idf': [False],
        'vect__norm': [None],
    }
]

```

```

        'clf__penalty': ['l2'],
        'clf__C': [1.0, 10.0]
    },
]

lr_tfidf = Pipeline([
    ('vect', tfidf),
    ('clf', LogisticRegression(solver='liblinear'))
])

# verbose – степень детализации вывода
# степени можно прочитать в документации
gs_lr_tfidf = GridSearchCV(lr_tfidf, small_param_grid,
                           scoring='accuracy', cv=5,
                           verbose=2, n_jobs=1)

gs_lr_tfidf.fit(X_train, y_train)

# Вывод:
# Fitting 5 folds for each of 8 candidates, totalling 40 fits
# [CV] END clf__C=1.0, clf__penalty=l2, vect__ngram_range=(1, 1), vect__stop_words=None,
#       vect__tokenizer=<function tokenizer at 0x7fc36c9325f0>; total time= 4.1s
# ...

```

Заметим, что мы используем для lr liblinear solver, а не дефолтный lbfgs, т.к. на больших датасетах он может работать лучше. В коде выше n_jobs=1, но лучше всего поменять на -1, чтобы задействовать все доступные cores. Также можно заменить [tokenizer, tokenizer_porter] на [str.split]. Так как число фич такого же размера как и размер самого большого словаря, то поиск может занять довольно много времени. В коде выше мы заменили CountVectorizer и TfidfTransformer на TfidfVectorizer, который комбинирует их вместе. Для эксперимента можно добавить ещё и l1 регуляризацию. После завершения поиска, посмотрим на результаты:

```

print(f'Best parameter set: {gs_lr_tfidf.best_params_}')
# Best parameter set: {'clf__C': 10.0, 'clf__penalty': 'l2',
#                      'vect__ngram_range': (1, 1), 'vect__stop_words': None,
#                      'vect__tokenizer': <function tokenizer at 0x7fc36c9325f0>}

# average 5-fold cross-validation accuracy scores:
print(f'CV Accuracy: {gs_lr_tfidf.best_score_:.3f}')
# CV Accuracy: 0.897

clf = gs_lr_tfidf.best_estimator_
# classification accuracy:
print(f'Test Accuracy: {clf.score(X_test, y_test):.3f}')
# Test Accuracy: 0.899

```

Пару слов про **naïve Bayes classifier**. Всё ещё популярный классификатор для текстовых данных – naïve Bayes classifier, который получил свою популярность в фильтрации спама для email. Этот алгоритм легко реализовать, он очень эффективен и довольно хорошо работает на маленьких датасетах по сравнению с остальными алгоритмами, безусловно, все версии этого алгоритма реализованы в scikit-learn.

8.4 Online algorithms and out-of-core learning

Выполняя код в предыдущей части, вы, наверное, заметили, что конструировать вектор фич для 50,000 примеров во время grid search – это вычислительно очень сложно. Во многих реальных задачах работать с данным, которые даже не влезают в память компьютера, абсолютно нормально. Есть техника **out-of-core learning**, которая как раз позволяет работать с такими датасетами, обучая классификатор постепенно на маленьких batches of a dataset. Также возможна классификация текста с помощью **recurrent neural networks**, в главе 15 мы к этой задаче вернёмся и обучим deep learning-based classifier. Этот neural network-based classifier использует тот же самый out-of-core принцип, используя sgd алгоритм, но не требует конструирования bag-of-words модели. Stochastic gradient descent, как мы поняли в главе 2, – это optimization algorithm, который обновляет веса модели, используя один пример за раз. Сейчас мы будем использовать partial_fit функцию у SGDClassifier, чтобы брать документы напрямую из нашего локального диска и обучать lr модель, используя small mini-batches из документов. Для начала объявим функцию tokenizer, которая чистит текст, разделяет его на токены и убирает stop words:

```

import numpy as np
import re
from nltk.corpus import stopwords
stop = stopwords.words('english')

```

```

def tokenizer(text):
    text = re.sub('<[^>]*>', ' ', text)
    emoticons = re.findall('(?:[:|;|=](?:-)?:\\) |\\(|d|p)', text.lower())
    text = (re.sub('[\\W]+', ' ', text.lower()) +
            ''.join(emoticons).replace('-', ''))
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized

```

Теперь объявим generator function, stream_docs, которая читает и возвращает один документ за раз:

```

def stream_docs(path):
    with open(path, 'r', encoding='utf-8') as csv:
        # пропустим заголовок:
        next(csv)
        for line in csv:
            # через берётся до -3, не включая -3
            # в данном случае -1 символ — это перевод
            # строки, -2 — это метка
            text, label = line[:-3], int(line[-2])
            yield text, label

# проверим, что правильно работает:
stream = stream_docs(path='movie_data.csv')
print(next(stream)) # первый пример
print(next(stream)) # второй пример

```

Объявим функцию get_minibatch, которая принимает stream документов и возвращает указанное число примеров:

```

def get_minibatch(doc_stream, size):
    docs, y = [], []
    try:
        for _ in range(size):
            text, label = next(doc_stream)
            docs.append(text)
            y.append(label)
    except StopIteration:
        return None, None
    return docs, y

```

К сожалению, мы не можем использовать CountVectorizer для out-of-core learning т.к. ему нужно держать весь словарь в памяти, а TfidfVectorizer нужно держать все вектора фич в памяти, чтобы посчитать idf. Однако есть ещё один полезный vectorizer — **HashingVectorizer** — это data-independent алгоритм, который использует hashing trick через 32-bit MurmurHash3 функцию. См. код:

```

from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.linear_model import SGDClassifier

# decode_error — инструкции, которые нужно выполнить,
# если символы не соответствуют кодировке

# n_features=2**21=2,097,152 — кол-во колонок (фич)
# в выходной матрице
vect = HashingVectorizer(decode_error='ignore',
                         n_features=2**21,
                         preprocessor=None,
                         tokenizer=tokenizer)
clf = SGDClassifier(loss='log', random_state=1)
doc_stream = stream_docs(path='movie_data.csv')

```

Мы создали logistic regression classifier, установив loss='log'. Выбирая большее кол-во фич, мы уменьшаем вероятность hash коллизий, но в тоже время мы увеличиваем число коэффициентов в нашей модели. Теперь, имея все дополнительные функции, можно начать out-of-core learning:

```

import pyprind
pbar = pyprind.ProgBar(45)
classes = np.array([0, 1])
for _ in range(45):
    # mini-batch
    X_train, y_train = get_minibatch(doc_stream, size=1000)
    if not X_train:
        break
    X_train = vect.transform(X_train)
    clf.partial_fit(X_train, y_train, classes=classes)
    pbar.update()

```

```

# 0% ##### 100% / ETA: 00:00:00
# Total time elapsed: 00:00:37

# после окончания incremental learning process,
# посчитаем performance:
X_test, y_test = get_minibatch(doc_stream, size=5000)
X_test = vect.transform(X_test)
print(f'Accuracy: {clf.score(X_test, y_test):.3f}')
# Accuracy: 0.868

```

Если вы столкнулись с NoneType error, то значит вы запустили последнюю ячейку кода два раза, то есть на тестовые данные не осталось документов, поэтому get_minibatch вернул None, чтобы это исправить, нужно запустить код, начиная с stream_docs функции. Мы получили приблизительно 87% точности, это меньше чем нам дал grid search, однако текущий алгоритм очень эффективен по памяти и работал он меньше минуты, а это очень круто. Также можно использовать последние 5,000 документов, чтобы обновить модель:

```
clf = clf.partial_fit(X_test, y_test)
```

Более современная альтернатива bag-of-words – **word2vec model** – это unsupervised алгоритм, основанный на нейронных сетях, который пытается автоматически выучить взаимоотношения между словами. Идея word2vec – класть слова, которые имеют похожие значения, в одинаковые кластеры; через умный **vector spacing** модель может воспроизводить конкретные слова, используя простую векторную математику, например, *king – man + woman = queen*.

8.5 Topic modeling with latent Dirichlet allocation

Topic modeling описывает широкую задачу присваивания topics непомеченным текстовым документам. Например, типичное применение – категоризация документов в большом объёме газетных статей. В применениях topic modeling наша цель – присвоить category labels к статьям, например, спорт, финансы, мировые новости, политика и т.д. В рамках главы 1, получается, что это unsupervised обучение и это задача кластеризации. В этой части мы обсудим популярную технику для topic modeling – **latent Dirichlet allocation (LDA)**. Не надо путать с linear discriminant analysis – supervised техника сжатия размерностей, которую мы прошли в главе 5. Поговорим про декомпозицию текстовых документов с помощью LDA. Так как математика для LDA довольно сложная и требует знания Bayesian inference, мы обсудим эту тему с точки зрения практики. LDA – это generative probabilistic модель, которая пытается найти группы слов, которые встречаются часто вместе в разных документах. Эти частые слова и представляют наши темы, в предположении, что каждый документ – смесь разных слов. Вход для LDA – bag-of-words модель, которую мы обсудили чуть ранее. Получая на вход bag-of-words матрицу, LDA раскладывает её на новые две матрицы: document-to-topic и word-to-topic матрицы. LDA раскладывает bag-of-words матрицу так, что если мы умножим полученные две матрицы, то мы сможем воспроизвести исходную матрицу с минимально возможной ошибкой. Единственный недостаток – нужно вручную указать гиперпараметр – кол-во топиков. Теперь посмотрим на LDA в scikit-learn – класс **LatentDirichletAllocation** для разложения movie dataset и его категоризации на разные топики. В примере, который будет далее, мы ограничим кол-во тем до 10, но для интереса можно попробовать и больше. А вот и код:

```

import pandas as pd
df = pd.read_csv('movie_data.csv', encoding='utf-8')
df = df.rename(columns={"0": "review", "1": "sentiment"})

# сделаем bag-of-words матрицу

# max_df – maximum document frequency слов, мы поставили
# 10%, чтобы исключить слишком частые слова в документах

# max_features – мы ограничили кол-во фич 5000 самыми
# частыми словами, чтобы уменьшить размерность, что в
# свою очередь улучшает качество модели
from sklearn.feature_extraction.text import CountVectorizer
count = CountVectorizer(stop_words='english',
                       max_df=.1,
                       max_features=5000)
X = count.fit_transform(df['review'].values)

# learning_method='batch' – мы позволяем lda делать
# вычисления на основе всего training датасета на
# одной итерации, это медленнее, чем альтернатива –
# 'online' learning method, но это даёт более точные результаты
# 'online' – аналог online/mini-batch learning, глава 2

```

```

from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_components=10,
                                 random_state=123,
                                 learning_method='batch')
X_topics = lda.fit_transform(X)

Реализация LDA в scikit-learn использует expectation-maximization (EM) алгоритм, чтобы обновлять оценки параметров итеративно; мы не проходили этот алгоритм. Теперь получим эти топики:

print(lda.components_.shape) # (10, 5000)
# матрица важности слов, здесь 5000 для каждого
# из 10 топиков в порядке возрастания

# для анализа выберем 5 самых важных слов для
# каждого топика
n_top_words = 5
# имена колонок в матрице
feature_names = count.get_feature_names_out()
for topic_idx, topic in enumerate(lda.components_):
    print(f'Topic {topic_idx + 1}:')
    print(' '.join([
        feature_names[i]
        # argsort - такие индексы, что массив возрастает
        for i in topic.argsort() \\
        # с конца до -n_top_words - 1 не вкл. с шагом -1
        [-n_top_words - 1:-1]]))

# Вывод:
# Topic 1:
# worst minutes awful script stupid ~ Generally bad movies
# Topic 2:
# family mother father children girl ~ Movies about families
# Topic 3:
# american war dvd music tv ~ War movies
# Topic 4:
# human audience cinema art sense ~ Art movies
# Topic 5:
# police guy car dead murder ~ Crime movies
# Topic 6:
# horror house sex girl woman ~ Horror movies
# Topic 7:
# role performance comedy actor performances ~ Comedy movie reviews
# Topic 8:
# series episode war episodes tv ~ Movies somehow related to TV shows
# Topic 9:
# book version original read novel ~ Movies based on books
# Topic 10:
# action fight guy guys cool ~ Action movies

```

Проверим наши результаты на корректность:

```

# [::-1] - переворот
# X_topics содержит вероятности каждого из топиков
# для всех 50,000 review
# 5 - индекс ужасиков, выберем 3 самых вероятных
horror = X_topics[:, 5].argsort() [::-1]
for iter_idx, movie_idx in enumerate(horror[:3]):
    print(f'\nHorror movie #{(iter_idx + 1)}:')
    print(df['review'][movie_idx][:60], '...')

# Вывод:
# Horror movie #1:
# House of Dracula works from the same basic premise as House ...
# Horror movie #2:
# Okay, what the hell kind of TRASH have I been watching now? ...
# Horror movie #3:
# <br /><br />Horror movie time, Japanese style. Uzumaki/Spira ...

```

В следующей главе мы посмотрим на другую категорию supervised learning – *regression analysis*, которая позволяет нам предсказывать outcome variables непрерывно по сравнению с categorical class метками для моделей классификации, с которыми мы всё это время работали.

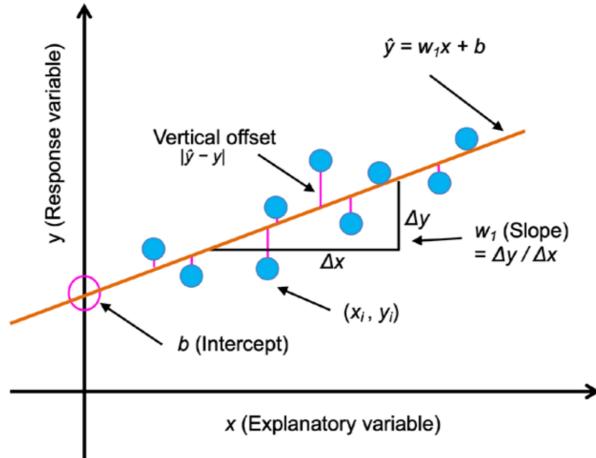
9 Regression Analysis

В этой главе мы будем заниматься предсказанием continuous target variables с помощью regression analysis. Так как мы даём непрерывный выход, то регрессия имеет применение в науке, прогнозах,

трендах и т.д.

9.1 Introducing linear regression

Цель линейной регрессии – моделировать отношения между одной или несколькими фичами и continuous target variable. Самый базовый тип лин. рег. – **simple linear regression**; мы поймём как обобщить на multivariate случай, где несколько фич. Как раз начнём с simple linear regression. Цель simple (univariate) лин. рег. – моделировать отношение между одной фичей (explanatory variable, x) и непрерывной переменной target (response variable, y). Она задаётся формулой $y = w_1x + b$. По сути мы ищем прямую линию, которая лучше всего подходит для наших данных. См. картинку:



Вот эту самую лучшую линию называют regression line, а вертикальные прямые от reg. line до примеров называют offsets или residuals – ошибки предсказания. Напишем формулу для **multiple linear regression**:

$$y = \sum_{i=1}^m w_i x_i + b = w^T x + b$$

9.2 Ames Housing dataset

Перед тем как написать свою первую модель лин. рег., мы проанализируем новый датасет о жилой недвижимости. Каждый новый датасет полезно визуализировать, чтобы лучше понять с чем мы имеем дело. Рекомендуется использовать `read_csv` функцию для tabular data в простом текстовом формате. Мы будем использовать от 1 до 5 фич, которые будут в коде далее, наша цель – предсказать цену. Немного кода:

```
import pandas as pd
columns = [ 'Overall Qual', 'Overall Cond', 'Gr Liv Area',
            'Central Air', 'Total Bsmt SF', 'SalePrice' ]
df = pd.read_csv('http://jse.amstat.org/v19n3/decock/AmesHousing.txt',
                 sep='\t', usecols=columns)
print(df.head())
print(df.shape) # (2930, 6)

df[ 'Central Air' ] = df[ 'Central Air' ].map({ 'N': 0, 'Y': 1})

df.isnull().sum()
# Total Bsmt SF      1
# есть только одно пропущенное значение
df = df.dropna(axis=0)
df.isnull().sum() # теперь всё хорошо
```

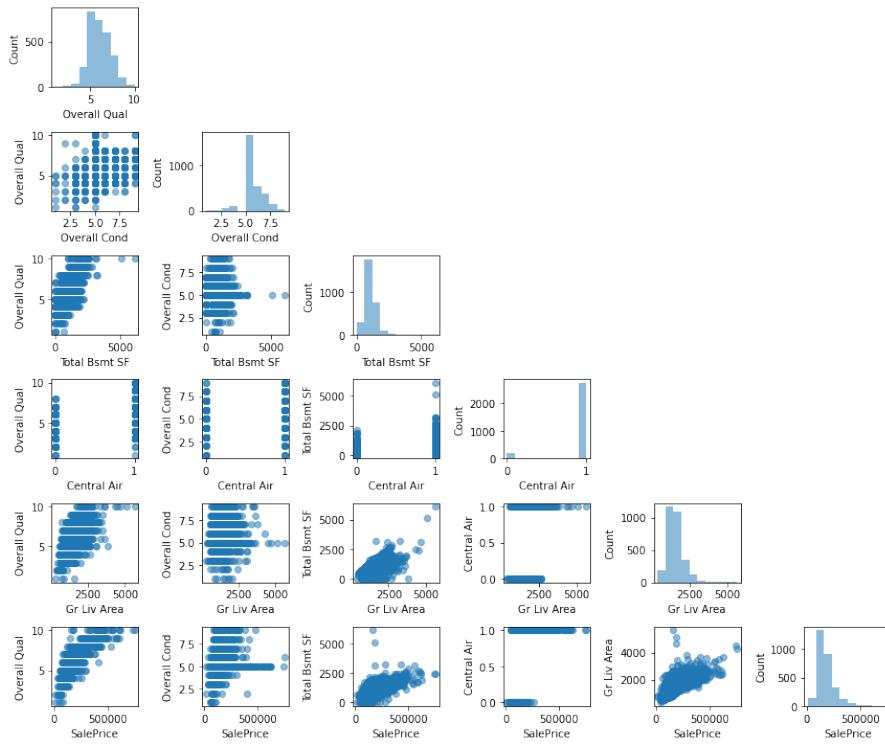
Теперь будем визуализировать важные характеристики датасета. Первый и рекомендованный шаг перед уже обучением самой модели – **Exploratory data analysis (EDA)**. Мы будем использовать простые, но всё же полезные техники из graphical EDA toolbox, которые могут помочь нам визуально определить выбросы, отношение между фичами или распределение данных. Сначала мы создадим **scatterplot matrix**, которая визуализирует попарную корреляцию между разными фичами в датасете. Чтобы нарисовать эту матрицу, будем использовать `scatterplotmatrix` функцию из `mlxtend` библиотеки (`pip install mlxtend`). А вот и код:

```
import matplotlib.pyplot as plt
from mlxtend.plotting import scatterplotmatrix
scatterplotmatrix(df.values, figsize=(12, 10)),
```

```

    names=df.columns , alpha=0.5)
plt.tight_layout()
plt.show()

```



Используя эту матрицу, мы можем быстро посмотреть как распределены данные и есть ли выбросы. Например, на пятой колонке последнего ряда наблюдается линейная зависимость цены от площади. Также мы можем посмотреть на histogram, это самый правый график последнего ряда, видно что переменная SalePrice искажена несколькими выбросами. Заметим, что несмотря на частое заблуждение, обучение модели лин. рег. не требует чтобы explanatory или target переменные были нормально распределены. Нормальное предположение требуется только для некоторых statistics и hypothesis тестов. Теперь посмотрим на взаимоотношения, используя **correlation matrix**. Мы визуализировали распределение данных в виде histograms и scatterplots, теперь мы сделаем correlation matrix, чтобы сделать количественную оценку линейных отношений между переменными. Correlation matrix очень похожа на covariance matrix, которую мы использовали для PCA. Мы можем интерпретировать эту матрицу как масштабированная версия covariance matrix. Матрица корреляций так же как и матрица ковариаций считается из стандартизованных фич. Correlation matrix – квадратная матрица, которая содержит **Pearson product-moment correlation coefficient (Pearson's r)**, который меряет линейную зависимость между парой фич. Коэффициент корреляции лежит от -1 до 1 . Две фичи имеют идеальную положительную корреляцию, если $r = 1$, не имеют корреляции при $r = 0$ и идеальную отрицательную корреляцию при $r = -1$. Pearson's correlation coefficient может быть посчитан как ковариация между двумя фичами x и y (числитель) разделённая на произведение их standard deviations (знаменатель):

$$r = \frac{\sum_{i=1}^n [(x^{(i)} - \mu_x)(y^{(i)} - \mu_y)]}{\sqrt{\sum_{i=1}^n (x^{(i)} - \mu_x)^2} \sqrt{\sum_{i=1}^n (y^{(i)} - \mu_y)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Здесь μ – среднее значение, σ_{xy} – ковариация между фичами x и y , а σ_x и σ_y – features' std's. Мы можем показать, что ковариация между парой стандартизованных фич равна их линейному коэффициенту корреляции. Покажем это, стандартизуем фичи x и y , чтобы получить их z -scores, которые обозначим как x' и y' :

$$x' = \frac{x - \mu_x}{\sigma_x}, \quad y' = \frac{y - \mu_y}{\sigma_y}$$

Напомню, что мы считаем covariance (population) между двумя фичами так:

$$\sigma_{xy} = \frac{1}{n} \sum_i^n (x^{(i)} - \mu_x)(y^{(i)} - \mu_y)$$

Так как стандартизация центрирует значения фич в нуле, covariance можно посчитать так:

$$\sigma'_{xy} = \frac{1}{n} \sum_i^n (x'^{(i)} - 0)(y'^{(i)} - 0)$$

Подставим значения стандартизированных фич:

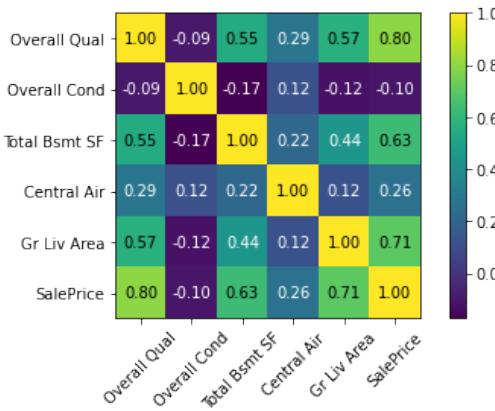
$$\sigma'_{xy} = \frac{1}{n} \sum_i^n \left(\frac{x - \mu_x}{\sigma_x} \right) \left(\frac{y - \mu_y}{\sigma_y} \right) = \frac{1}{n \cdot \sigma_x \sigma_y} \sum_i^n (x^{(i)} - \mu_x)(y^{(i)} - \mu_y) = \frac{\sigma_{xy}}{\sigma_x \sigma_y} = r$$

В этом коде мы будем использовать функцию corrcoef из NumPy на пяти фичах из предыдущего примера, а также функцию heatmap из mlxtend, чтобы нарисовать correlation matrix массив как heat map:

```
# для совместимости, иначе будет
# ошибка импорта
! pip install mlxtend==0.19.0

import numpy as np
from mlxtend.plotting import heatmap

cm = np.corrcoef(df.values.T)
hm = heatmap(cm, row_names=df.columns, column_names=df.columns)
plt.tight_layout()
plt.show()
```



Correlation matrix обеспечивает нас другой полезной информацией, которая может помочь нам выбрать фичи на основании их лин. корреляции. Чтобы обучить модель лин. рег., мы заинтересованы в тех фичах, которые умеют высокую корреляцию с нашей target переменной, т.е. SalePrice. Например, площадь имеет корреляцию 0.71, что выглядит как хороший выбор для exploratory переменной, чтобы сделать простую линейную регрессию.

9.3 Ordinary least squares linear regression model

В начале этой главы мы сказали, что лин. рег. можно понять как построение лучшей линии через наши тренировочные примеры, однако мы так и не определили что такое ‘лучшая’ линия и не обсудили разные техники обучения такой модели. Мы будем использовать **ordinary least squares (OLS)** метод (иногда называют **linear least squares**), чтобы оценить параметры линии, которая минимизирует сумму квадратов вертикальных расстояний (residuals или errors). Будем решать регрессию для параметров регрессии через gradient descent. Вспомним нашу реализацию Adaline, artificial neuron использует линейную функцию активации, также мы вводили loss функцию $L(w)$, которую мы оптимизировали через GD или SGD. Loss функция в Adaline – это была mean squared error (MSE), которая такая же для loss функции, которую мы используем в OLS:

$$L(w, b) = \frac{1}{2n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

Здесь $\hat{y} = w^T x + b$, на самом деле OLS регрессию можно понять как Adaline без threshold функции, то есть мы получаем continuous target переменные вместо классов 0 и 1. Чтобы это продемонстрировать, возмём GD реализацию для Adaline из главы 2 и уберём threshold функцию:

```

class LinearRegressionGD:
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.array([0.])
        self.losses_ = []
        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (errors ** 2).mean()
            self.losses_.append(loss)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_) + self.b_

    def predict(self, X):
        return self.net_input(X)

```

Чтобы увидеть наш LinearRegressionGD regressor в действии, будем использовать площадь дома как explanatory переменную, также мы стандартизируем переменные для лучшей сходимости GD алгоритма:

```

X = df[['Gr_Liv_Area']].values
y = df['SalePrice'].values
from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
sc_y = StandardScaler()
X_std = sc_x.fit_transform(X)
# flatten — выравнивает массив:
# np.array([[1], [[2]], [[3]]]).flatten() -> array([1, 2, 3])
y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()
lr = LinearRegressionGD(eta=0.1)
lr.fit(X_std, y_std)

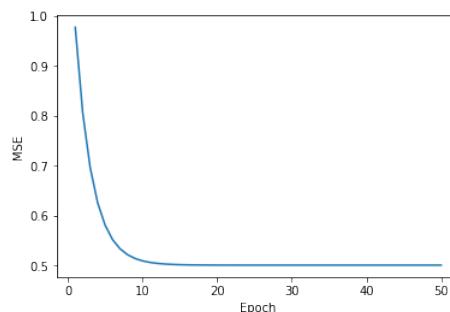
```

Большинство классов препроцессирования в scikit-learn ожидают на вход двумерный массив, поэтому мы сначала превратили y в столбик массивов (поменяв местами ‘:’ и newaxis, можно просто добавить размерность), после того как StandardScaler вернул масштабированные переменные, мы конвертировали массив обратно в одномерный. Хороший тон – рисовать loss функцию в зависимости от кол-ва эпох, когда мы используем алгоритмы оптимизации, чтобы проверить, что алгоритм сходится в минимуме (здесь он глобальный):

```

plt.plot(range(1, lr.n_iter+1), lr.losses_)
plt.ylabel('MSE')
plt.xlabel('Epoch')
plt.show()

```



Как мы видим, GD алгоритм сопротивляется уже приблизительно после 10 эпохи. Теперь давайте визуализируем нашу линию, для этого объявим простую функцию, которая будет рисовать scatterplot тренировочных примеров и саму линию:

```

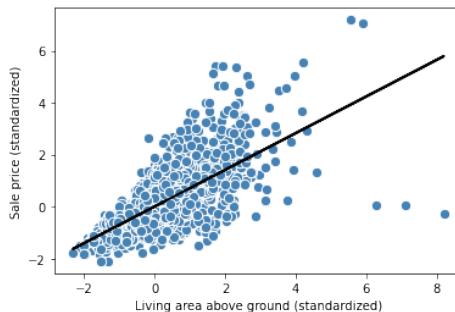
def lin_regplot(X, y, model):
    plt.scatter(X, y, c='steelblue', edgecolor='white', s=70)
    plt.plot(X, model.predict(X), color='black', lw=2)

```

```

lin_regplot(X_std, y_std, lr)
plt.xlabel('Living area above ground (standardized)')
plt.ylabel('Sale price (standardized)')
plt.show()

```



Как мы видим, эта прямая отражает общий тренд роста цены дома в зависимости от площади. Во многих случаях площадь даёт плохую оценку стоимости жилья, чуть позже мы научимся оценивать performance нашей модели. Мы можем видеть несколько выбросов, мы также обсудим как с ними бороться. Иногда важно дать предсказание в исходном масштабе:

```

feature_std = sc_x.transform(np.array([[2500]]))
target_std = lr.predict(feature_std)
# добавляем новую размерность с помощью reshape
target_reverted = sc_y.inverse_transform(target_std.reshape(-1, 1))
print(f'Sales price: ${target_reverted.flatten()[0]:.2f}')
# Sales price: $292507.07

```

Сделаем небольшое замечание, чисто технически мы не должны менять intercept параметр (например, b), если мы работаем со стандартизированными переменными, т.к. intercept оси y равен 0, проверим это:

```

print(f'Slope: {lr.w_[0]:.3f}') # Slope: 0.707
print(f'Intercept: {lr.b_[0]:.3f}') # Intercept: -0.000

```

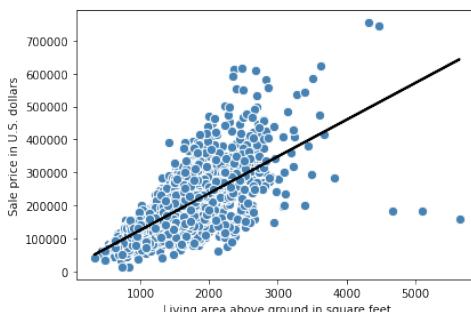
Поговорим про оценку коэффициента regression model через scikit-learn. Мы конечно реализовали рабочую версию лин. рег., однако в реальном мире нам интересны более эффективные реализации. Например, много моделей из scikit-learn для регрессии используют реализацию OLS из `scipy.linalg.lstsq`, которая использует оптимизированный код, основанный на Linear Algebra Package (LAPACK). Модель лин. рег. также работает лучше с нестандартизированными переменными, т.к. не использует (S)GD алгоритмы, поэтому шаг станд. можно опустить:

```

from sklearn.linear_model import LinearRegression
slr = LinearRegression()
slr.fit(X, y)
y_pred = slr.predict(X)
print(f'Slope: {slr.coef_[0]:.3f}') # 111.666
print(f'Intercept: {slr.intercept_:.3f}') # 13342.979

# коэффициенты другие, но прямая очень похожая
lin_regplot(X, y, slr)
plt.xlabel('Living area above ground in square feet')
plt.ylabel('Sale price in U.S. dollars')
plt.tight_layout()
plt.show()

```



Существуют аналитические решения лин. рег. Есть решение OLS с использованием системы лин. уравнений:

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

Реализация в Python:

```
# добавляем колонку из единиц:
Xb = np.hstack((np.ones((X.shape[0], 1)), X))
w = np.zeros(X.shape[1])
z = np.linalg.inv(np.dot(Xb.T, Xb))
w = np.dot(z, np.dot(Xb.T, y))
print(f'Slope: {w[1]:.3f}') # 111.666
print(f'Intercept: {w[0]:.3f}') # 13342.979
```

Преимущество метода в том, что мы гарантированно находим оптимальное решение аналитически. Однако, если мы работаем с большими датасетами, то это может быть вычислительно очень сложно обратить матрицу в этой формуле (иногда её называют normal equation), также матрица может быть сингулярной (необратимой), поэтому мы иногда предпочитаем итеративные методы. Есть разные решения линейной регрессии: GD, SGD, closed-form solution (как раз наша формула), QR факторизация и singular vector decomposition. Есть две библиотеки для этих и других моделей: *mlxtend* и *statsmodels*.

9.4 Продвинутое обсуждение линейной регрессии

Первая из трёх тем, которую мы обсудим в этой части – обучение надёжной regression model с использованием **RANSAC**. На модели лин. рег. сильно влияют выбросы, в некоторых ситуациях маленько подмножество данных может иметь большой эффект на коэффициенты модели. Много статистических тестов улавливают выбросы, однако удаление выбросов – задача именно data scientists. В качестве альтернативы удалению выбросов, мы посмотрим на надёжный метод регрессии – RANdom SAmple Consensus (RANSAC) алгоритм, который обучает модель регрессии подмножеству данных **inliers**. Напишем 5 шагов итеративного RANSAC алгоритма:

1. Выберем случайное число примеров, которые будут inliers и на которых обучим модель.
2. Протестируем остальные точки на нашу обученную модель, и добавим те точки, которые попадут в заданный нами допуск (tolerance to the inliers).
3. Переобучим модель на всех inliers.
4. Оценим ошибку обученной модели versus inliers.
5. Закончим алгоритм, если качество модели достигло некоторого threshold или прошло фиксированное число итераций.

Напишем код:

```
from sklearn.linear_model import RANSACRegressor
# max_trials – кол-во раундов
# min_samples – минимальное число примеров из исходного
# датасета
# residual_threshold – максимальный residual примера,
# чтобы считать его inlier
ransac = RANSACRegressor(
    LinearRegression(),
    max_trials=100, # default value
    min_samples=0.95,
    residual_threshold=None, # default value
    random_state=123)
ransac.fit(X, y)
```

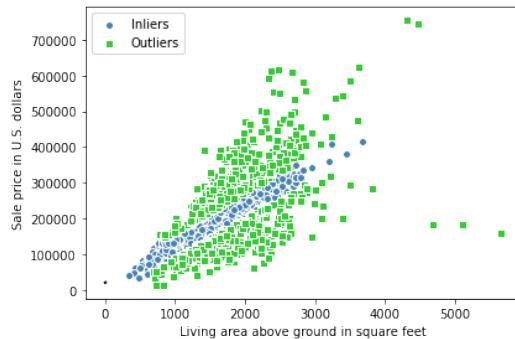
По умолчанию (`residual_threshold=None`), scikit-learn использует **MAD** оценку, чтобы выбрать inlier threshold, где MAD – это **median absolute deviation** target переменных y . Однако выбор inlier threshold зависит от задачи, что является недостатком этого алгоритма. В последние годы было придумано множество разных подходов для автоматического выбора inlier threshold. Давайте получим inliers и outliers из обученной RANSAC лин. рег. модели и нарисуем их вместе:

```
inlier_mask = ransac.inlier_mask
outlier_mask = np.logical_not(inlier_mask)
line_X = np.arange(3, 10, 1)
line_y_ransac = ransac.predict(line_X[:, np.newaxis])
plt.scatter(X[inlier_mask], y[inlier_mask],
            c='steelblue', edgecolor='white',
            marker='o', label='Inliers')
plt.scatter(X[outlier_mask], y[outlier_mask],
            c='limegreen', edgecolor='white',
            marker='s', label='Outliers')
plt.plot(line_X, line_y_ransac,
```

```

color='black', lw=2)
plt.xlabel('Living area above ground in square feet')
plt.ylabel('Sale price in U.S. dollars')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()

```



Как мы видим модель лин. рег. была обучена на множестве inliers. Когда мы напечатаем slope и intercept модели, linear regression line будет немного отличаться от той, которую мы получили ранее без использования RANSAC:

```

print(f'Slope: {ransac.estimator_.coef_[0]:.3f}') # 106.348
print(f'Intercept: {ransac.estimator_.intercept_:.3f}') # 20190.093

```

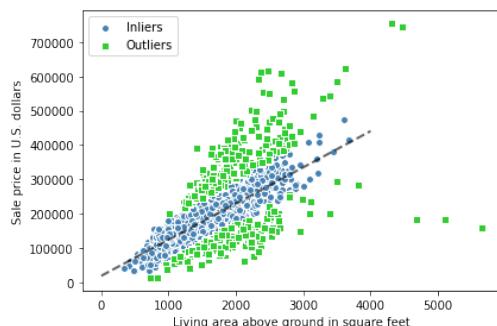
Мы использовали MAD, чтобы посчитать threshold для обозначения inliers и outliers. MAD для нашего датасета можно посчитать так:

```

def mean_absolute_deviation(data):
    return np.mean(np.abs(data - np.mean(data)))
print(mean_absolute_deviation(y))
# -> 58269.561754979375

```

Поэтому, если мы хотим обозначить как outliers меньшие точки, то нам надо выбрать значение residual_threshold больше чем MAD. Например, на этой картинке threshold равен 65,000:



Используя RANSAC, мы сократили потенциальный эффект выбросов, однако мы не знаем положительный ли эффект это окажет на предсказание новых данных. Поэтому нам нужны разные подходы для оценки regression модели, что является очень важной частью исследования. Вторая тема в этой части – вычисление **performance** linear regression моделей. Вместо того, чтобы использовать slr, сейчас мы будем работать со всеми пятью фичами и обучать multiple regression модель:

```

from sklearn.model_selection import train_test_split
target = 'SalePrice'
features = df.columns[df.columns != target]
X = df[features].values
y = df[target].values
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=123)
slr = LinearRegression()
slr.fit(X_train, y_train)
y_train_pred = slr.predict(X_train)
y_test_pred = slr.predict(X_test)

```

Т.к. наша модель использует multiple explanatory variables, то мы не можем визуализировать linear regression line (точнее гиперплоскость) на двумерном графике, но мы можем нарисовать residuals в зависимости от предсказанных значений. **Residual plots** – это часто используемое графическое

средство для исследования модели, оно помогает выявить нелинейность и выбросы, а также проверить случайно ли распределены ошибки. Мы сейчас нарисуем residual plot, где мы просто вычтем true target variables из наших predicted responses:

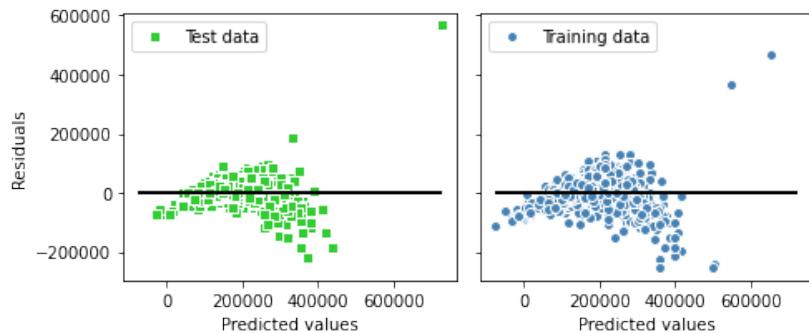
```
x_max = np.max(
    [np.max(y_train_pred), np.max(y_test_pred)])
x_min = np.min(
    [np.min(y_train_pred), np.min(y_test_pred)])

# sharey=True – в нашем случае не будет
# дублирования оси Oy
fig, (ax1, ax2) = plt.subplots(
    1, 2, figsize=(7, 3), sharey=True)

ax1.scatter(
    y_test_pred, y_test_pred - y_test,
    c='limegreen', marker='s', edgecolor='white',
    label='Test data')

ax2.scatter(
    y_train_pred, y_train_pred - y_train,
    c='steelblue', marker='o', edgecolor='white',
    label='Training data')
ax1.set_ylabel('Residuals')

for ax in (ax1, ax2):
    ax.set_xlabel('Predicted values')
    ax.legend(loc='upper left')
    ax.hlines(y=0, xmin=x_min-100, xmax=x_max+100,
              color='black', lw=2)
plt.tight_layout()
plt.show()
```



В случае идеального предсказания, residuals будут точно равны нулю, чего не достичь в реальных задачах. Однако для хорошей regression модели нужно ожидать, что ошибки будут случайно распределены и что residuals будут случайно нарисованы вокруг центральной линии. Если мы видим закономерность в residual plot, то это значит, что модель не может уловить какую-то explanatory information, которая утекла в residuals, это можно увидеть на нашем предыдущем графике. Увидеть outliers просто – это точки, у которых большое отклонение от центральной линии. Другая количественная мера нашей модели – **mean squared error (MSE)**, которую мы уже обсуждали раньше как функцию потерь. Мы используем версию MSE без $\frac{1}{2}$, которую использовали для упрощения производной в GD:

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

Мы можем использовать MSE для cross-validation и model selection как и для prediction accuracy в классификации. Как и classification accuracy, MSE нормализуется числом n , чтобы можно было сравнивать ошибку среди разных размеров n , например, в контексте learning curves. Посчитаем MSE для train и test сетов:

```
from sklearn.metrics import mean_squared_error
mse_train = mean_squared_error(y_train, y_train_pred)
mse_test = mean_squared_error(y_test, y_test_pred)
print(f'MSE train: {mse_train:.2f}') # 1497216245.85
print(f'MSE test: {mse_test:.2f}') # 1516565821.00
```

Мы видим, что MSE на train датасете больше чем на test, что означает небольшое переобучение. Показывать ошибку в начальной размерности (здесь доллары вместо их квадратов) может быть

интуитивно понятнее, поэтому можно выбрать корень из MSE – *root mean squared error* или **mean absolute error (MAE)**, которая подчёркивает неправильное предсказание немного меньше:

$$MAE = \frac{1}{n} \sum_{i=1}^n |y^{(i)} - \hat{y}^{(i)}|$$

Напишем аналогичный код:

```
from sklearn.metrics import mean_absolute_error
mae_train = mean_absolute_error(y_train, y_train_pred)
mae_test = mean_absolute_error(y_test, y_test_pred)
print(f'MAE train: {mae_train:.2f}') # 25983.03
print(f'MAE test: {mae_test:.2f}') # 24921.29
```

На основании MAE ошибки на тест сете можно сказать, что модель делать ошибку в \$25,000 в среднем. Когда мы используем MAE или MSE для сравнивания моделей, надо иметь в виду, что они неограничены в отличие, например, от classification accuracy. Другими словами, интерпретация MAE и MSE зависит от датасета и feature scaling. Например, модель, которая считает в K, а не просто в \$, будет давать ошибку меньше: $|\$500K - 550K| < |\$500,000 - 550,000|$. Поэтому иногда бывает полезно считать **coefficient of determination (R^2)**, который можно понять как стандартизированную версию MSE. Другими словами, R^2 – это доля response variance, которая поймана моделью. А вот и формула:

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

SSE – это просто сумма квадратов ошибок без нормализации:

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

А SST – это общая сумма квадратов или, другими словами, просто variance of the response:

$$SST = \sum_{i=1}^n (y^{(i)} - \mu_y)^2$$

Кратко покажем, что R^2 – это просто rescaled версия MSE:

$$R^2 = 1 - \frac{SSE}{SST} = 1 - \frac{\frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{n} \sum_{i=1}^n (y^{(i)} - \mu_y)^2} = 1 - \frac{MSE}{Var(y)}$$

Для train датасета R^2 лежит в отрезке $[0, 1]$, но может быть < 0 для test датасета. Отрицательное значение R^2 означает, что модель обучается на данных хуже, чем горизонтальная линия, которая представляет sample mean (это может быть из-за очень сильного переобучения или из-за того, что мы забыли масштабировать test датасет так же как и train датасет). Если $R^2 = 1$, то модель обучается идеально, т.е. $MSE = 0$. А вот и код:

```
from sklearn.metrics import r2_score
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)
print(f'R^2 train: {train_r2:.2f}, test_r2: {test_r2:.2f}')
# R^2 train: 0.77, 0.75
```

Значение 0.77 конечно не большое, но хорошее, учитывая маленько подмножество фич, но значение $0.75 < 0.77$ говорит о небольшом переобучении. Последняя тема в этой части – это **regularized methods** для регрессии. Регуляризация – это один из подходов борьбы с переобучением благодаря прибавлению дополнительной информации, иначе говоря сжатию параметров модели. Самые популярные подходы для regularized linear regression: **ridge regression, least absolute shrinkage and selection operator (LASSO) и elastic net**. Ridge regression – это L2 модель, где мы просто добавляем сумму квадратов весов к MSE:

$$L(w)_{Ridge} = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \|w\|_2^2$$

Здесь L2 слагаемое выглядит так:

$$\lambda \|w\|_2^2 = \lambda \sum_{j=1}^m w_j^2$$

Важно, что bias unit в сумме не участвует. Альтернативный подход, который приводит к разреженными моделям – LASSO. В зависимости от силы регуляризации некоторые веса могут стать равными нулю, что делает LASSO также полезной supervised feature selection техникой:

$$L(w)_{Lasso} = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \|w\|_1$$

Здесь L1 слагаемое определяется как сумма абсолютных магнитуд весов модели:

$$\lambda \|w\|_1 = \lambda \sum_{j=1}^m |w_j|$$

Однако есть ограничение в LASSO: он выбирает не больше n фич, если $m > n$, где n – число train примеров. Это может быть нежелательно в некоторых областях feature selection. Однако на практике это свойство LASSO часто является преимуществом т.к. помогает избежать насыщенных моделей. Насыщение модели происходит если число train примеров равно кол-ву фич, что является формой overparameterization. Как следствие модель идеально обучается на train данных, но в этом случае она является формой интерполяции, что приводит к плохому обобщению. Компромисс между ridge regression и LASSO – это elastic net, которая имеет L1, чтобы генерировать разреженность и L2, чтобы можно было выбрать больше n фич, если $m > n$:

$$L(w)_{Elastic\ Net} = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 + \lambda_2 \|w\|_2^2 + \lambda_1 \|w\|_1$$

Эти модели доступны в scikit-learn, их использование аналогично обычной модели за исключением того, что нужно указывать regularization strength, которое оптимизируется, например через k-fold cross-validation. А вот и код:

```
from sklearn.linear_model import Ridge
# alpha = gamma
ridge = Ridge(alpha=1.0)

from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1.0)

from sklearn.linear_model import ElasticNet
elinet = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

Реализация ElasticNet позволяет менять отношение L1 к L2, например, если мы ставим `l1_ratio = 1`, то это просто получится LASSO регрессия, почему это так можно посмотреть в документации.

9.5 Polynomial regression

В предыдущих частях мы предполагали только линейные отношения между x и y , но можно использовать polynomial regression, добавив полиномиальные члены:

$$y = w_1x + w_2x^2 + \cdots + w_dx^d + b$$

Хотя мы можем использовать пол. рег., чтобы получить нелинейную границу, это всё ещё multiple linear regression модель из-за линейных коэффициентов w . Рассмотрим добавление polynomial terms с помощью scikit-learn. Сейчас мы научимся использовать трансформер PolynomialFeatures, чтобы добавить квадратичный член:

```
# добавим квадратичный член
from sklearn.preprocessing import PolynomialFeatures
X = np.array([258.0, 270.0, 294.0, 320.0, 342.0,
            368.0, 396.0, 446.0, 480.0, 586.0]) \
    [:, np.newaxis]
y = np.array([236.4, 234.4, 252.8, 298.6, 314.2,
            342.2, 360.8, 368.0, 391.2, 390.8])
lr = LinearRegression()
pr = LinearRegression()
quadratic = PolynomialFeatures(degree=2)
X_quad = quadratic.fit_transform(X)

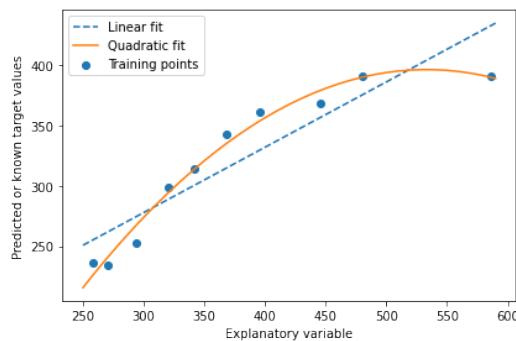
# простая linear regression model для сравнения
lr.fit(X, y)
X_fit = np.arange(250, 600, 10)[:, np.newaxis]
y_lin_fit = lr.predict(X_fit)
```

```

# обучим multiple regression на transformed features для
# polynomial regression
pr.fit(X_quad, y)
y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))

# нарисуем:
plt.scatter(X, y, label='Training points')
plt.plot(X_fit, y_lin_fit,
         label='Linear fit', linestyle='--')
plt.plot(X_fit, y_quad_fit,
         label='Quadratic fit')
plt.xlabel('Explanatory variable')
plt.ylabel('Predicted or known target values')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()

```



Как мы видим кривая полиномиальной модели намного лучше понимает отношение между x и y . Теперь посчитаем MSE и R^2 метрики:

```

y_lin_pred = lr.predict(X)
y_quad_pred = pr.predict(X_quad)
mse_lin = mean_squared_error(y, y_lin_pred)
mse_quad = mean_squared_error(y, y_quad_pred)
print(f'Training MSE linear: {mse_lin:.3f}',
      f', quadratic: {mse_quad:.3f}')
# Training MSE linear: 569.780, quadratic: 61.330

r2_lin = r2_score(y, y_lin_pred)
r2_quad = r2_score(y, y_quad_pred)
print(f'Training R^2 linear: {r2_lin:.3f}',
      f', quadratic: {r2_quad:.3f}')
# Training R^2 linear: 0.832, quadratic: 0.982

```

Теперь применим эту технику к нашему датасету, мы обучим модель с квадратичным и кубическим членами, в качестве x опять будет площадь. Мы начнём с того, что уберём три outliers с площадью $> 4,000$, которые мы видели на предыдущем графике, чтобы эти outliers не искали нашу регрессию:

```

X = df[['Gr Liv Area']].values
y = df['SalePrice'].values
X = X[(df['Gr Liv Area'] < 4000)]
y = y[(df['Gr Liv Area'] < 4000)]

```

Теперь обучим модели регрессии:

```

regr = LinearRegression()

quadratic = PolynomialFeatures(degree=2)
cubic = PolynomialFeatures(degree=3)
X_quad = quadratic.fit_transform(X)
X_cubic = cubic.fit_transform(X)

X_fit = np.arange(X.min()-1, X.max()+2, 1)[:, np.newaxis]
regr = regr.fit(X, y)
y_lin_fit = regr.predict(X_fit)
linear_r2 = r2_score(y, regr.predict(X))

regr = regr.fit(X_quad, y)
y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
quadratic_r2 = r2_score(y, regr.predict(X_quad))

regr = regr.fit(X_cubic, y)
y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))

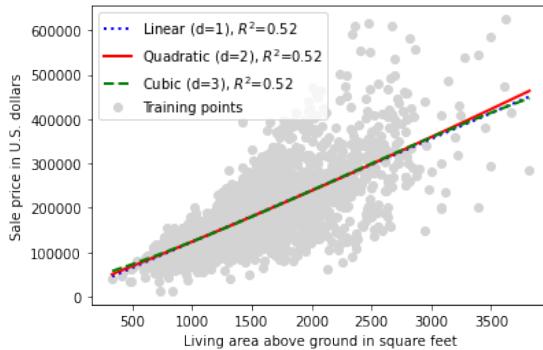
```

```

cubic_r2 = r2_score(y, regr.predict(X_cubic))

plt.scatter(X, y, label='Training points', color='lightgray')
plt.plot(X_fit, y_lin_fit,
         label=f'Linear (d=1), $R^2$={linear_r2:.2f} ,',
         color='blue', lw=2, linestyle=':')
plt.plot(X_fit, y_quad_fit,
         label=f'Quadratic (d=2), $R^2$={quadratic_r2:.2f} ,',
         color='red', lw=2, linestyle='--')
plt.plot(X_fit, y_cubic_fit,
         label=f'Cubic (d=3), $R^2$={cubic_r2:.2f} ,',
         color='green', lw=2, linestyle='---')
plt.xlabel('Living area above ground in square feet')
plt.ylabel('Sale price in U.S. dollars')
plt.legend(loc='upper left')
plt.show()

```



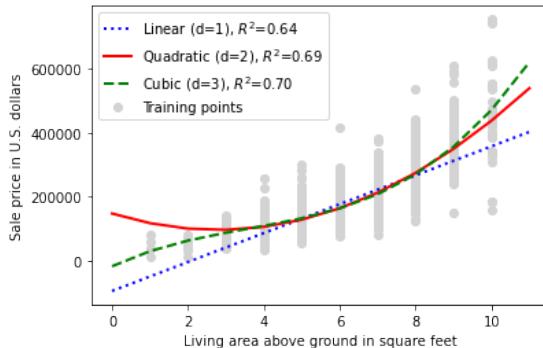
Как мы видим даже кубические фичи практически не имели эффекта, т.к. отношения судя по всему линейные. Давайте посмотрим на другую фичу Overall Qual:

```

X = df[['Overall Qual']].values
y = df['SalePrice'].values

```

Запустим код выше ещё раз и посмотрим на графики:



Мы видим, что полиномиальные члены лучше улавливают взаимосвязь x и y , однако нужно понимать, что с добавлением фич сильно растёт сложность модели, то есть шанс переобучения намного выше.

9.6 Dealing with nonlinear relationships using random forests

В этой части мы посмотрим на **random forest** regression, которая концептуально отличается от тех регрессий, которые мы уже прошли. Её можно понять как набор кусочно-линейных функций, в отличие от глобальной линейной или полиномиальной регрессий. Другими словами, через decision tree algorithm мы делим входное пространство на меньшие части, с которыми справиться проще. Сначала поговорим про **decision tree regression**. Плюс decision tree алгоритма в том, что он работает с произвольными фичами и не требует никакой трансформации фич, если мы имеем дело с нелинейными данными, т.к. дерево анализирует одну фичу за раз, а не их взвешанную комбинацию. Мы растим дерево, последовательно разбивая вершины пока не останутся листья или пока

не выполнен какой-то критерий. Мы вводили entropy как меру impurity, чтобы определить какое разбиение максимизирует IG, который для бинарного случая выглядит так:

$$IG(D_p, x_i) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

Напомню, что x_i – фича для разбиения, I – impurity функция, D – некое подмножество датасета. Наша цель – найти такое разбиение, которое максимизирует IG, т.е. разбиение, которое минимизирует impurity в детях. Мы обсуждали Gini impurity и entropy как меры impurity в классификации. Чтобы использовать decision tree для regression, нам нужна impurity метрика, которая подходит для непрерывных переменных, поэтому определим impurity вершины t как MSE:

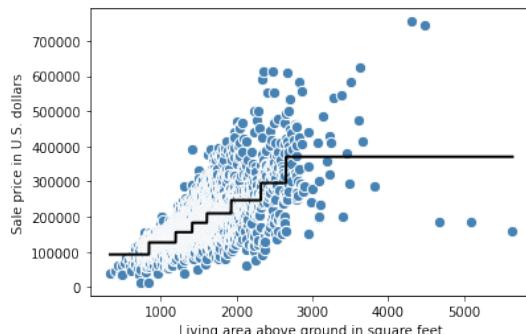
$$I(t) = MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} (y^{(i)} - \hat{y}_t)^2$$

Здесь \hat{y}_t – предсказанное значение (sample mean):

$$\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} y^{(i)}$$

В контексте decision tree regression, MSE часто называют **within-node variance**, из-за чего splitting criterion лучше известен как **variance reduction**. Чтобы посмотреть как выглядит граница, будем использовать DecisionTreeRegressor:

```
from sklearn.tree import DecisionTreeRegressor
X = df[['Gr Liv Area']].values
y = df['SalePrice'].values
tree = DecisionTreeRegressor(max_depth=3)
tree.fit(X, y)
# иначе будет не пойти что
# будет много линий в разные точки
sort_idx = X.flatten().argsort()
lin_regplot(X[sort_idx], y[sort_idx], tree)
plt.xlabel('Living area above ground in square feet')
plt.ylabel('Sale price in U.S. dollars')
plt.show()
```



Мы видим, что дерево хорошо улавливает общий тренд данных, можно представить, что в нелинейных данных будет также хороший результат. Недостаток данной модели в том, что она не улавливает непрерывность и дифференцируемость желаемой функции. Также важно правильно выбрать глубину дерева, иначе можно либо недообучиться, либо сильно переобучиться. Можно сделать эксперимент, поменяв глубину дерева на большую или фичу на Overall Qual. Сейчас мы пройдём более надёжный способ обучения regression trees – **random forest regression**. У леса ошибка обобщения лучше благодаря случайности, что позволяет уменьшить переобучение, также он не требует масштабирования и менее чувствителен к выбросам. Алгоритм очень похож на тот, который мы уже проходили, только здесь мы используем MSE критерий, а также ответ считается как среднее ответов всех деревьев. Обучим лес на всех фичах и посмотрим на результат:

```
target = 'SalePrice'
features = df.columns[df.columns != target]
X = df[features].values
y = df[target].values
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=123)

from sklearn.ensemble import RandomForestRegressor
```

```

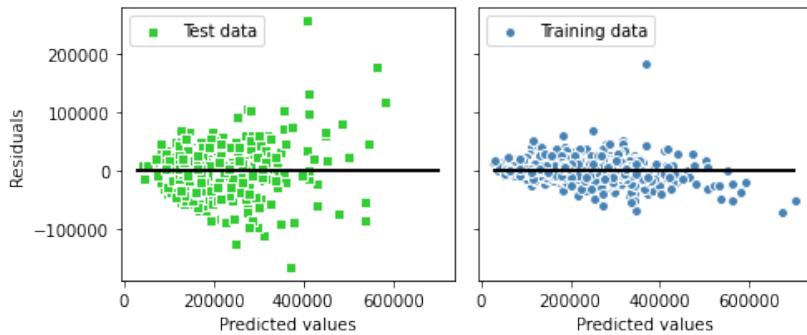
forest = RandomForestRegressor(
    n_estimators=1000,
    criterion='squared_error',
    random_state=1, n_jobs=-1)
forest.fit(X_train, y_train)
y_train_pred = forest.predict(X_train)
y_test_pred = forest.predict(X_test)

mae_train = mean_absolute_error(y_train, y_train_pred)
mae_test = mean_absolute_error(y_test, y_test_pred)
print(f'MAE train: {mae_train:.2f}') # 8305.18
print(f'MAE test: {mae_test:.2f}') # 20821.77

r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)
print(f'R^2 train: {r2_train:.2f}') # 0.98
print(f'R^2 test: {r2_test:.2f}') # 0.85

```

К сожалению, лес всё равно переобучился; хотя лин. рег. на таком же датасете меньше переобучилась, она дала результат хуже. Наконец, посмотрим на residuals наших предсказаний, код для такого графика мы недавно уже писали, здесь он один в один такой же:



Выбросы по оси y опять же показывают наличие переобучения. Распределение вокруг нуля также не кажется абсолютно случайным, то есть модель не уловила всю exploratory information, однако видно явное улучшение по сравнению с тем, что мы рисовали раньше в этой главе. В идеале наша ошибка должна быть абсолютно случайной, то есть ошибка не должна соотноситься с какой-либо информацией в explanatory variables. Если мы видим шаблон в ошибках, например на residual plot, это значит, что residual plots содержит информацию о предсказаниях, причина в том, что explanatory information утекает в residuals. К сожалению, нет общего рецепта, можно трансформировать переменные, поменять гиперпараметры, выбрать сложность модели, убрать выбросы или включить доп. переменные. В следующей главе мы пройдём другую интересную под область ml – unsupervised learning, а также как использовать cluster analysis, чтобы искать скрытые структуры в данных в отсутствии target переменных.

10 Clustering Analysis

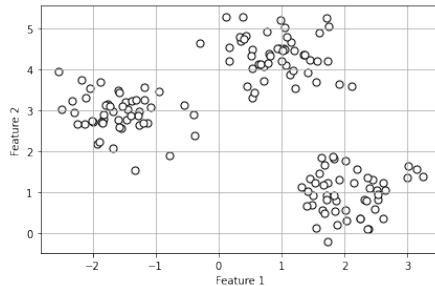
В этой главе мы будем исследовать cluster analysis, категорию unsupervised learning техники, которая позволяет искать скрытые структуры в данных, где мы заранее не знаем правильного ответа. Цель clustering – найти естественную группировку данных, чтобы элементы в одних кластерах были более похожи друг на друга нежели на элементы в других кластерах. В этой главе мы пройдём: поиск centers of similarity с помощью алгоритма **k-means**, использование bottom-up подхода для построения hierarchical clustering trees, идентификация произвольных форм объектов с помощью density-based clustering подхода.

10.1 Grouping objects by similarity using k-means

В этой части мы пройдём один из самых популярных алгоритмов кластеризации – k-means. Примеры clustering в реальной жизни: группировка документов, музыки, фильмов или покупателей с похожими интересами для системы рекомендаций. Поговорим про k-means кластеризацию с помощью scikit-learn. Как мы увидим, этот алгоритм очень просто реализовать, при этом он также вычислительно очень эффективен; этот алгоритм принадлежит категории **prototype-based clustering**. Другие две категории **hierarchical** и **density-based clustering** мы обсудим позже. Prototype-based кластеризация означает, что каждый кластер представлен его прототипом, который часто или **centroid** (*average*) похожих точек с непрерывными фичами, или **medoid** (*representative*

или точка, которая минимизирует расстояние до всех других точек в кластере) в случае categorical фич. Этот алгоритм хорошо находит кластеры сферической формы, недостаток в том, что нужно указывать кол-во кластеров k . Позже в этой части мы обсудим **elbow** метод и **silhouette plots** – полезные техники для определения качества кластеризации, это поможет выбрать оптимальное k . Хотя k-means можно применить для больших размерностей, мы посмотрим на двумерный случай в целях визуализации:

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=150,
                   n_features=2,
                   centers=3,
                   cluster_std=0.5,
                   shuffle=True,
                   random_state=0)
import matplotlib.pyplot as plt
plt.scatter(X[:, 0], X[:, 1], c='white',
            marker='o', edgecolor='black',
            s=50)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid()
plt.tight_layout()
plt.show()
```



На датасете мы видим 150 примеров, которые сгруппированы в 3 региона большой плотности. Наша цель – сгруппировать примеры на основе похожести их фич. Напишем 4 шага k-means алгоритма:

1. Случайно выберем k центроидов из примеров как начальные центры кластеров.
2. Отнесём каждый пример к ближайшему центроиду, $\mu^{(j)}$, $j \in \{1, \dots, k\}$.
3. Сдвинем центроиды в центры примеров, которые были присвоены каждому из них.
4. Повторяем шаги 2 и 3 пока кластеры не перестанут двигаться, или пока не пройдёт макс. число итераций, или пока изменение не будет меньше некоторого user-defined числа.

А как мы меряем похожесть объектов? Можно объявить похожесть как противоположность расстоянию, часто используемое расстояние в случае непрерывных фич – **squared Euclidean distance** между точками x и y в m -мерном пространстве:

$$d(x, y)^2 = \sum_{j=1}^m (x_j - y_j)^2 = \|x - y\|_2^2$$

Теперь, на основании этой метрики, мы можем описать k-means алгоритм как простую задачу оптимизации, итеративный подход минимизации within-cluster **sum of squared errors (SSE)** (или **cluster inertia**):

$$SSE = \sum_{i=1}^n \sum_{j=1}^k w^{(i,j)} \|x^{(i)} - \mu^{(j)}\|_2^2$$

Здесь μ_j – это representative point (centroid) для кластера j . Функция $w^{(i,j)}$:

$$w^{(i,j)} = \begin{cases} 1, & \text{if } x^{(i)} \in j \\ 0, & \text{otherwise} \end{cases}$$

Применим k-means к нашему датасету с помощью класса KMeans:

```

from sklearn.cluster import KMeans
km = KMeans(n_clusters=3, init='random',
             n_init=10, max_iter=300,
             tol=1e-04, random_state=0)
y_km = km.fit_predict(X)
# массив y_km содержит номер кластера,
# к которому принадлежит каждый пример

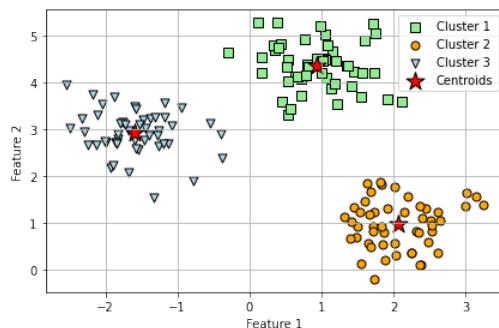
```

С помощью параметра `n_init=10` мы сказали, что хотим запустить k-means независимо 10 раз с разными случайными центроидами, чтобы выбрать лучшую модель с самым маленьким значением SSE. Эта реализация заканчивает свою работу, если алгоритм уже сопротивляется, но максимальное число итераций не было выполнено. Однако иногда k-means может не сопротивляться, что будет довольно вычислительно сложно при больших значениях `max_iter`. Один из способов с этим бороться – выбрать значение побольше для `tol` – это параметр, который контролирует величину изменения within-cluster SSE, чтобы объявить о сходимости. Проблема с k-means в том, что один или более кластеров могут быть пустыми. Этой проблемы нет для k-medoids или fuzzy C-means, алгоритма, который мы пройдём далее. Если кластер пустой, то алгоритм будет искать пример, который находится дальше всего от центроида этого пустого кластера. Затем он переприсвоит центроид в эту самую дальнюю точку. Когда мы применяем k-means для реальных данных и используем Евклидову метрику, то нужно применять z-score стандартизацию или min-max scaling. Визуализируем кластеры и их центроиды, которые находятся в атрибуте `cluster_centers_`:

```

plt.scatter(X[y_km == 0, 0], X[y_km == 0, 1],
            s=50, c='lightgreen', marker='s',
            edgecolor='black', label='Cluster 1')
plt.scatter(X[y_km == 1, 0], X[y_km == 1, 1],
            s=50, c='orange', marker='o',
            edgecolor='black', label='Cluster 2')
plt.scatter(X[y_km == 2, 0], X[y_km == 2, 1],
            s=50, c='lightblue', marker='v',
            edgecolor='black', label='Cluster 3')
plt.scatter(km.cluster_centers_[:, 0], # 1 столбец
            km.cluster_centers_[:, 1], # 2 столбец
            s=250, marker='*', c='red',
            edgecolor='black', label='Centroids')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
# scatterpoints – число marker points
plt.legend(scatterpoints=1)
plt.grid()
plt.tight_layout()
plt.show()

```



Хотя k-means хорошо показал себя на этом игрушечном примере, главный недостаток в том, что нужно выбирать значение k . Это число не всегда очевидно в реальных задачах, особенно если мы работаем с большими размерностями, которые тяжело визуализировать. Другие свойства k-means в том, что кластеры не перекрываются и в том, что они не hierarchical, а также в каждом кластере должен быть хотя бы один элемент. Позже в этой главе, мы узнаем другие типы clustering алгоритмов – hierarchical и density-based clustering. Ни один из этих типов алгоритмов не требует задания кол-ва кластеров заранее или предположения о их сферической структуре. Далее мы посмотрим на популярный вариант классического алгоритма k-means – **k-means++**. Хотя он и не исправляет те недостатки, которые мы обсудили, он может значительно улучшить результат кластеризации благодаря более умному начальному выбору центров кластеров. Плохой начальный выбор может также привести к очень медленной сходимости. Один из способов решения – это запустить алгоритм несколько раз и выбрать лучшую модель на основании SSE. Другая стратегия – располагать начальные центроиды далеко друг от друга через k-means++ алгоритм, инициализацию в k-means++ можно записать в виде 6 шагов:

1. Заведём пустой set M , чтобы хранить k выбранных центроидов.
2. Случайно выберем первый центроид $\mu^{(j)}$ из входных примеров и присвоим его M .
3. Для каждого примера $x^{(i)}$, который не в M , найдём минимальный квадрат расстояния до любого из центроидов из M , $d(x^{(i)}, M)^2$.
4. Чтобы случайно выбрать следующий центроид $\mu^{(p)}$, будем использовать взвешенное распределение вероятностей, которое равно

$$\frac{d(\mu^{(p)}, M)^2}{\sum_i d(x^{(i)}, M)^2}$$

Например, мы собираем все точки в массиве и выбираем взвешенную случайную выборку так, что, чем больше квадрат расстояния, тем больше вероятность, что точка будет выбрана в качестве следующего центроида.

5. Повторим шаги 3 и 4 пока не будет выбрано k центроидов.
6. Дальше выполним классический k-means алгоритм.

Чтобы использовать k-means++ в классе KMeans, надо просто поставить init параметр в k-means++. На самом деле k-means++ – это значение по умолчанию у параметра init, которое строго рекомендовано на практике. Теперь обсудим такую тему: hard или soft кластеризация. **Hard clustering** описывает семейство алгоритмов, где каждый пример из датасета присваивается ровно к одному кластеру, ровно как и в k-means(++) алгоритмах, которые мы обсудили. А вот в **soft clustering** (иногда их называют **fuzzy clustering**) алгоритмах каждый пример присваивается к одному или более кластерам. Популярный пример soft clustering – это **fuzzy C-means (FCM)** алгоритм (иногда его называют **soft k-means** или **fuzzy k-means**). Процедура FCM очень похожа на k-means, однако мы заменяем hard cluster присваивание на вероятности принадлежности к кластерам для каждой точки. В k-means мы можем выразить cluster membership примера x с помощью sparse вектора бинарных значений:

$$\begin{cases} x \in \mu^{(1)} \rightarrow w^{(i,j)} = 0 \\ x \in \mu^{(2)} \rightarrow w^{(i,j)} = 1 \\ x \in \mu^{(3)} \rightarrow w^{(i,j)} = 0 \end{cases}$$

А вот в FCM такой вектор может выглядеть так:

$$\begin{cases} x \in \mu^{(1)} \rightarrow w^{(i,j)} = 0.10 \\ x \in \mu^{(2)} \rightarrow w^{(i,j)} = 0.85 \\ x \in \mu^{(3)} \rightarrow w^{(i,j)} = 0.05 \end{cases}$$

Запишем 4 ключевых шага для FCM:

1. Определим число центроидов k и случайно присвоим cluster memberships для каждой точки.
2. Посчитаем cluster centroids $\mu^{(j)}$, $j \in \{1, \dots, k\}$.
3. Обновим cluster memberships для каждой точки.
4. Повторим шаги 2 и 3 пока membership coefficients не перестанут меняться, или пока user-defined tolerance (или макс. число итераций) не будет достигнуто.

Objective function для FCM J_m очень похожа на within-cluster SSE:

$$J_m = \sum_{i=1}^n \sum_{j=1}^k w^{(i,j)m} \|x^{(i)} - \mu^{(j)}\|_2^2$$

Однако заметим, что membership indicator $w^{(i,j)}$ – это не бинарное значение, а веc. значение вероятности $w^{(i,j)} \in [0, 1]$. Также мы добавили экспоненту m , любое число ≥ 1 (обычно $m = 2$), её называют **fuzziness coefficient** (или просто **fuzzifier**), который контролирует степень *fuzziness*. Большее значение m приводит к меньшим cluster membership значениям $w^{(i,j)}$, что приводит к более размытым кластерам. Cluster membership probability можно посчитать так:

$$w^{(i,j)} = \left[\sum_{c=1}^k \left(\frac{\|x^{(i)} - \mu^{(j)}\|_2}{\|x^{(i)} - \mu^{(c)}\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

Центр кластера $\mu^{(j)}$ считается как среднее значение всех примеров, взвешенных по степени принадлежности каждого примера к этому кластеру:

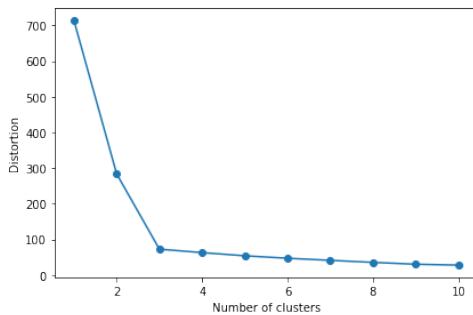
$$\mu^{(j)} = \frac{\sum_{i=1}^n w^{(i,j)^m} x^{(i)}}{\sum_{i=1}^n w^{(i,j)^m}}$$

Только посмотрев на формулу для cluster memberships, можно сказать, что каждая итерация в FCM намного дороже, чем итерация в k-means. С другой стороны, FCM обычно требует меньше итераций, чтобы достичь сходимости. Однако было показано на практике, что и k-means, и FCM дают очень похожие clustering outputs. К сожалению, FCM алгоритм не реализован в scikit-learn, однако можно найти реализацию в пакете *scikit-fuzzy*. Теперь пройдём **elbow method** для поиска оптимального числа кластеров k . Главный вызов unsupervised learning в том, что мы не знаем точного ответа или метод, чтобы применить техники из главы 6 для вычисления эффективности supervised модели. Поэтому, чтобы оценить качество кластеризации, мы должны использовать intrinsic metrics, такие как within-cluster SSE (distortion) для сравнения разных k-means clustering моделей. Удобно, что не надо явно считать SSE, так как он уже доступен через `inertia_` атрибут после обучения KMeans модели:

```
print(f'Distortion: {km.inertia_:.2f}')
# Distortion: 72.48
```

На основании SSE мы можем использовать graphical tool **elbow method**, чтобы оценить оптимальное число кластеров k . Мы можем сказать, что если k растёт, то distortion уменьшается, т.к. примеры будут ближе к центроидам, к которым они принадлежат. Идея elbow method в том, чтобы найти значение k , где distortion начинает расти быстрее всего; это станет яснее, если мы нарисуем distortion для разных значений k :

```
distortions = []
for i in range(1, 11):
    km = KMeans(n_clusters=i,
                 init='k-means++',
                 n_init=10,
                 max_iter=300,
                 random_state=0)
    km.fit(X)
    distortions.append(km.inertia_)
plt.plot(range(1,11), distortions, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.tight_layout()
plt.show()
```



На графике видно, что *elbow* находится в точке $k = 3$, что на самом деле является лучшим значением. Теперь пройдём количественную оценку качества кластеризации с помощью **silhouette plots**. Другая intrinsic metric для оценки качества кластеризации – **silhouette analysis**, который также можно применить и к другим алгоритмам (их мы обсудим далее). Silhouette analysis можно использовать как graphical tool, чтобы рисовать степень того, насколько плотно сгруппированы примеры в кластерах. Чтобы посчитать **silhouette coefficient** одного примера, нужно применить следующие 3 шага:

1. Посчитаем **cluster cohesion** $a^{(i)}$ как среднее расстояние между примером $x^{(i)}$ и всеми другими примерами в этом же кластере.
2. Посчитаем **cluster separation** $b^{(i)}$ от след. ближайшего кластера как среднее расстояние между примером $x^{(i)}$ и всеми примерами в этом близк. кластере.
3. Посчитаем silhouette $s^{(i)}$:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\{b^{(i)}, a^{(i)}\}}$$

Silhouette coefficient лежит от -1 до 1 . На основе формулы выше, мы видим, что коэффициент равен 0 , если $b^{(i)} = a^{(i)}$. Кроме того, мы приближаемся к идеальному коэффициенту равному 1 , если $b^{(i)} \gg a^{(i)}$. Этот коэффициент доступен как silhouette_samples, также для удобства можно импортировать функцию silhouette_scores. Эта функция считает средний silhouette coefficient по всем примерам, что эквивалентно `numpy.mean(silhouette_samples(...))`. Следующим кодом мы нарисуем график silhouette coefficients для k-means кластеризации при $k = 3$:

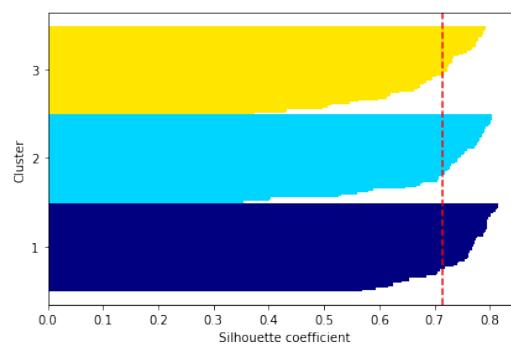
```

km = KMeans(n_clusters=3,
             init='k-means++',
             n_init=10,
             max_iter=300,
             tol=1e-04,
             random_state=0)
y_km = km.fit_predict(X)

import numpy as np
from matplotlib import cm
from sklearn.metrics import silhouette_samples
cluster_labels = np.unique(y_km)
n_clusters = cluster_labels.shape[0]
silhouette_vals = silhouette_samples(
    X, y_km, metric='euclidean')
y_ax_lower, y_ax_upper = 0, 0
yticks = []
for i, c in enumerate(cluster_labels):
    c_silhouette_vals = silhouette_vals[y_km == c]
    c_silhouette_vals.sort()
    y_ax_upper += len(c_silhouette_vals)
    color = cm.jet(float(i) / n_clusters)
    plt.barh(range(y_ax_lower, y_ax_upper),
            c_silhouette_vals,
            height=1.0,
            edgecolor='none',
            color=color)
    yticks.append((y_ax_lower + y_ax_upper) / 2.)
    y_ax_lower += len(c_silhouette_vals)

silhouette_avg = np.mean(silhouette_vals)
plt.axvline(silhouette_avg,
            color='red',
            linestyle='—')
plt.yticks(yticks, cluster_labels + 1)
plt.ylabel('Cluster')
plt.xlabel('Silhouette coefficient')
plt.tight_layout()
plt.show()

```



Для лучшего понимания приведу код с использованием функции silhouette_scores, а также пояснение того, что такое `cm.jet`:

```

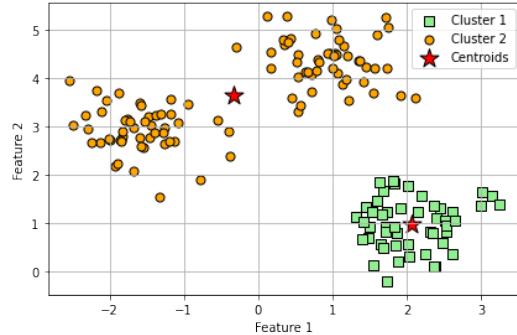
for i in range(100):
    k = float(i)/100
    plt.plot([0, 1], [k, k], color=cm.jet(k))
plt.show()

from sklearn.metrics import silhouette_score
print(f"Score: {silhouette_score(X, y_km, metric='euclidean'):.2f}")
# Score: 0.71

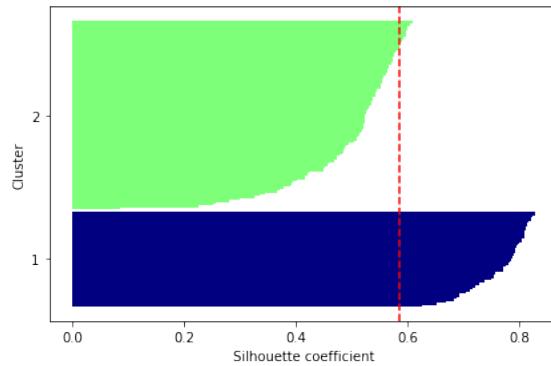
```

В целях экономии места картинку вставлять не буду, но на ней должны быть все цвета от красного к синему, сверху вниз. Посмотрев на график, мы можем быстро узнать размеры разных кластеров,

а также найти кластеры, которые содержат outliers. На нашем графике мы видим, что значения далеки от нуля, а также они находятся примерно на одном расстоянии от average silhouette score. Чтобы увидеть silhouette plot для относительно плохой кластеризации, поставим нашему алгоритму 2 центроида:



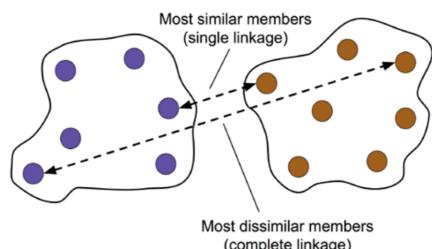
Код для этой картинки мы уже писали, надо только кол-во кластеров поменять на два вместо трёх. Помним, что обычно у нас нет такой роскоши визуализировать датасет на плоскости, поэтому теперь опять нарисуем silhouette plot:



Код мы также уже писали, в нём даже ничего не надо менять. Как мы видим, силуэты имеют явно разные длины и широты, так что это признак плохой или как минимум неоптимальной кластеризации. Теперь, после того, как мы разобрались в работе clustering, в следующей части мы пройдём hierarchical clustering как альтернативный подход к k-means.

10.2 Organizing clusters as a hierarchical tree

В этой части мы посмотрим на альтернативный подход к prototype-based clustering: **hierarchical clustering**. Один из плюсов этого алгоритма в том, что он позволяет рисовать **dendrograms** (визуализации бинарной иерархической кластеризации), которые могут помочь с интерпретацией результатов с помощью создания полезных taxonomies. Ещё один плюс в том, что не надо выбирать значение k заранее. Два главных подхода к hierarchical clustering – это **agglomerative** и **divisive**. В divisive кластеризации мы начинаем с одного кластера – весь датасет, а затем делим его на меньшие, пока каждый кластер не будет состоять ровно из одного примера. В этой части мы будем работать с agglomerative кластеризацией, она имеет обратный подход. Мы начинаем с индивидуальных кластеров-примеров, а затем уже сливаем близайшие пары кластеров, пока не останется ровно один. Начнём с группировки кластеров по **bottom-up** принципу. Два стандартных алгоритма agglomerative кластеризации – это **single linkage** и **complete linkage**. В первом мы считаем расстояния между самыми **похожими** членами для каждой пары кластеров и сливаем те два кластера, у которых расстояние между самыми похожими примерами самое маленькое. Во втором подходе мы делаем тоже самое, только сравниваем самые **непохожие** члены, чтобы сделать слияние. Для лучшего понимания см. картинку:



Какие ещё типы linkages существуют? Другие часто используемые алгоритмы для agglomerative класт. включают *average linkage* и *Ward's linkage*. В первом мы сливаем на основе minimum average distances между всеми группами членов в двух кластерах. Во втором сливаются те два кластера, которые приводят к минимальному увеличению общего within-cluster SSE. В этой части мы рассмотрим agglomerative кластеризацию с complete linkage подходом. Hierarchical complete linkage clustering – это итеративная процедура, которую можно записать в виде 5 шагов:

1. Посчитаем попарную distance matrix всех примеров.
2. Представим каждую точку как отдельный кластер.
3. Сольём два ближайших кластера на основании расстояний между самыми dissimilar членами.
4. Обновим cluster linkage матрицу.
5. Повторим шаги 3-4 пока не останется лишь один кластер.

Далее мы будем считать distance matrix, однако сначала сделаем случайные данные:

```
import pandas as pd
import numpy as np
np.random.seed(123)
variables = [ 'X' , 'Y' , 'Z' ]
labels = [ 'ID_0' , 'ID_1' , 'ID_2' , 'ID_3' , 'ID_4' ]
X = np.random_sample([5 , 3])*10
df = pd.DataFrame(X, columns=variables , index=labels)
print(df)
# Выход:
#          X      Y      Z
# ID_0  6.964692  2.861393  2.268515
# ID_1  5.513148  7.194690  4.231065
# ID_2  9.807642  6.848297  4.809319
# ID_3  3.921175  3.431780  7.290497
# ID_4  4.385722  0.596779  3.980443
```

Теперь поговорим про выполнение hierarchical clustering по матрице расстояний. Чтобы посчитать distance matrix на вход этому алгоритму, напишем следующий код:

```
from scipy.spatial.distance import pdist , squareform
# pdist считает попарные расстояния
# squareform просто делает симметричную матрицу
row_dist = pd.DataFrame(squareform(
    pdist(df , metric='euclidean')) ,
    columns=labels , index=labels)
print(row_dist)
# Выход:
#          ID_0      ID_1      ID_2      ID_3      ID_4
# ID_0  0.000000  4.973534  5.516653  5.899885  3.835396
# ID_1  4.973534  0.000000  4.347073  5.104311  6.698233
# ID_2  5.516653  4.347073  0.000000  7.244262  8.316594
# ID_3  5.899885  5.104311  7.244262  0.000000  4.382864
# ID_4  3.835396  6.698233  8.316594  4.382864  0.000000
```

Функция pdist вернула так называемую condensed матрицу. Дальше мы применим complete linkage agglomeration к нашим кластерам, мы будем использовать функцию *linkage*, которая вернёт **linkage matrix**. Однако перед тем, как её вызывать, посмотрим на её документацию:

```
from scipy.cluster.hierarchy import linkage
help(linkage)
```

Вывод конечно копировать я не буду, он очень большой, посмотрите его сами. На основе этого вывода мы понимаем, что надо использовать condensed distance matrix (pdist функция) как входной параметр. С другой стороны можно дать изначальный массив на вход, указав метрику 'euclidean'. Перечислим правильные варианты:

```
row_clusters = linkage(pdist(df , metric='euclidean') ,
                      method='complete')
# или с помощью design матрицы:
row_clusters = linkage(df.values ,
                      method='complete' ,
                      metric='euclidean')
```

Чтобы ближе посмотреть на результаты, мы можем конвертировать его в pandas DataFrame:

```
print(pd.DataFrame(row_clusters ,
columns=[ 'row_label_1' ,
'row_label_2' ,
```

```

        'distance',
        'no. of items in clust.'],
index=[f'cluster {(i + 1)}' for i in
       range(row_clusters.shape[0])]))
# Вывод:
#      row label 1  row label 2  distance  no. of items in clust.
# cluster 1          0.0          4.0  3.835396                  2.0
# cluster 2          1.0          2.0  4.347073                  2.0
# cluster 3          3.0          5.0  5.899885                  3.0
# cluster 4          6.0          7.0  8.316594                  5.0

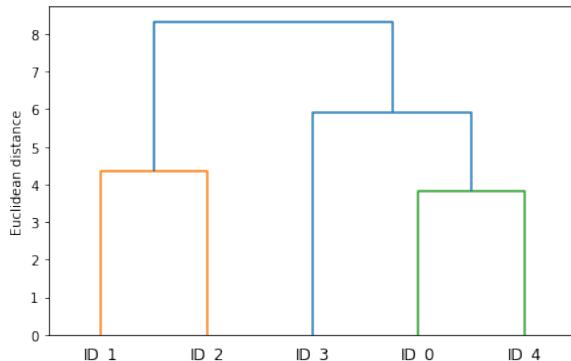
```

Как мы видим, linkage matrix состоит из нескольких рядов, где каждый ряд представляет одно слияние. Первая и вторая колонка обозначают самые непохожие члены в каждом кластере, третья колонка содержит расстояние между ними. Последняя колонка содержит число членов в каждом кластере. После того, как мы посчитали linkage matrix, мы можем визуализировать результат в форме dendrogram:

```

from scipy.cluster.hierarchy import dendrogram
# from scipy.cluster.hierarchy import set_link_color_palette
# set_link_color_palette(['black'])
row_dendr = dendrogram(
    row_clusters,
    labels=labels,
    # color_threshold=np.inf
)
plt.tight_layout()
plt.ylabel('Euclidean distance')
plt.show()

```



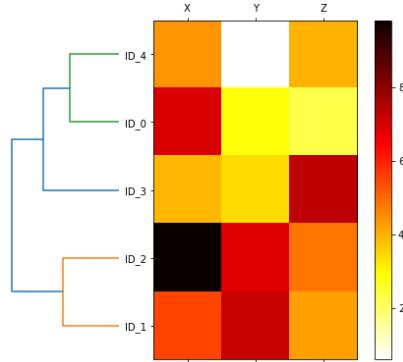
Ветки нашей дендрограммы цветные, чтобы сделать её просто чёрной, нужно убрать комментарии с трёх строк. Цветовая схема получена из списка цветов Matplotlib, которые циклически изменяются для пороговых значений расстояния в дендрограмме. Теперь поговорим про прикрепление dendrograms к heat map. На практике обычно используют дендрограммы в комбинации с **heat map**, которая позволяет кодировать значения в цвета. Прикрепить дендр. не очень просто, поэтому пройдём по шагам:

```

# facecolor – цвет фона
fig = plt.figure(figsize=(8, 8), facecolor='white')
# x u y axis positions, width, height of the dendrogram
axd = fig.add_axes([0.09, 0.1, 0.2, 0.6])
# orientation='left' – поворот против часовой стрелки
row_dendr = dendrogram(row_clusters,
                       orientation='left')
# переупорядочим датасет в порядке [4, 0, 3, 2, 1]
df_rowclust = df.iloc[row_dendr['leaves'][::-1]]
axm = fig.add_axes([0.23, 0.1, 0.6, 0.6])
cax = axm.matshow(df_rowclust,
                   interpolation='nearest',
                   cmap='hot_r')
axd.set_xticks([])
axd.set_yticks([])
for i in axd.spines.values():
    i.set_visible(False)
fig.colorbar(cax)
axm.set_xticklabels([''] + list(df_rowclust.columns))
axm.set_yticklabels([''] + list(df_rowclust.index))
plt.show()

```

Такая комбинация heat map и дендрограммы даёт лучшее понимание наших данных и кластеризации:



Поговорим про то, как работать с agglomerative clustering в scikit-learn. В scikit-learn есть реализация AgglomerativeClustering, которая позволяет выбрать число кластеров, это полезно, если мы хотим ограничить hierarchical cluster tree:

```
from sklearn.cluster import AgglomerativeClustering
ac = AgglomerativeClustering(n_clusters=3,
                             affinity='euclidean',
                             linkage='complete')
labels = ac.fit_predict(X)
print(f'Cluster labels: {labels}')
# Cluster labels: [1 0 0 2 1]
```

Примеры ID_0 и ID_4 были присвоены в первый кластер, ID_1 и ID_2 во второй, а вот ID_3 был присвоен в свой отдельный кластер. Заметим, что результаты сходятся с теми, что мы видим на дендрограмме. Также мы видим, что пример ID_3 более близок к 0 и 4 примерам, а не к 1 и 2, что не видно из просто scikit-learn's clustering результатов. Перезапустим код с `n_clusters=2`, результат такой, как и ожидали: [0 1 1 0 0].

10.3 Locating regions of high density via DBSCAN

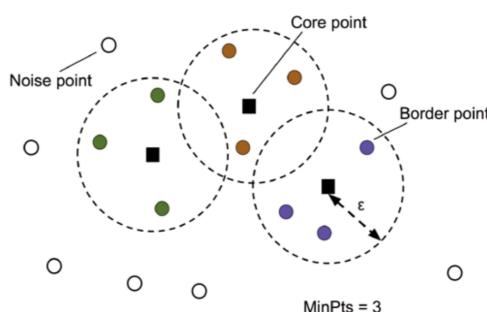
Пройдём ещё один подход кластеризации – **density-based spatial clustering of applications with noise (DBSCAN)**, который не делает предположений о сферической форме как в k-means, а также ему не нужна точка отсечения как в иерархиях. Как видно из названия, эта техника присваивает кластеры на основе плотности регионов точек. В DBSCAN понятие плотности определяется как число точек в пределах заданного радиуса ϵ . В соответствии DBSCAN алгоритму, special label присваивается каждому примеру, используя следующие критерии:

- Точка считается **core point**, если как минимум $MinPts$ соседних точек попадают в радиус ϵ .
- **Border point** – это точка, которая имеет меньше точек, чем $MinPts$ в радиусе ϵ , но лежит в радиусе ϵ от core point.
- Остальные точки называются **noise points**.

После распределения точек на core, border и noise, DBSCAN алгоритм можно записать в виде 2 шагов:

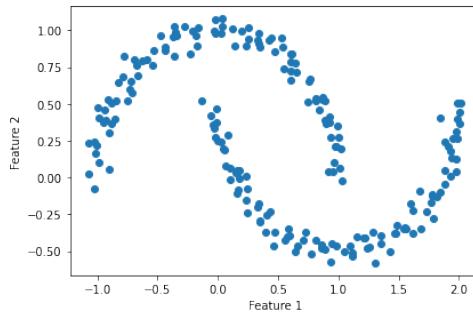
1. Сделаем отдельный кластер для каждой *core point* или для каждой связанной группы из *core points* (две *core point* связаны, если они на расстоянии меньше ϵ).
2. Присвоим каждую *border point* к кластеру соответствующей *core point*.

Для лучшего понимания см. картинку:



Одно из главных преимуществ при использовании DBSCAN в том, что он не предполагает сферическую форму кластеров как в k-means, также он отличается от тех алгоритмов, которые мы уже прошли, тем, что он не обязательно присваивает каждую точку к какому-то кластеру, то есть он может убрать шум. Создадим новый датасет из *half-moon-shaped* структур, чтобы сравнить k-means, hierarchical clustering и DBSCAN:

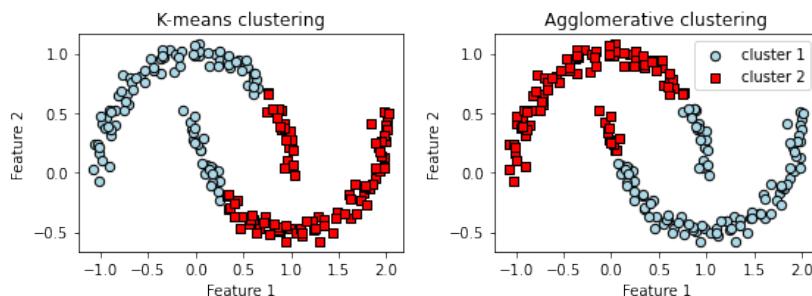
```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200,
                   noise=0.05,
                   random_state=0)
plt.scatter(X[:, 0], X[:, 1])
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.tight_layout()
plt.show()
```



Начнём с k-means и complete linkage кластеризаций, чтобы посмотреть, может ли кто-то из них хорошо решить эту задачу:

```
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
km = KMeans(n_clusters=2, random_state=0)
y_km = km.fit_predict(X)
ax1.scatter(X[y_km == 0, 0], X[y_km == 0, 1],
            c='lightblue', edgecolor='black',
            marker='o', s=40, label='cluster 1')
ax1.scatter(X[y_km == 1, 0], X[y_km == 1, 1],
            c='red', edgecolor='black',
            marker='s', s=40, label='cluster 2')
ax1.set_title('K-means clustering')
ax1.set_xlabel('Feature 1')
ax1.set_ylabel('Feature 2')

ac = AgglomerativeClustering(n_clusters=2,
                             affinity='euclidean',
                             linkage='complete')
y_ac = ac.fit_predict(X)
ax2.scatter(X[y_ac == 0, 0], X[y_ac == 0, 1])
ax2.scatter(X[y_ac == 1, 0], X[y_ac == 1, 1])
ax2.set_title('Agglomerative clustering')
ax2.set_xlabel('Feature 1')
ax2.set_ylabel('Feature 2')
plt.legend()
plt.tight_layout()
plt.show()
```



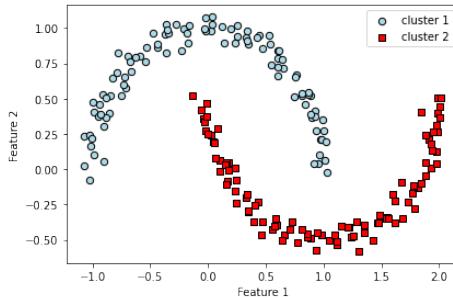
Мы видим, что оба алгоритма не справились с задачей. Наконец попробуем DBSCAN:

```
from sklearn.cluster import DBSCAN
db = DBSCAN(eps=0.2, min_samples=5,
```

```

metric='euclidean')
y_db = db.fit_predict(X)
plt.scatter(X[y_db == 0 ...])
plt.scatter(X[y_db == 1 ...])
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.tight_layout()
plt.show()

```



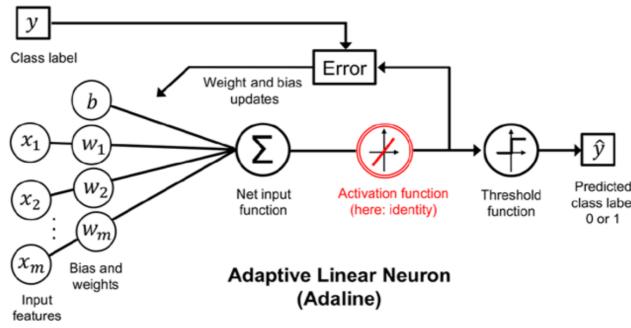
Как мы видим, алгоритм отлично справился с задачей, что подчёркивает сильную сторону DBSCAN – кластеризация данных произвольной формы. Однако важно упомянуть некоторые минусы нашего DBSCAN. С увеличением числа фич (при фиксированном числе примеров) начинает усиливаться *curse of dimensionality*. Это особенно становится проблемой, если мы используем Euclidean distance метрику. Однако эта проблема не уникальна для DBSCAN, она также есть и для k-means, и для hierarchical алгоритмов, которые тоже используют Euclidean distance метрику. Также у нас есть два гиперпараметра *MinPts* и ϵ , найти их хорошую комбинацию может быть сложно, если разность плотностей в датасете относительно большая. Есть четвёртый класс более продвинутых алгоритмов кластеризации – это **graph-based clustering**. Самые выдающиеся члены этой группы – это **spectral clustering** алгоритмы. Хотя есть много разных реализаций spectral clustering, у них есть общая черта – они используют собственные вектора similarity (или distance) матрицы, чтобы вывести взаимоотношения между кластерами. На практике не всегда очевидно какой алгоритм будет работать лучше, особенно в случае больших размерностей. Успех зависит не только от гиперпараметров и алгоритма, а ещё от выбора подходящей метрики. В контексте curse of dimensionality часто применяют техники уменьшения размерности (PCA, t-SNE из главы 5). Также часто сжимают датасет на двумерное подпространство, что позволяет визуализировать кластеры и присваивать метки на основании two-dimensional scatterplots, а это очень полезно для оценки результатов. Пришло время представить одни из самых интересных алгоритмов в supervised learning – multilayer artificial neural networks. Благодаря недавно разработанным *deep learning* алгоритмам, нейронные сети считаются самыми современными для решения таких сложных задач как image classification, natural language processing и speech recognition. В главе 12 мы будем работать с библиотекой *PyTorch*, которая специализируется на тренировке многослойных нейросетей. Эффективность достигается благодаря *graphics processing units (gpu)*.

11 Multilayer Artificial Neural Network

Deep learning – это подобласть ml, которая специализируется на эффективном обучении многослойных artificial **neural networks (NNs)**. В этой главе мы пройдём основы для более сложных глав про **deep neural network (DNN)**, которые хорошо подходят для анализа изображений и текста.

11.1 Modeling complex functions with artificial NNs

В самом начале, в главе 2, мы начали изучать **artificial neurons**, которые являются строительными блоками для многослойной NN. После первой реализации **McCulloch-Pitts neuron** модели (Rosenblatt's perceptron), прошло много времени перед тем, как NNs стали опять популярны, когда представили backpropagation алгоритм для более эффективного обучения. На википедии можно почитать историю **artificial intelligence (AI)**, ml и NNs. Кратко вспомним single-layer NN, которую мы обсудили в главе 2, **ADaptive LInear NEuron (Adaline)** алгоритм. Мы писали этот алгоритм для бинарной классификации и использовали GD, чтобы обучить веса. Вспомним пару формул: $\Delta w_j = -\eta \frac{\partial L}{\partial w_j}$ и $\Delta b = -\eta \frac{\partial L}{\partial b}$. Мы считали градиент на основе всего train датасета и обновляли веса, делая шаг в противоположную сторону к градиенту $\nabla L(w)$. Чтобы найти правильные веса, мы оптимизировали objective function MSE $L(w)$. Мы аккуратно выбирали η , чтобы быстро обучаться, но при этом, чтобы не перепрыгнуть через глобальный минимум. См. картинку:



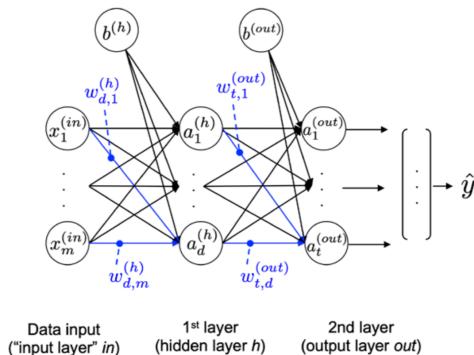
В GD мы обновляли все веса одновременно после каждой эпохи, частные производные:

$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{n} \sum_i (y^{(i)} - a^{(i)})^2 = -\frac{2}{n} \sum_i (y^{(i)} - a^{(i)}) x_j^{(i)}$$

Здесь $a^{(i)}$ – активация нейрона – линейная функция в случае Adaline. Далее мы определили функцию активации $\sigma(\cdot) = z = a$. Net input выглядит так: $z = \sum_j w_j x_j + b = w^T x + b$. Также у нас была threshold функция:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0; \\ 0 & \text{otherwise} \end{cases}$$

Заметим, что, хотя у нас входной и выходной слой (то есть их 2), мы всё равно называем сеть однослойной, т.к. связь у нас всего одна. У нас был трюк для более быстрого обучения – SGD, он приближает потери от одного примера (online learning) или маленького подмножества примеров (mini-batch learning), мы будем использовать эту концепцию, когда будем писать **multilayer perceptron (MLP)**. Благодаря частому обновлению весов noisy структура полезна для обучения multilayer NNs с нелинейной функцией активации, которая не имеет выпуклой loss функции. Дополнительный шум может помочь избежать локального минимума, мы обсудим это позже. Теперь наконец-то поговорим про multilayer NN архитектуру. Мы научимся соединять несколько single neurons в multilayer feedforward NN – специальный тип *fully connected* network, также называют MLP. Посмотрим на идею MLP из двух слоёв:



Units в hidden layer fully connected к входным данным, а выходной слой fully connected к hidden layer. Если такая сеть имеет больше одного hidden layer, то её называют **deep NN**. Мы можем добавлять любое число hidden layers, на практике о числе layers и units можно думать как о гиперпараметрах. Однако, loss gradients для обновления параметров сети, которые мы посчитаем позже с помощью backpropagation, будут становиться всё меньше и меньше с каждым новым слоем. Эта *vanishing gradient* проблема делает обучение модели более сложным. Поэтому были придуманы специальные алгоритмы, чтобы обучать такие DNN структуры, они известны под названием **deep learning**. Как мы видим на картинке, мы обозначаем i th activation unit в l th layer как $a_i^{(l)}$. На самом деле $b^{(h)}$ и $b^{(out)}$ – вектора с числом элементов равным числу вершин в слое. Каждая вершина в слое l соединена со всеми вершинами в слое $l+1$ через весовой коэффициент. Например, соединение между k th unit в слое l с j th unit в слое $l+1$ записывается как $w_{j,k}^{(l+1)}$. Мы обозначаем weight matrix, которая соединяет входной слой с hidden layer $W^{(h)}$, а которая соединяет hidden layer с выходным – $W^{(out)}$. Одного unit в output layer достаточно для бинарной классификации, но на картинке мы видели общую идею, которая позволяет сделать multiclass classification с помощью обобщения OvA техники. Чтобы лучше понять как это работает, вспомним one-hot представление

categorical переменных (глава 4). Например, три типа меток можно закодировать так:

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Такое кол-во индексов сначала кажется избыточным, но это приобретёт смысл, когда мы векторизируем NN представление. Ещё раз, матрица $W^{(h)}$, которая соединяет входной и скрытый слои, имеет размерность $d \times m$. Поговорим про активацию neural network с помощью **forward propagation** – процесса для подсчёта выхода MLP модели. Чтобы понять, зачем это нужно для обучения MLP модели, запишем процесс обучения MLP в виде трёх шагов:

1. Начиная на входном слое, мы forward propagate шаблоны тренировочных данных через сеть, чтобы сгенерировать выход.
2. На основе выхода сети мы считаем loss, который мы хотим минимизировать, используя loss функцию (опишем её дальше).
3. Мы backpropagate loss, считаем его производные относительно весов и bias unit, обновляем модель.

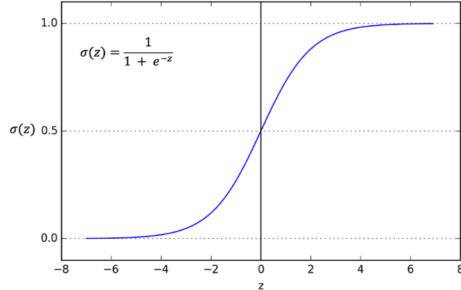
Повторим эти три шага для нескольких эпох, дальше будем применять forward propagation, чтобы считать выход, к которому уже применим threshold функцию, чтобы получить предсказанные классы в one-hot виде. Пройдём по шагам forward propagation. Сначала мы считаем $a_1^{(h)}$:

$$\begin{aligned} z_1^{(h)} &= x_1^{(in)} w_{1,1}^{(h)} + x_2^{(in)} w_{1,2}^{(h)} + \cdots + x_m^{(in)} w_{1,m}^{(h)} \\ a_1^{(h)} &= \sigma(z_1^{(h)}) \end{aligned}$$

Здесь $z_1^{(h)}$ – это net input, а $\sigma(\cdot)$ – это функция активации, которая должна быть дифференцируемой, чтобы обучать веса. Чтобы уметь решать сложные задачи (например, классификация изображений), нам нужны нелинейные функции активации, например, *sigmoid (logistic) activation function* (глава 3):

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

Эта функция, которая сопоставляет net input z к logistic distribution в диапазоне от 0 до 1:



MLP – типичный пример feedforward artificial NN. Термин **feedforward** обозначает то, что каждый слой служит входом для следующего слоя без циклов, в отличие от recurrent NNs – архитектура, которую мы обсудим дальше в этой главе, а более подробно в главе 15. Термин *multilayer perceptron* может звучать немного странно, т.к. здесь *artificial neurons* – это обычно sigmoid units, а не perceptrons. Мы можем думать о этих нейронах как о logistic regression units. В целях эффективности напишем активацию в более компактной форме, что позволит нам векторизовать нашу реализацию:

$$\begin{aligned} z^{(h)} &= x^{(in)} W^{(h)T} + b^{(h)} \\ a^{(h)} &= \sigma(z^{(h)}) \end{aligned}$$

Здесь $x^{(in)}$ – это вектор фич размера $1 \times m$, $W^{(h)}$ – это weight matrix размера $d \times m$, где d – число units в hidden layer, в итоге получаем вектор размера $1 \times d$ – net input $z^{(h)}$. Теперь можно обобщить на все n примеров:

$$\begin{aligned} Z^{(h)} &= X^{(in)} W^{(h)T} + b^{(h)} \\ A^{(h)} &= \sigma(Z^{(h)}) \end{aligned}$$

Аналогично мы можем написать активацию для выходного слоя:

$$Z^{(out)} = A^{(h)} W^{(out)T} + b^{(out)}$$

11.2 Classifying handwritten digits

Перед обсуждением алгоритма для обучения весов, backpropagation, посмотрим на NN в действии. Мы будем использовать популярный Mixed National Institute of Standards and Technology (**MNIST**) датасет. Обсудим получение и подготовку MNIST датасета, который доступен по [ссылке](#). Вместо того, чтобы загружать эти четыре архива и препроцессировать их в NumPy массивы, мы будем использовать функцию `fetch_openml` из scikit-learn:

```
from sklearn.datasets import fetch_openml
# return_X_y=True — вернуть в удобном
# формате (data, target)
X, y = fetch_openml('mnist_784', version=1,
                    return_X_y=True)
X = X.values
y = y.astype(int).values
```

Эта функция скачивает датасет из OpenML ([ссылка](#) на сайт) в виде pandas DataFrame и Series objects, поэтому нужно явно вызвать атрибут `.values`, чтобы получить NumPy массив. Проверим размерность данных:

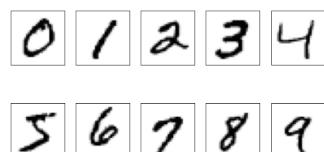
```
print(X.shape) # (70000, 784)
print(y.shape) # (70000,)
```

Изображения размера 28×28 состоят из пикселей, каждый из которых представляет grayscale intensity value. Функция автоматически развернула двумерный массив в строку из 784 фич. Теперь нормализуем значения пикселей в отрезок от -1 до 1 (изначально от 0 до 255):

```
X = ((X / 255.) - .5) * 2
```

Причина в том, что gradient-based оптимизации намного стабильнее при этих условиях (см. главу 2). Обращу внимание, что способ нормализации отличается от тех, что мы применяли в предыдущих главах. С помощью функции `imshow` нарисуем эти цифры:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(nrows=2, ncols=5,
                      sharex=True, sharey=True)
ax = ax.flatten()
for i in range(10):
    img = X[y == i][0].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys')
ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
plt.show()
```



Также выведем несколько примеров одной цифры:

```
fig, ax = plt.subplots(nrows=5, ncols=5,
                      sharex=True, sharey=True)
ax = ax.flatten()
for i in range(25):
    img = X[y == 7][i].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys')
ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
plt.show()
```

Изображение вставлять не буду, оно занимает много места, попробуйте сами. Теперь разделим датасет на training, validation и test подмножества:

```
from sklearn.model_selection import train_test_split
X_temp, X_test, y_temp, y_test = train_test_split(
    X, y, test_size=10000, random_state=123, stratify=y)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_temp, y_temp, test_size=5000, random_state=123, stratify=y_temp)
```

Наконец-то реализуем multilayer perceptron. Код, который мы напишем, будет содержать части, которые мы ещё не обсудили (но обсудим позже), например, backpropagation алгоритм. Перед самим кодом MLP напишем две вспомогательные функции `sigmoid` и `int_to_onehot`:

```

import numpy as np

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

def int_to_onehot(y, num_labels):
    ary = np.zeros((y.shape[0], num_labels))
    for i, val in enumerate(y):
        ary[i, val] = 1
    return ary

```

Первая часть кода:

```

class NeuralNetMLP:
    def __init__(self, num_features, num_hidden,
                 num_classes, random_seed=123):
        super().__init__()
        self.num_classes = num_classes
        rng = np.random.RandomState(random_seed)
        self.weight_h = rng.normal(
            loc=0.0, scale=0.1, size=(num_hidden, num_features))
        self.bias_h = np.zeros(num_hidden)
        self.weight_out = rng.normal(
            loc=0.0, scale=0.1, size=(num_classes, num_hidden))
        self.bias_out = np.zeros(num_classes)

    def forward(self, x):
        z_h = np.dot(x, self.weight_h.T) + self.bias_h
        a_h = sigmoid(z_h)

        z_out = np.dot(a_h, self.weight_out.T) + self.bias_out
        a_out = sigmoid(z_out)
        return a_h, a_out

```

Метод `forward` возвращает activation values и от hidden layer, и от output layer (`a_h` и `a_out`). `a_out` показывает class-membership вероятности, которые можно конвертировать в метки. `a_h` нужны, чтобы оптимизировать параметры модели (веса и bias units для hidden и output layers). Теперь напишем `backward` метод:

```

class NeuralNetMLP:
    # ...

    def backward(self, x, a_h, a_out, y):
        y_onehot = int_to_onehot(y, self.num_classes)

        # Часть 1.
        # dLoss/dOutWeights =
        # dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight
        d_loss_d_a_out = 2. * (a_out - y_onehot) / y.shape[0]
        d_a_out_d_z_out = a_out * (1. - a_out) # произведение от sigmoid
        delta_out = d_loss_d_a_out * d_a_out_d_z_out # [n_examples, n_classes]
        d_z_out_dw_out = a_h

        # [n_classes, n_examples] dot [n_examples, n_hidden] =
        # [n_classes, n_hidden]
        d_loss_dw_out = np.dot(delta_out.T, d_z_out_dw_out)
        d_loss_db_out = np.sum(delta_out, axis=0)

        # Часть 2.
        # dLoss/dHiddenWeights = DeltaOut * dOutNet/dHiddenAct *
        # dHiddenAct/dHiddenNet * dHiddenNet/dWeight
        d_z_out_a_h = self.weight_out
        d_loss_a_h = np.dot(delta_out, d_z_out_a_h) # [n_examples, n_hidden]
        d_a_h_d_z_h = a_h * (1. - a_h) # произведение от sigmoid
        d_z_h_d_w_h = x

        d_loss_d_w_h = np.dot((d_loss_a_h * d_a_h_d_z_h).T,
                             d_z_h_d_w_h)
        d_loss_d_b_h = np.sum((d_loss_a_h * d_a_h_d_z_h), axis=0)

    return (d_loss_dw_out, d_loss_db_out,
            d_loss_d_w_h, d_loss_d_b_h)

```

Этот метод реализует *backpropagation* алгоритм, который считает градиенты, их мы используем, чтобы обновить модель с помощью GD. Функция потерь для простоты такая же как и в Adaline,

т.е. MSE. В следующих главах мы посмотрим на другие loss функции, например, на multi-category cross-entropy loss, которая является обобщением loss функции для бинарной лог. рег. на несколько классов. Заметим, что это ООП способ реализации, он отличается от обычного scikit-learn API, это сделано для того, чтобы лучше понять как информация проходит по нейронной сети. С другой стороны такая реализация очень похожа на API других deep learning libraries таких, как PyTorch. Создадим объект:

```
model = NeuralNetMLP(num_features=28*28,
                      num_hidden=50,
                      num_classes=10)
```

Мы используем `sigmoid` функцию после обоих слоёв, в следующих главах мы пройдём и другие. Теперь реализуем функцию тренировки, с помощью которой будем учить модель на mini-batches через backpropagation. Сначала напишем mini-batch generator для SGD:

```
import numpy as np
num_epochs = 50
minibatch_size = 100

def minibatch_generator(X, y, minibatch_size):
    # список чисел от 0 до размера-1
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices)
    for start_idx in range(0, indices.shape[0] - minibatch_size + 1,
                           minibatch_size):
        batch_idx = indices[start_idx:start_idx + minibatch_size]
        yield X[batch_idx], y[batch_idx]

# проместируем работу:
for i in range(num_epochs):
    minibatch_gen = minibatch_generator(
        X_train, y_train, minibatch_size)
    for X_train_mini, y_train_mini in minibatch_gen:
        break
    break
print(X_train_mini.shape) # (100, 784)
print(y_train_mini.shape) # (100,)
```

Теперь нам нужны loss функция и performance метрика:

```
def mse_loss(targets, probas, num_labels=10):
    onehot_targets = int_to_onehot(
        targets, num_labels=num_labels)
    # среднее значение по всем числам
    return np.mean((onehot_targets - probas)**2)

def accuracy(targets, predicted_labels):
    return np.mean(predicted_labels == targets)

# проместируем:
_, probas = model.forward(X_valid)
mse = mse_loss(y_valid, probas)
print(f'Initial validation MSE: {mse:.1f}')
# Initial validation MSE: 0.3

predicted_labels = np.argmax(probas, axis=1)
acc = accuracy(y_valid, predicted_labels)
print(f'Initial validation accuracy: {acc*100:.1f}%')
# Initial validation accuracy: 9.4%
```

На практике не имеет значения, берём мы среднее значение сначала по строке или сначала по столбцу squared-difference матрицы, поэтому мы не указывали `axis`. На практике память компьютера обычно ограничена тем, сколько данных модель может обработать за раз (из-за больших матричных вычислений), поэтому мы перепишем функции на основе mini-batch генератора:

```
def compute_mse_and_acc(nnet, X, y, num_labels=10,
                        minibatch_size=100):
    mse, correct_pred, num_examples = 0., 0, 0
    minibatch_gen = minibatch_generator(X, y, minibatch_size)
    for i, (features, targets) in enumerate(minibatch_gen):
        _, probas = nnet.forward(features)
        predicted_labels = np.argmax(probas, axis=1)
        onehot_targets = int_to_onehot(
            targets, num_labels=num_labels)
        loss = np.mean((onehot_targets - probas)**2)
        correct_pred += (predicted_labels == targets).sum()
        num_examples += targets.shape[0]
        mse += loss
```

```

mse = mse/i
acc = correct_pred/num_examples
return mse, acc

mse, acc = compute_mse_and_acc(model, X_valid, y_valid)
print(f'Initial valid MSE: {mse:.1f}')
# Initial valid MSE: 0.3
print(f'Initial valid accuracy: {acc*100:.1f}%')
# Initial valid accuracy: 9.4%

```

Результаты как мы видим такие же, кроме небольшой ошибки округления в MSE (0.27 vs 0.28). Напишем код для тренировки модели:

```

def train(model, X_train, y_train, X_valid, y_valid, num_epochs,
          learning_rate=0.1):
    epoch_loss = []
    epoch_train_acc = []
    epoch_valid_acc = []

    for e in range(num_epochs):
        minibatch_gen = minibatch_generator(
            X_train, y_train, minibatch_size)
        for X_train_mini, y_train_mini in minibatch_gen:
            a_h, a_out = model.forward(X_train_mini)
            d_loss_d_w_out, d_loss_d_b_out, \
            d_loss_d_w_h, d_loss_d_b_h = \
                model.backward(X_train_mini, a_h, a_out, y_train_mini)

            model.weight_h -= learning_rate * d_loss_d_w_h
            model.bias_h -= learning_rate * d_loss_d_b_h
            model.weight_out -= learning_rate * d_loss_d_w_out
            model.bias_out -= learning_rate * d_loss_d_b_out

        train_mse, train_acc = compute_mse_and_acc(
            model, X_train, y_train)
        valid_mse, valid_acc = compute_mse_and_acc(
            model, X_valid, y_valid)
        train_acc, valid_acc = train_acc*100, valid_acc*100

        epoch_train_acc.append(train_acc)
        epoch_valid_acc.append(valid_acc)
        epoch_loss.append(train_mse)

        print(f'Epoch: {e+1:03d}/{num_epochs:03d} '
              f'| Train MSE: {train_mse:.2f} '
              f'| Train Acc: {train_acc:.2f}% '
              f'| Valid Acc: {valid_acc:.2f}%')

    return epoch_loss, epoch_train_acc, epoch_valid_acc

```

Теперь запустим эту функцию и обучим модель, это может занять пару минут:

```

# для перемешивания training set:
np.random.seed(123)
epoch_loss, epoch_train_acc, epoch_valid_acc = train(
    model, X_train, y_train, X_valid, y_valid,
    num_epochs=50, learning_rate=0.1)
# Вывод такой:
# Epoch: 001/050 | Train MSE: 0.05 | Train Acc: 76.15% | Valid Acc: 75.98%
# Epoch: 002/050 | Train MSE: 0.03 | Train Acc: 85.45% | Valid Acc: 85.04%
#
# ...
# Epoch: 050/050 | Train MSE: 0.01 | Train Acc: 95.61% | Valid Acc: 94.78%

```

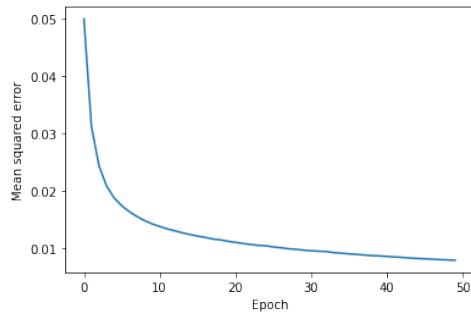
Причина, по которой мы пишем весь данный вывод в том, что в обучении NN очень полезно сравнивать train и validation accuracy. Это помогает понять хорошая ли у нас архитектура и гиперпараметры модели. Обучение таких моделей довольно дорогое, поэтому лучше остановить заранее и запустить с другими параметрами. Если мы видим увеличение зазора между training и validation показателями (переобучение), то мы тоже можем увидеть это, не дожидаясь конца обучения. Теперь обсудим performance нашей NN модели более детально. Визуализируем MSE loss:

```

plt.plot(range(len(epoch_loss)), epoch_loss)
plt.ylabel('Mean squared error')
plt.xlabel('Epoch')
plt.show()

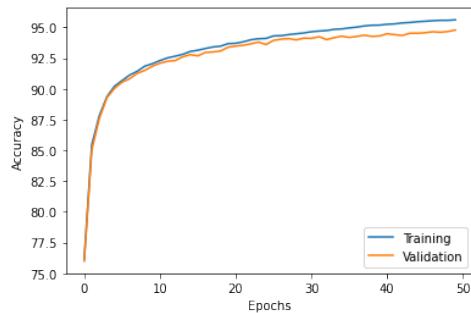
```

Первые 10 эпох алгоритм сходится очень быстро, последние 10 – довольно медленно. Однако небольшой наклон между 40 и 50 эпохами говорит о том, что при дополнительных эпохах mse будет дальше уменьшаться:



Теперь взглянем на training и validation accuracy:

```
plt.plot(range(len(epoch_train_acc)), epoch_train_acc,
         label='Training')
plt.plot(range(len(epoch_valid_acc)), epoch_valid_acc,
         label='Validation')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend(loc='lower right')
plt.show()
```



На графике видно, что зазор между training и validation accuracy увеличивается с ростом числа эпох. Уменьшить переобучение можно с помощью L2 regularization (глава 3), другая полезная техника для борьбы с переобучением в NNs – *dropout*, которую мы рассмотрим в главе 14. Вычислим generalization performance нашей модели:

```
test_mse, test_acc = compute_mse_and_acc(model, X_test, y_test)
print(f'Test accuracy: {test_acc*100:.2f}%')
# Test accuracy: 94.54%
```

Мы видим, что test accuracy очень близка к validation accuracy (94.74%). Чтобы дальше улучшить модель, можно увеличить число hidden units, поменять learning rate или использовать другие трюки (были придуманы в последнее время, но они не в рамках конспекта). В главе 14 мы узнаем о разных NN архитектурах, которые хорошо показывают себя при работе с изображениями. Также в ней мы узнаем дополнительные трюки такие, как *adaptive learning rates*, более сложный SGD-based optimization алгоритм, batch normalization и dropout. Другие частые техники, которые в не в рамках конспекта:

- Добавление skip-connections, которые являются основным вкладом residual NNs.
- Использование learning rate schedulers, которые меняют lr во время обучения.
- Прикрепление loss functions к ранним слоям сети, так сделано в популярной архитектуре *Inception v3*.

Посмотрим на изображения, в которых наша сеть ошибается:

```
X_test_subset = X_test[:1000, :]
y_test_subset = y_test[:1000]
_, probas = model.forward(X_test_subset)
test_pred = np.argmax(probas, axis=1)
misclassified_images = \
    X_test_subset[y_test_subset != test_pred][:25]
misclassified_labels = test_pred[y_test_subset != test_pred][:25]
correct_labels = y_test_subset[y_test_subset != test_pred][:25]

fig, ax = plt.subplots(nrows=5, ncols=5,
                      sharex=True, sharey=True,
```

```

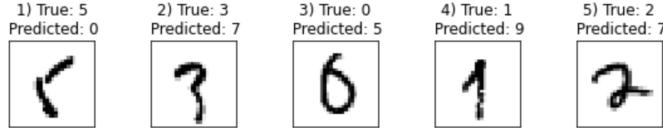
    figsize=(8, 8))

ax = ax.flatten()
for i in range(25):
    img = misclassified_images[i].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys', interpolation='nearest')
    ax[i].set_title(f'{i+1}:\n' +
                    f'True: {correct_labels[i]}\n' +
                    f'Predicted: {misclassified_labels[i]}')

ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
plt.show()

```

Всю картинку вставлять не буду, только её верхнюю часть:



Если посмотреть на картинку (все 25 примеров, а не только 5), то можно заметить, что модели тяжело определить цифру 7, у которой есть горизонтальная линия по середине. Вспомнив наши изображения цифры 7 в начале главы, можно сделать предположение, что цифра 7 с хвостиком очень мало представлена в датасете, именно поэтому модель и ошибается.

11.3 Training an artificial neural network

Теперь давайте копнём глубже в такие концепции как *loss computation* и *backpropagation algorithm*. Начнём с подсчёта loss функции. Мы использовали MSE loss, т.к. это делает вывод градиентов более простым; multi-category cross-entropy loss более частый выбор для обучения NN моделей. Если мы предсказываем цифру 2, то активация 3-его слоя может выглядеть так:

$$a^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

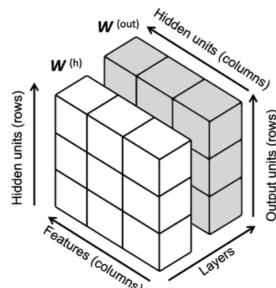
Таким образом, наши MSE потери должны либо суммироваться, либо усредняться по t activation units в дополнение к усреднению по примерами:

$$L(W, b) = \frac{1}{n} \sum_1^n \frac{1}{t} \sum_{j=1}^t \left(y_j^{[i]} - a_j^{(out)[i]} \right)^2$$

Для минимизации этой функции, нужно посчитать:

$$\frac{\partial L(W, b)}{\partial w_{j,i}^{(l)}}$$

W состоит из нескольких матриц, посмотрим на визуализацию трёхмерного тензора:



На этой картинке может показаться, что у двух матриц кол-во строк и столбцов равно, но это не так; такое может быть только, если кол-во входных, скрытых и выходных нейронов равно одному числу. Мы можем думать о backpropagation как о очень эффективном подходе для посчёта частных производных у сложных, невыпуклых loss функций в многослойных NNs; поверхность ошибки NN loss функции не выпуклая и не гладкая по отношению к параметрам. Вспомним правило цепочки,

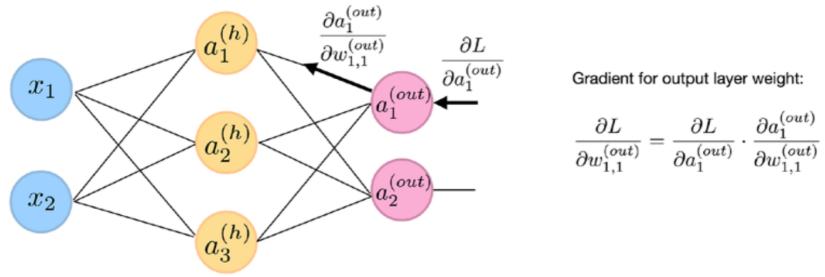
это способ считать производную вложенной функции, например, пусть $F(x) = f(g(h(u(v(x))))$, тогда:

$$\frac{dF}{dx} = \frac{d}{dx} F(x) = \frac{d}{dx} f(g(h(u(v(x))))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$$

В контексте компьютерной алгебры, набор техник **automatic differentiation** был изобретён для эффективного решения таких задач. У данных техник есть два режима, *forward* и *reverse*, наш backpropagation – это просто специальный случай reverse-mode a.d. Ключевой пункт в том, что когда мы применяем правило цепочки в forward режиме, то это может быть довольно дорого, т.к. мы должны будем умножать большие матрицы для каждого слоя (Jacobians), которые в итоге надо будет умножать на вектор, чтобы получить выход. Трюк reverse mode заключается в том, что мы обрабатываем правило цепочки справа налево. Мы умножаем матрицу на вектор, что приводит к другому вектору и т.д. Естественно умножение матриц намного дороже, чем умножение матрицы на вектор, именно поэтому backpropagation – это один из самых популярных алгоритмов для обучения NN. Теперь наконец-то посмотрим как обучить NN с помощью этого алгоритма. Вспомним forward propagation:

$$\begin{aligned} Z^{(h)} &= X^{(in)} W^{(h)T} + b^{(h)} \quad (\text{net input of the hidden layer}) \\ A^{(h)} &= \sigma(Z^{(h)}) \quad (\text{activation of the hidden layer}) \\ Z^{(out)} &= A^{(h)} W^{(out)T} + b^{(out)} \quad (\text{net input of the output layer}) \\ A^{(out)} &= \sigma(Z^{(out)}) \quad (\text{activation of the output layer}) \end{aligned}$$

В backpropagation мы проталкиваем ошибку справа налево, мы можем думать об этом, как о применении правила цепочки для вычисления прямого хода, чтобы посчитать градиент потерь. См. картинку для лучшего понимания:



Если мы включим net inputs z , то это превратится в такое равенство:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}}$$

Для простоты мы опустим усреднение по примерам, т.е. мы уберём член $\frac{1}{n} \sum_{i=1}^n$ из следующих равенств. Начнём с первого множителя, это частная производная MSE loss функции (это просто squared error, т.к. мы опустили усреднение):

$$\frac{\partial L}{\partial a_1^{(out)}} = \frac{\partial}{\partial a_1^{(out)}} \left(y_1 - a_1^{(out)} \right)^2 = 2(a_1^{(out)} - y_1)$$

Следующий множитель – это производная logistic sigmoid activation функции:

$$\frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} = \frac{\partial}{\partial z_1^{(out)}} \frac{1}{1 + e^{z_1^{(out)}}} = \dots = \left(\frac{1}{1 + e^{z_1^{(out)}}} \right) \left(1 - \frac{1}{1 + e^{z_1^{(out)}}} \right) = a_1^{(out)}(1 - a_1^{(out)})$$

Последним шагом считаем производную net input:

$$\frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = \frac{\partial}{\partial w_{1,1}^{(out)}} a_1^{(h)} w_{1,1}^{(out)} + b_1^{(out)} = a_1^{(h)}$$

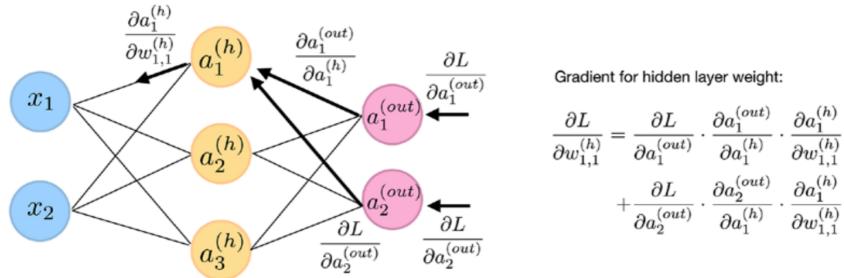
Осталось собрать три множителя вместе, а затем использовать это значение, чтобы обновить вес через SGD:

$$w_{1,1}^{(out)} := w_{1,1}^{(out)} - \eta \frac{\partial L}{\partial w_{1,1}^{(out)}}$$

В нашей реализации `NeuralNetMLP()` мы считали производную в векторной форме, пролистайте назад конспект и прочитайте опять реализацию `.backward()` метода. Вспомним как мы посчитали `d_loss_db_out = np.sum(delta_out, axis=0)` и поймём почему это так. Если мы будем брать теперь производную по $b_1^{(out)}$, то мы получим матрицу из единиц, что при аналогичном умножении (как в строке выше) даст это же суммирование. Для удобства переиспользования мы создали след. переменную:

$$\sigma_1^{(out)} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}}$$

Поговорим о весах в hidden layer, опять приложу картинку для лучшего понимания:



Важно сказать, что так как вес $w_{1,1}^{(h)}$ соединён с обеими выходными вершинами, то мы должны использовать *multi-variable* chain rule, чтобы просуммировать оба пути. Аналогично можно включить z и посчитать отдельные части:

$$\frac{\partial L}{\partial w_{1,1}^{(h)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} + \frac{\partial L}{\partial a_2^{(out)}} \cdot \frac{\partial a_2^{(out)}}{\partial z_2^{(out)}} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}$$

Вспомним про член $\sigma^{(out)}$ и упростим:

$$\frac{\partial L}{\partial w_{1,1}^{(h)}} = \sigma_1^{(out)} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} + \sigma_2^{(out)} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}$$

11.4 Ещё немного про NNs

Сначала поговорим о сходимости в нейронных сетях. Почему мы используем не обычный GD, а mini-batch learning. Мы как-то уже обсуждали, что можно использовать SGD для online learning, в нём мы считали градиент на основании одного примера, $k = 1$. Этот подход имеет быструю сходимость и точные результаты. Mini-batch learning – это специальная форма SGD, где мы считаем градиент на подмножестве примеров размера k , $1 < k < n$. Преимущество над онлайн обучением в том, что можно использовать векторную реализацию для повышения эффективности. Когда мы считаем вероятного президента до выборов, то мы берём каких-то случайных людей, т.е. формируем mini-batch, это очень близко к реальным выборам; так и у нас. Многослойные NNs намного сложнее обучать по сравнению с простыми алгоритмами, у нас обычно сотни или миллионы весов, а поверхность далеко не гладкая, поэтому можно попасть в локальный минимум. Если lr большой, то с одной стороны мы вылезем из локального минимума, с другой стороны мы можем перескочить глобальный минимум вовсе. Может возникнуть вопрос, зачем мы реализовали всё это с нуля, а не взяли из библиотеки. Более сложные NN модели в след. главах мы будем тренировать, используя библиотеку PyTorch. Написание кода дало хорошее понимание backpropagation и NN обучения в целом. Т.к. мы понимаем работу feedforward NNs, мы готовы писать более сложные DNNs, используя PyTorch, что позволяет конструировать сети более эффективно (глава 12). PyTorch популярен т.к. он умеет оптимизировать математические выражения для вычислений на многомерных массивах с использованием graphics processing units (GPUs). Scikit-learn имеет базовую реализацию MLP, `MLPClassifier`, но лучше использовать специализированные библиотеки.

12 Parallelizing NN Training with PyTorch

12.1 PyTorch and training performance

Для начала поговорим про проблемы с производительностью. Мы уже видели, что многие функции в scikit-learn позволяют разделить вычисления на несколько processing units. Однако, по умолчанию,

Python выполняется на одном ядре из-за **global interpreter lock (GIL)**. Всё равно большинство даже крутых компьютеров редко имеют больше 8 или 16 ядер. В главе 11 мы реализовали MLP с одним скрытым слоем из 100 units, нам нужно было оптимизировать $(784 \cdot 100 + 100) + (100 \cdot 10 + 10) = 79,510$ весовых параметров, хотя изображение размером всего 28×28 . Очевидное решение – использовать **graphics processing units (GPUs)**. Современные GPUs работают намного лучше по сравнению с **central processing units (CPUs)**. Например, Intel Core i9-11900KB Processor имеет 16 ядер и скорость 45.8GB/s, а вот NVIDIA GeForce RTX 3080 Ti имеет 10240 ядер и скорость 912.1GB/s. Мы получаем в 640 раз больше ядер, также мы можем считать примерно в 46 раз больше вычислений с плавающей точкой в секунду. Мы не используем их везде, т.к. писать код под них сложнее, чем просто выполнить код в интерпретаторе. Чтобы использовать GPU есть специальные пакеты CUDA и OpenCL, однако это не самый удобный способ писать алгоритмы ml. Именно поэтому и был написан и придуман PyTorch. Что такое PyTorch? PyTorch – это удобный интерфейс для работы с ml, который изначально придумали в **Facebook AI Research (FAIR)** лаборатории; кстати, его использует, например, Tesla Autopilot. Чтобы улучшить производительность, PyTorch разрешает выполнение на CPUs, GPUs и XLA устройствах (compiler-based linear algebra execution engine), таких как TPUs (Tensor Processing Unit, специализированная интегральная схема ускорителя искусственного интеллекта (ASIC)). PyTorch официально поддерживает разработку CUDA-enabled и ROCm (Open Software Platform for GPU Compute) GPUs. PyTorch построен на computation graph, каждая вершина которого представляет операцию с нулём или более входами и выходами. PyTorch предоставляет императивную среду, которая считает операции и выполняет вычисления. Следовательно, computation graph в PyTorch определяется неявно, а не строится заранее и выполняется после. Тензоры в PyTorch похожи на NumPy массивы за тем исключением, что они оптимизированы для automatic differentiation и могут выполняться на GPUs.

12.2 First steps with PyTorch

В этой части мы сделаем первые шаги в использовании low-level PyTorch API. Для начала установим PyTorch:

```
# latest stable version
! pip install torch torchvision
```

Если мы хотим использовать GPUs, нам нужна совместимая NVIDIA graphics card, которая поддерживает CUDA и cuDNN, установим PyTorch с GPU поддержкой:

```
import torch
import torchvision
print(torch.__version__) # 1.12.1+cu113
print(torchvision.__version__) # 0.13.1+cu113
! pip install torch==1.12.1+cu113 torchvision==0.13.1+cu113 \
-f https://download.pytorch.org/whl/torch_stable.html
```

PyTorch активно развивается, поэтому каждый месяц может выйти новая версия, так что нужно проверить его версию:

```
! python -c 'import torch; print(torch.__version__)' # 1.12.1+cu113
```

Посмотрим как можно создать тензор:

```
import torch
import numpy as np
np.set_printoptions(precision=3)
a = [1, 2, 3]
b = np.array([4, 5, 6], dtype=np.int32)
t_a = torch.tensor(a)
t_b = torch.from_numpy(b)
print(t_a) # tensor([1, 2, 3])
print(t_b) # tensor([4, 5, 6], dtype=torch.int32)

t_ones = torch.ones(2, 3)
print(t_ones.shape) # torch.Size([2, 3])
print(t_ones)
# tensor([[1., 1., 1.],
#         [1., 1., 1.]])>

rand_tensor = torch.rand(2, 3)
print(rand_tensor)
```

Теперь попробуем менять тип данных и форму тензора, мы посмотрим на функции, которые cast, reshape, transpose и squeeze (убрать размерности) наши тензоры. См. код:

```
t_a_new = t_a.to(torch.int64)
print(t_a_new.dtype) # torch.int64
```

```

t = torch.rand(3, 5)
# 0 и 1 — размерности, которые нужно
# транспонировать
t_tr = torch.transpose(t, 0, 1)
print(t.shape, ' → ', t_tr.shape)
# torch.Size([3, 5]) → torch.Size([5, 3])

t = torch.zeros(30)
t_reshape = t.reshape(5, 6)
print(t_reshape.shape) # torch.Size([5, 6])

# unnecessary dimensions — dimensions that have size 1, which are not needed
t = torch.zeros(1, 2, 1, 4, 1)
# dim=2 — if given, the input will be squeezed only in this dimension
t_sqz = torch.squeeze(t, 2)
print(t.shape, ' → ', t_sqz.shape)
# torch.Size([1, 2, 1, 4, 1]) → torch.Size([1, 2, 4, 1])

t_sqz = torch.squeeze(t)
print(t.shape, ' → ', t_sqz.shape) # torch.Size([2, 4])

```

Теперь посмотрим на математические операции для тензоров:

```

torch.manual_seed(1)
# uniform distribution in the range from -1 to 1
# torch.rand — uniform distribution in the range of [0, 1)
t1 = 2 * torch.rand(5, 2) - 1
# standard normal distribution
t2 = torch.normal(mean=0, std=1, size=(5, 2))
t3 = torch.multiply(t1, t2) # element-wise product
print(t3.shape) # torch.Size([5, 2])

```

Чтобы посчитать mean, sum и standard deviation нужно использовать `torch.mean()`, `torch.sum()` и `torch.std()` функции. Например:

```
t4 = torch.mean(t1, axis=0)
print(t4.shape) # torch.Size([2])
```

Посчитаем обычное матричное умножение $t_1 \times t_2^T$, $t_1^T \times t_2$:

```
# matrix-matrix product:
t5 = torch.matmul(t1, torch.transpose(t2, 0, 1))
print(t5.shape) # torch.Size([5, 5])
t6 = torch.matmul(torch.transpose(t1, 0, 1), t2)
print(t6.shape) # torch.Size([2, 2])
```

Используем функцию `torch.linalg.norm()`, чтобы посчитать L^p норму тензора:

```
norm_t1 = torch.linalg.norm(t1, ord=2, dim=1)
print(norm_t1) # tensor([0.6785, 0.5078, 1.1162, 0.5488, 0.1853])
```

Для самопроверки запустим этот код: `np.sqrt(np.sum(np.square(t1.numpy()), axis=1))`. Теперь поговорим про split, stack и concatenate для тензоров. Есть удобная `torch.chunk()` функция, чтобы разбить тензор на тензоры равных размеров. См. код:

```
torch.manual_seed(1)
t = torch.rand(6)
print(t)
# tensor([0.7576, 0.2793, 0.4031, 0.7347, 0.0293, 0.7999])
t_splits = torch.chunk(t, chunks=3, dim=0)
[item.numpy() for item in t_splits]
```

Если размер не делится на `chunks` значение, то последний chunk просто будет меньше остальных. Теперь посмотрим на `torch.split()` функцию:

```
torch.manual_seed(1)
t = torch.rand(5)
print(t) # tensor([0.7576, 0.2793, 0.4031, 0.7347, 0.0293])
t_splits = torch.split(t, split_size_or_sections=[3, 2])
[item.numpy() for item in t_splits]
```

Посмотрим на `torch.stack()` и `torch.cat()` функции:

```
A, B = torch.ones(3), torch.zeros(2)
C = torch.cat([A, B], axis=0)
print(C) # tensor([1., 1., 1., 0., 0.])
```

```
A, B = torch.ones(3), torch.zeros(3)
S = torch.stack([A, B], axis=1)
print(S)
```

```
# tensor ([[1.,  0.],
#         [1.,  0.],
#         [1.,  0.]])
```

12.3 Building input pipelines in PyTorch

Когда мы учим deep NN модель, то обычно используем итеративный алгоритм оптимизации, например, SGD. Модуль для построения NN моделей – `torch.nn`. Иногда нам нужны data-processing pipelines, например, чтобы добавить шум данным для избежания переобучения. Применять все шаги каждый раз очень неудобно, поэтому уже есть эффективный pipeline в PyTorch. В этой части мы посмотрим как создавать `Dataset` и `DataLoader` и работать с ними. Рассмотрим создание `DataLoader` из существующего тензора. См. пример:

```
from torch.utils.data import DataLoader
t = torch.arange(6, dtype=torch.float32)
data_loader = DataLoader(t)
# вывод тензоров с числами от 0 до 5
for item in data_loader:
    print(item)
```

А теперь научимся создавать batches:

```
# drop_last – убирает ли последний batch в случае, если
# его размер меньше остальных
data_loader = DataLoader(t, batch_size=3, drop_last=False)
# 1 – начало нумерации
for i, batch in enumerate(data_loader, 1):
    print(f'batch {i}:', batch)
# batch 1: tensor([0., 1., 2.])
# batch 2: tensor([3., 4., 5.])
```

Теперь поговорим про соединение двух тензоров в один набор данных. Бывает, что наши данные лежат в разных тензорах, например, фичи и метки, тогда нам нужно их склеить в один набор, чтобы брать их в виде tuples. Создадим игрушечный пример:

```
torch.manual_seed(1)
t_x = torch.rand([4, 3], dtype=torch.float32)
t_y = torch.arange(4)
```

Теперь нужно создать joint dataset из двух тензоров. Сначала создадим `Dataset` класс:

```
from torch.utils.data import Dataset
class JointDataset(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]
```

Свой `Dataset` класс должен иметь `__init__()` и `__getitem__()` методы, чтобы его можно было потом использовать в `DataLoader`. Применим этот класс:

```
joint_dataset = JointDataset(t_x, t_y)
for example in joint_dataset:
    print('x: ', example[0], 'y: ', example[1])
# x: tensor([0.7576, 0.2793, 0.4031]) y: tensor(0)
# ...
```

Мы можем просто использовать `torch.utils.data.TensorDataset` класс, если второй датасет – это датасет меток в форме тензоров. А вот и код:

```
from torch.utils.data import TensorDataset
joint_dataset = TensorDataset(t_x, t_y)
```

Поговорим про функции `shuffle`, `batch` и `repeat`. В главе 2 мы говорили, что при использовании SGD нужно подавать данные в случайному порядке в виде batches. Посмотрим как можно перемешать данные и пройти по ним снова. Создадим перемешанную версию нашего joint датасета и пройдём по нему:

```
torch.manual_seed(1)
data_loader = DataLoader(dataset=joint_dataset, batch_size=2, shuffle=True)
for i, batch in enumerate(data_loader, 1):
    print(f'batch {i}:', 'x:', batch[0],
          '\n          y:', batch[1])
```

Во время тренировки нужно ходить по датасету `epochs` раз, каждый раз перемешивая его, пройдём два раза и посмотрим, что будет:

```
for epoch in range(2):
    print(f'epoch {epoch+1}')
    for i, batch in enumerate(data_loader, 1):
        print(f'batch {i}: ', 'x: ', batch[0],
              '\n          y: ', batch[1])
```

То есть для каждой эпохи датасет тоже перемешивается. Научимся создавать датасет из локальных файлов на диске, здесь мы будем хранить изображения. Мы будем использовать два дополнительных модуля: `Image` из `PIL`, чтобы читать содержимое изображения и `transforms` из `torchvision`, чтобы расшифровать содержимое и поменять размер. Картинки можно скачать по [ссылке](#), поместите их в папку `cat_dog_images`. Посмотрим на содержимое картинок, мы будем использовать библиотеку `pathlib`, чтобы сгенерировать список файлов:

```
import pathlib
imgdir_path = pathlib.Path('cat_dog_images')
# glob делает поиск файлов и наносит на pattern
file_list = sorted([str(path) for path in
                    imgdir_path.glob('*.*')])
print(file_list) # ['cat_dog_images/cat-01.jpg', ...]

import matplotlib.pyplot as plt
import os
from PIL import Image
fig = plt.figure(figsize=(10, 5))
for i, file in enumerate(file_list):
    img = Image.open(file)
    print('Image shape:', np.array(img).shape)
    ax = fig.add_subplot(2, 3, i+1)
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(img)
    # basename - returns the final component of a pathname
    ax.set_title(os.path.basename(file), size=15)
plt.tight_layout()
plt.show()
```

Картинки вставлять не буду, их вы и так уже видели, когда скачивали. Размеры изображений такие: (900, 1200, 3), (900, 742, 3), (800, 1200, 3), ... Из этих данных уже видно, что изображения имеют разные aspect ratios, позже мы приведём данные к одному размеру. Теперь присвоим метки:

```
labels = [1 if 'dog' in os.path.basename(file) else 0 \
          for file in file_list]
print(labels) # [0, 0, 0, 1, 1, 1]
```

Сделаем joint датасет:

```
class ImageDataset(Dataset):
    def __init__(self, file_list, labels):
        self.file_list = file_list
        self.labels = labels

    def __getitem__(self, index):
        file = self.file_list[index]
        label = self.labels[index]
        return file, label

    def __len__(self):
        return len(self.labels)

image_dataset = ImageDataset(file_list, labels)
for file, label in image_dataset:
    print(file, label)
# cat_dog_images/cat-01.jpg 0
# cat_dog_images/dog-03.jpg 1
# ...
```

Теперь нам нужно трансформировать датасет: загрузить изображение по пути, декодировать raw content, а также поменять размер. Мы будем использовать `torchvision.transforms` модуль, чтобы поменять размер и конвертировать пиксели в тензоры:

```
import torchvision.transforms as transforms
img_height, img_width = 80, 120
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((img_height, img_width)),
])
```

Теперь можно и обновить класс `ImageDataset`:

```
class ImageDataset(Dataset):
    def __init__(self, file_list, labels, transform=None):
        self.file_list = file_list
        self.labels = labels
        self.transform = transform

    def __getitem__(self, index):
        img = Image.open(self.file_list[index])
        if self.transform is not None:
            img = self.transform(img)
        label = self.labels[index]
        return img, label

    def __len__(self):
        return len(self.labels)

image_dataset = ImageDataset(file_list, labels, transform)
```

Посмотрим на эти изображения после трансформации:

```
fig = plt.figure(figsize=(10, 6))
for i, example in enumerate(image_dataset):
    ax = fig.add_subplot(2, 3, i+1)
    ax.set_xticks([]); ax.set_yticks([])
    # у нас получаются размеры (3, 80, 120),
    # поэтому, чтобы перенести тройку в конец,
    # нужно сделать транспонирование
    ax.imshow(example[0].numpy().transpose((1, 2, 0)))
    ax.set_title(f'{example[1]}', size=15)
plt.tight_layout()
plt.show()
```

Опять же картинку вставлять не буду, на ней должны быть те же 6 изображений, только теперь они одинакового размера (3, 80, 120) и имеют метки 0 или 1. Дальше можно аналогично сделать перемешивание и разделение на batches с помощью data loader. Теперь посмотрим как использовать уже доступные датасеты из библиотеки `torchvision.datasets`. Библиотека `torchvision.datasets` представляет хорошую коллекцию датасетов изображений, аналогично есть похожая библиотека `torchtext.datasets`, которая содержит датасеты для NLP. Все доступные датасеты есть по [ссылке](#). Эти датасеты хорошо форматированы, идут с большим описанием, включая формат фич и меток, их тип и размерности. Также плюс в том, что все эти классы наследуются от класса `torch.utils.data.Dataset`. Мы будем использовать два датасета: CelebA (`celeb_a`) и MNIST. Начнём работать с первым. Если мы пройдём по ссылке и прочитаем про этот датасет, то мы узнаем много информации. Датасет делится на три части: "train", "valid" и "test", изображения хранятся в формате `PIL.Image`. Можно получить трансформированную версию, используя свою `transform` функцию (`ToTensor`, `Resize` и т.д.). Есть разные targets: "attributes" (40 атрибутов лица, например, выражение лица, причёска или макияж), "identity" (id человека) и "landmarks" (позиция глаз, носа и т.д.). Теперь скачаем данные, сохраним их на диске и загрузим в виде `Dataset` объекта:

```
import torchvision
image_path = './'
celeba_dataset = torchvision.datasets.CelebA(
    image_path, split='train', target_type='attr', download=True
)
```

Может случиться ошибка `BadZipFile` или `RuntimeError`, это ограничение Google Drive, которое можно обойти, только подождав некоторое время. Конечно можно скачать архив явно руками. Проверим родителя нашего объекта:

```
assert isinstance(celeba_dataset, torch.utils.data.Dataset)
```

Посмотрим на данные:

```
# iter возвращает итератор
example = next(iter(celeba_dataset))
print(example)
# (<PIL.JpegImagePlugin.JpegImageFile ...>, tensor([0, ..., 0, 1]))
```

Если мы хотим сделать supervised deep learning модель, то нам нужно сделать тензор фич, а также метки, в качестве метки возьмём "Smiling" категорию (31 позиция). Визуализируем:

```
from itertools import islice
fig = plt.figure(figsize=(12, 8))
# делаем срез - islice(iterator, start, stop[, step])
for i, (image, attributes) in islice(enumerate(celeba_dataset), 18):
    ax = fig.add_subplot(3, 6, i+1)
```

```

ax.set_xticks([]); ax.set_yticks([])
ax.imshow(image)
ax.set_title(f'{attributes[31]}', size=15)
plt.show()

```

Картинку также вставлять не буду, на ней должны быть лица знаменитостей, а также сверху должна быть метка (0 или 1), которая говорит, улыбается человек или нет. Сделаем тоже самое с MNIST датасетом:

```

mnist_dataset = torchvision.datasets.MNIST(
    image_path, 'train', download=True)
assert isinstance(mnist_dataset, torch.utils.data.Dataset)
example = next(iter(mnist_dataset))
print(example)
# (<PIL.Image image mode=L size=28x28 at 0x7FBA4E4734D0>, 5)
fig = plt.figure(figsize=(15, 6))
for i, (image, label) in enumerate(mnist_dataset, 10):
    ax = fig.add_subplot(2, 5, i+1)
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(image, cmap='gray_r')
    ax.set_title(f'{label}', size=15)
plt.show()

```

Аналогично без изображения, тут просто цифры.

12.4 Building an NN model in PyTorch

Мы начнём с обучения простой модели линейной регрессии, так как PyTorch более гибкий, но в тоже время более сложный для изучения. `torch.nn` – это крутой модуль для создания и тренировки NNs. Сначала мы напишем простую лин. рег. без фич из `torch.nn`, а потом будем постепенно добавлять фичи из этого и `torch.optim` модулей. Самый частый подход – это использовать `nn.Module` модуль, который даёт больше контроля при forward проходе, т.к. можно выбирать слои. Наконец-то приступим к написанию регрессии, создадим датасет:

```

import numpy as np
import matplotlib.pyplot as plt
X_train = np.arange(10, dtype='float32').reshape((10, 1))
y_train = np.array([1.0, 1.3, 3.1, 2.0, 5.0, 6.3, 6.6,
                   7.4, 8.0, 9.0], dtype='float32')
plt.plot(X_train, y_train, 'o', markersize=10)
plt.xlabel('x')
plt.ylabel('y')
plt.show()

```

Опять без изображения, должна быть практически прямая с небольшими отклонениями. Стандартизуем фичи и сохраним в формате PyTorch:

```

import torch
from torch.utils.data import TensorDataset, DataLoader
X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
X_train_norm = torch.from_numpy(X_train_norm)
y_train = torch.from_numpy(y_train)
train_ds = TensorDataset(X_train_norm, y_train)
batch_size = 1
train_dl = DataLoader(train_ds, batch_size, shuffle=True)

```

Объявим модель в виде $z = wx + b$, мы будем использовать слои из `torch.nn`, но для начала научимся делать модель с нуля. Создадим параметры модели `weight` и `bias`, а затем объявим функцию `model()`, которая будет делать выход из входа:

```

torch.manual_seed(1)
weight = torch.randn(1)
weight.requires_grad_()
bias = torch.zeros(1, requires_grad=True)

def model(xb):
    # в контексте умножения матриц a @ b
    # вызывает a._matmul_(b)
    return xb @ weight + bias

def loss_fn(input, target):
    return (input - target).pow(2).mean()

```

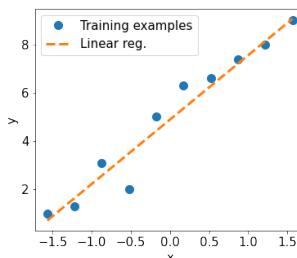
Далее, чтобы обучать параметры модели, мы будем использовать SGD. Сейчас мы напишем обучение сами, но далее будем использовать SGD метод из пакета `torch.optim`. Чтобы реализовать SGD, нам нужно считать градиенты. Мы будем использовать `torch.autograd.backward` функцию,

чтобы не считать градиенты руками. Мы обсудим `torch.autograd` и его классы и функции для реализации a.d. в главе 13. Код для тренировки модели с датасетом из batches:

```
learning_rate = 0.001
num_epochs = 200
log_epochs = 10
for epoch in range(num_epochs):
    for x_batch, y_batch in train_dl:
        pred = model(x_batch)
        loss = loss_fn(pred, y_batch)
        loss.backward()
        with torch.no_grad():
            weight -= weight.grad * learning_rate
            bias -= bias.grad * learning_rate
            weight.grad.zero_()
            bias.grad.zero_()
        if epoch % log_epochs==0:
            print(f'Epoch {epoch} Loss {loss.item():.4f}')
# Epoch 0 Loss 46.5720
# ...
# Epoch 190 Loss 0.0012
```

Протестируем модель и нарисуем график:

```
print('Final Parameters:', weight.item(), bias.item()) # 2.6 4.8
X_test = np.linspace(0, 9, num=100, dtype='float32').reshape(-1, 1)
X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
X_test_norm = torch.from_numpy(X_test_norm)
# detach:
# Returns a new Tensor, detached from the current graph.
y_pred = model(X_test_norm).detach().numpy()
fig = plt.figure(figsize=(13, 5))
# чтобы график не был сильно вытянут по оси x
ax = fig.add_subplot(1, 2, 1)
plt.plot(X_train_norm, y_train, 'o', markersize=10)
plt.plot(X_test_norm, y_pred, '--', lw=3)
plt.legend(['Training examples', 'Linear reg.'], fontsize=15)
ax.set_xlabel('x', size=15)
ax.set_ylabel('y', size=15)
ax.tick_params(axis='both', which='major', labelsize=15)
plt.show()
```



Мы написали тренировку модели с кастомной loss функцией `loss_fn()`, однако написание этих функций и обновление градиентов – это частая задача. `torch.nn` содержит множество loss функций, а `torch.optim` поддерживает большинство алгоритмов оптимизации для обновления параметров на основе градиентов. Чтобы лучше это понять, посмотрим на пример:

```
import torch.nn as nn
loss_fn = nn.MSELoss(reduction='mean')
input_size = 1
output_size = 1
model = nn.Linear(input_size, output_size)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Теперь обучим модель и сравним результаты:

```
for epoch in range(num_epochs):
    for x_batch, y_batch in train_dl:
        pred = model(x_batch)[:, 0]
        loss = loss_fn(pred, y_batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
    if epoch % log_epochs==0:
        print(f'Epoch {epoch} Loss {loss.item():.4f}')
print('Final Parameters:', model.weight.item(), model.bias.item())
# Final Parameters: 2.6496422290802 4.87706995010376
```

Теперь построим многослойный персепtron для Iris датасета. Мы будем классифицировать три вида цветов, используя двухслойную модель. Сначала получим данные:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
iris = load_iris()
X = iris['data']
y = iris['target']
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=1./3, random_state=1)

X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
X_train_norm = torch.from_numpy(X_train_norm).float()
y_train = torch.from_numpy(y_train)
train_ds = TensorDataset(X_train_norm, y_train)
torch.manual_seed(1)
batch_size = 2
train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

С помощью класса `nn.Module` можно делать много слоёв, все слои можно найти по [ссылке](#). Для нашей задачи мы будем использовать `Linear` слой, он также известен как fully connected layer или dense layer, он представлен как $f(w \times x + b)$, где f – функция активации. Каждый слой в NN получает входы с предыдущего слоя, т.к. что его ранг постоянен. У нас будет два скрытых слоя. Первый скрытый слой получает 4 фичи и выдаёт 16 нейронов. Второй слой получает эти 16 нейронов и проецирует их на 3 выходных нейрона, т.к. у нас 3 метки. См. код:

```
# nn.Module – Base class for all neural network modules.
class Model(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.layer1(x)
        x = nn.Sigmoid()(x)
        x = self.layer2(x)
        # dim=1 – для каждого ряда, 0 – для каждой колонки
        x = nn.Softmax(dim=1)(x)
        return x

input_size = X_train_norm.shape[1]
hidden_size = 16
output_size = 3
model = Model(input_size, hidden_size, output_size)
```

Здесь мы использовали `nn.Softmax()` функцию, она приводит каждый элемент в отрезок $[0, 1]$, при этом сумма по указанной размерности равна 1. А вот и формула:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Эта функция позволяет поддерживать multiclass классификацию, разные функции активации мы обсудим позже в этой главе. Сейчас нам надо объявить loss функцию – это будет cross-entropy loss, а также оптимизатор – это будет Adam, он основан также на градиентах и он очень надёжный (обсудим в главе 14). См. код:

```
learning_rate = 0.001
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

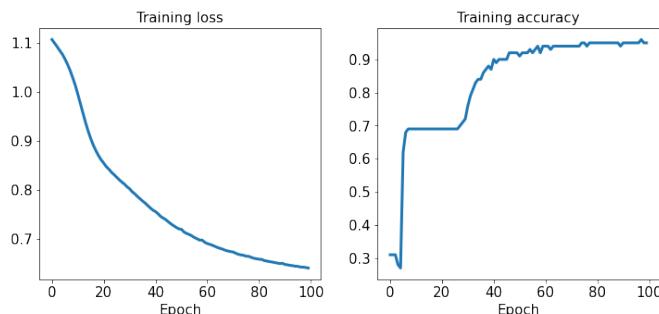
num_epochs = 100
loss_hist = [0] * num_epochs
accuracy_hist = [0] * num_epochs
for epoch in range(num_epochs):
    for x_batch, y_batch in train_dl:
        pred = model(x_batch)
        loss = loss_fn(pred, y_batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        loss_hist[epoch] += loss.item() * y_batch.size(0)
        is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
        accuracy_hist[epoch] += is_correct.sum()
    loss_hist[epoch] /= len(train_dl.dataset)
    # кол-во всех примеров а не кол-во батчей
    accuracy_hist[epoch] /= len(train_dl.dataset)
```

Теперь можно нарисовать learning curves:

```
fig = plt.figure(figsize=(12, 5))

ax = fig.add_subplot(1, 2, 1)
ax.plot(loss_hist, lw=3)
ax.set_title('Training loss', size=15)
ax.set_xlabel('Epoch', size=15)
ax.tick_params(axis='both', which='major', labelsize=15)

ax = fig.add_subplot(1, 2, 2)
ax.plot(accuracy_hist, lw=3)
ax.set_title('Training accuracy', size=15)
ax.set_xlabel('Epoch', size=15)
ax.tick_params(axis='both', which='major', labelsize=15)
plt.show()
```



Теперь посмотрим качество модели на тестовом датасете:

```
X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
X_test_norm = torch.from_numpy(X_test_norm).float()
y_test = torch.from_numpy(y_test)
pred_test = model(X_test_norm)
correct = (torch.argmax(pred_test, dim=1) == y_test).float()
accuracy = correct.mean()
print(f'Test Acc.: {accuracy:.4f}')
# Test Acc.: 0.9800
```

Научимся сохранять и загружать тренированную модель. Модель можно сохранить на диск для использования в будущем. См. код:

```
path = 'iris_classifier.pt'
torch.save(model, path)
```

Мы сохранили и архитектуру модели, и все обученные параметры. Мы можем сохранять модель с расширением `pt` или `pth`, хотя это просто соглашение. А теперь загрузим модель:

```
model_new = torch.load(path)
```

Посмотрим на архитектуру модели:

```
model_new.eval()
# Model(
#   (layer1): Linear(in_features=4, out_features=16, bias=True)
#   (layer2): Linear(in_features=16, out_features=3, bias=True)
# )
```

Посчитаем результат модели на тестовых данных, чтобы убедиться в правильности наших действий:

```
pred_test = model_new(X_test_norm)
correct = (torch.argmax(pred_test, dim=1) == y_test).float()
accuracy = correct.mean()
print(f'Test Acc.: {accuracy:.4f}') # 0.9800
```

Давайте попробуем сохранить только обученные параметры:

```
path = 'iris_classifier_state.pth'
torch.save(model.state_dict(), path)

model_new = Model(input_size, hidden_size, output_size)
model_new.load_state_dict(torch.load(path))
```

12.5 Choosing activation functions for multilayer NNs

Для простоты мы использовали только функцию sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$. Технически мы можем использовать любую дифференцируемую функцию, можно использовать даже линейную функцию, но это не даст хороших результатов, т.к. мы не сможем сделать нелинейные зависимости, чтобы справиться со сложной задачей. Logistic (sigmoid) функция активации наиболее похожа на настоящий нейрон, мы можем думать о ней как о вероятности срабатывания нейрона. Однако такая функция может быть плоха, если мы имеем сильно отрицательный вход, т.к. тогда мы получим значения очень близкие к нулю. Значения около нуля приведут к медленному обучению, также вероятно, что алгоритм может попасть в локальный минимум. Именно поэтому люди часто предпочитают **hyperbolic tangent** функцию в скрытых слоях. Перед тем, как на неё посмотреть, давайте вспомним logistic функцию и обсудим её обобщение на несколько меток. Вспомним пару формул:

$$z = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^m w_i x_i = w^T x$$

$$\sigma_{\text{logistic}}(z) = \frac{1}{1 + e^{-z}}$$

Заметим, что w_0 – это bias unit (пересечение с осью y), т.е. $x_0 = 1$. Приведём конкретный пример:

```
import numpy as np
X = np.array([1, 1.4, 2.5])
w = np.array([0.4, 0.3, 0.5])
def net_input(X, w):
    return np.dot(X, w)
def logistic(z):
    return 1.0 / (1.0 + np.exp(-z))
def logistic_activation(X, w):
    z = net_input(X, w)
    return logistic(z)
print(f'P(y=1|x) = {logistic_activation(X, w):.3f}')
# P(y=1/x) = 0.888
```

В главе 11 мы прошли one-hot encoding технику и сделали выходной слой из нескольких logistic activation units. Однако, как покажет следующий код, такой выходной слой не даёт адекватные значения вероятностей. А вот и код:

```
# (n_output_units, n_hidden_units+1)
# первая колонка – это bias units
W = np.array([[1.1, 1.2, 0.8, 0.4],
              [0.2, 0.4, 1.0, 0.2],
              [0.6, 1.5, 1.2, 0.7]])
# (n_samples, n_hidden_units + 1)
A = np.array([[1, 0.1, 0.4, 0.6]])
Z = np.dot(W, A[0])
y_probas = logistic(Z)
print('Net Input: \n', Z) # [1.78 0.76 1.65]
print('Output Units:\n', y_probas)
# [0.85569687 0.68135373 0.83889105]
```

Данные значения нельзя воспринимать как значения вероятностей для трёх классов, т.к. сумма не равна 1. Однако это не проблема, если нам нужна просто метка, а не её вероятность:

```
y_class = np.argmax(Z, axis=0)
print('Predicted class label:', y_class) # 0
```

Однако иногда бывает важно знать и вероятности в multiclass случае, давайте посмотрим на обобщение logistic функции – softmax функция – это soft форма argmax функции, multinomial logistic regression. А вот и формула:

$$p(z) = \sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^M e^{z_j}}$$

Напишем код:

```
def softmax(z):
    return np.exp(z) / np.sum(np.exp(z))
y_probas = softmax(Z)
print('Probabilities:\n', y_probas)
# [0.44668973 0.16107406 0.39223621]
print(np.sum(y_probas)) # 1.0
```

Посмотрим на эту функцию в PyTorch:

```
torch.softmax(torch.from_numpy(Z), dim=0)
# tensor([0.4467, 0.1611, 0.3922], dtype=torch.float64)
```

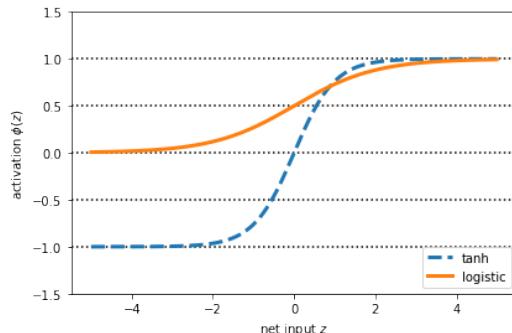
Рассмотрим расширение выходного спектра с помощью **hyperbolic tangent** (**tanh**) – другая популярная sigmoidal функция для скрытых слоёв, которую можно понимать как масштабированную версию logistic функции:

$$\sigma_{\text{logistic}}(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma_{\tanh}(z) = 2 \times \sigma_{\text{logistic}}(2z) - 1 = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Её преимущество в том, что она имеет более широкий спектр выходных данных в диапазоне $(-1, 1)$, что может улучшить сходимость backpropagation алгоритма. Для сравнения нарисуем две эти функции:

```
import matplotlib.pyplot as plt
def tanh(z):
    e_p = np.exp(z)
    e_m = np.exp(-z)
    return (e_p - e_m) / (e_p + e_m)
z = np.arange(-5, 5, 0.005)
log_act = logistic(z)
tanh_act = tanh(z)
plt.ylim([-1.5, 1.5])
plt.xlabel('net input $z$')
plt.ylabel('activation $\phi(z)$')
[plt.axhline(x, color='black', linestyle=':')]
for x in [1, 0.5, 0, -0.5, -1]:
    plt.plot(z, tanh_act, linewidth=3,
              linestyle='--', label='tanh')
    plt.plot(z, log_act, linewidth=3,
              label='logistic')
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()
```



Как мы видим, форма двух sigmoidal функций очень похожа, однако наша функция имеет выход в два раза больше. На практике нужно конечно использовать функцию **tanh** из NumPy или **torch.tanh(x)**:

```
print(np.tanh(z))
print(torch.tanh(torch.from_numpy(z)))
```

Аналогично про logistic функцию:

```
from scipy.special import expit
print(expit(z))
print(torch.sigmoid(torch.from_numpy(z)))
```

Заметим, что **torch.sigmoid(x)** даёт такой же результат как и **torch.nn.Sigmoid()(x)**, однако в ней можно контролировать поведение через параметры. Поговорим про активацию **rectified linear unit (ReLU)**, она часто используется в deep NNs. Перед тем, как углубиться в неё, мы должны понять *vanishing gradient* проблему в tanh и logistic активациях. Пусть мы имеем net input $z_1 = 20$, который меняется в $z_2 = 25$, tanh активация даёт $\sigma(z_1) = 1.0$ и $\sigma(z_2) = 1.0$, что не показывает разницы в выходе (из-за асимптотического поведения tanh и численных ошибок). Всё это значит, что производная активации по отношению к net input уменьшается с ростом z , из-за этого обучение становится медленным, т.к. градиенты становятся близкими к нулю. ReLU это решает, вот и формула:

$$\sigma(z) = \max(0, z)$$

ReLU всё ещё нелинейная, поэтому она подходит для сложных функций при обучении NNs. Кроме того, производная по отношению к положительному входу всегда равна 1, что решает проблему исчезновения градиента. Применим её в коде:

```
torch.relu(torch.from_numpy(z))
```

Мы будем использовать эту функцию в следующей главе для multilayer convolutional NNs. Посмотрим на картинку со всеми функциями активации, которые мы использовали в этом конспекте:

| Activation function | Equation | Example | 1D graph |
|--------------------------------|---|------------------------------------|----------|
| Linear | $\sigma(z) = z$ | Adaline, linear regression | |
| Unit step (Heaviside function) | $\sigma(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$ | Perceptron variant | |
| Sign (sigmoid) | $\sigma(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$ | Perceptron variant | |
| Piece-wise linear | $\sigma(z) = \begin{cases} 0 & z \leq -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \leq z \leq \frac{1}{2} \\ 1 & z \geq \frac{1}{2} \end{cases}$ | Support vector machine | |
| Logistic (sigmoid) | $\sigma(z) = \frac{1}{1 + e^{-z}}$ | Logistic regression, multilayer NN | |
| Hyperbolic tangent (tanh) | $\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ | Multilayer NN, RNNs | |
| ReLU | $\sigma(z) = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}$ | Multilayer NN, CNNs | |

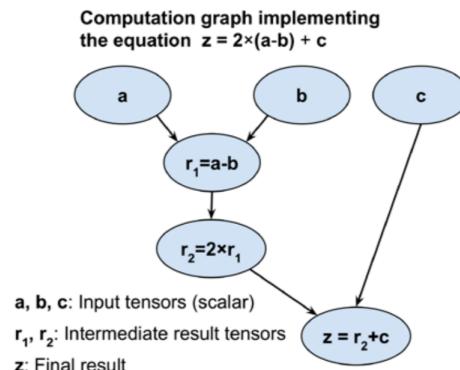
Все функции активации, доступные в модуле `torch.nn`, можно найти по [ссылке](#). В следующей главе мы углубимся в PyTorch, мы будем работать с PyTorch computation graphs и пакетом для automatic differentiation, также мы узнаем новые концепции gradient computations.

13 The Mechanics of PyTorch

Чтобы показать разные способы построения моделей с помощью `torch.nn` модуля, мы рассмотрим классическую задачу **exclusive or (XOR)**. Мы посмотрим на два класса: `Sequential` и `nn.Module`. Давайте немного поговорим про ключевые фичи PyTorch. Этот framework использует dynamic computational graphs, которые имеют преимущество в гибкости по сравнению со статическими аналогами, также их можно отлаживать – можно чередовать объявление графа и шаги его вычисления, можно запускать код построчно, имея доступ ко всем переменным. PyTorch поддерживает mobile deployment, что делает его очень подходящим для промышленности. В следующей части мы посмотрим как тензор и функция взаимосвязаны через computation graph.

13.1 PyTorch's computation graphs

PyTorch делает свои вычисления на основе **directed acyclic graph (DAG)**. В этой части мы посмотрим как можно объявить эти графы для вычисления простой арифметической операции, дальше мы посмотрим на dynamic graph парадигму. Пусть мы имеем тензоры ранга 0 – a , b и c , мы хотим вычислить $z = 2 \times (a - b) + c$. См. картинку:



Чтобы создать график, можно объявить обычную Python функцию:

```

import torch
def compute_z(a, b, c):
    r1 = torch.sub(a, b)
    r2 = torch.mul(r1, 2)
    z = torch.add(r2, c)
    return z

```

Посмотрим как сделать вычисления:

```

print('Scalar Inputs:', compute_z(torch.tensor(1),
                                   torch.tensor(2), torch.tensor(3)))
print('Rank 1 Inputs:', compute_z(torch.tensor([1]),
                                   torch.tensor([2]), torch.tensor([3])))
print('Rank 2 Inputs:', compute_z(torch.tensor([[1]]),
                                   torch.tensor([[2]]), torch.tensor([[3]])))
# tensor(1), tensor([1]), tensor([[1]])

```

13.2 Tensors for storing and updating model parameters

Специальный тензор, для которого нужно считать градиенты, позволяет хранить и обновлять параметры нашей модели во время тренировки. Такой тензор можно создать, написав параметр `requires_grad=True`, градиент могут требовать только тензор с complex или float point dtype. См. пример кода:

```

a = torch.tensor(3.14, requires_grad=True)
print(a) # tensor(3.1400, requires_grad=True)
b = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
print(b) # tensor([1., 2., 3.], requires_grad=True)

```

По умолчанию `requires_grad=False`, но его можно изменить на `True`, запустив `requires_grad_()`. Посмотрим на пример:

```

w = torch.tensor([1.0, 2.0, 3.0])
print(w.requires_grad) # False
w.requires_grad_()
print(w.requires_grad) # True

```

Для NNs важно присваивать случайные веса, чтобы ломать симметрию в backpropagation, иначе от многослойной сети будет столько же пользы, сколько от logistic regression. Можно генерировать случайные тензоры, случайные числа в них получаются на основе разных probability distributions ([ссылка](#)). В следующем примере мы посмотрим на несколько стандартных методов инициализации, которые доступны в `torch.nn.init` модуле ([ссылка](#)). Посмотрим как создать тензор с **Glorot** инициализацией:

```

import torch.nn as nn
torch.manual_seed(1)
w = torch.empty(2, 3)
nn.init.xavier_normal_(w)
print(w)
# tensor([[ 0.4183,  0.1688,  0.0390],
#         [-0.3930, -0.2858, -0.1051]])

```

Поговорим немного про Xavier (или Glorot) инициализацию. На ранних этапах deep learning заметили, что random normal weight initialization часто приводит к плохим результатам. Общая идея в том, чтобы примерно сбалансировать дисперсию градиентов в разных слоях. То есть каким-то слоям уделяется больше внимания во время обучения, чем другим. Если мы хотим присвоить веса с помощью uniform distribution, ему нужно выбрать интервал вот так:

$$W \sim Uniform\left(-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right)$$

Здесь n_{in} – кол-во входных нейронов, а n_{out} – кол-во выходных нейронов. Если мы присваиваем веса с помощью Gaussian (normal) distribution, то нужно выбрать std вот такое:

$$\sigma = \frac{\sqrt{2}}{\sqrt{n_{in} + n_{out}}}$$

PyTorch поддерживает Xavier инициализацию и в uniform, и в normal распределениях весов. А вот и пример кода:

```

class MyModule(nn.Module):
    def __init__(self):
        super().__init__()

```

```

self.w1 = torch.empty(2, 3, requires_grad=True)
nn.init.xavier_normal_(self.w1)
self.w2 = torch.empty(1, 2, requires_grad=True)
nn.init.xavier_normal_(self.w2)

```

13.3 Computing gradients via automatic differentiation

По мимо того, что градиенты нужны для алгоритмов оптимизации, они иногда нужны, чтобы понять почему сеть делает определённое предсказание для какого-то примера. Поговорим про подсчёт градиентов для функции потерь по отношению к train variables. PyTorch поддерживает *automatic differentiation*, о нём можно думать как о применении *chain rule* для вложенных функций. Мы будем называть градиентами и градиенты ($\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots\right)$), и частные производные (например, $\frac{\partial f}{\partial x_1}$ у функции $f(x_1, x_2, \dots)$). Когда мы объявляем серию операций, которая приводит к какому-то выводу или промежуточному тензору, PyTorch обеспечивает подсчёт градиентов этих тензоров по отношению к начальным вершинам в computation graph. Чтобы их посчитать, нужно вызвать `backward` метод из `torch.autograd` модуля, он считает сумму градиентов данного тензора по отношению к листьям графа. Посмотрим на простой пример, где $z = wx + b$, а $Loss = (y - z)^2$. См. код:

```

w = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(0.5, requires_grad=True)
x = torch.tensor([1.4])
y = torch.tensor([2.1])
z = torch.add(torch.mul(w, x), b)
loss = (y - z).pow(2).sum()
loss.backward()
print('dL/dw : ', w.grad) # tensor(-0.5600)
print('dL/db : ', b.grad) # tensor(-0.4000)

```

Давайте проверим результат руками и посчитаем значение $\frac{\partial Loss}{\partial w} = 2x(wx + b - y)$:

```

print(2 * x * ((w * x + b) - y))
# tensor([-0.5600], grad_fn=<MulBackward0>)

```

Для лучшего понимания посмотрим на пример. Пусть $y = f(g(h(x)))$, тогда мы имеем 4 шага: $u_0 = x$, $u_1 = h(x)$, $u_2 = g(u_1)$, $u_3 = f(u_2) = y$. Производную $\frac{dy}{dx}$ можно посчитать двумя способами. Первый способ – это forward accumulation, $\frac{du_3}{dx} = \frac{du_3}{du_2} \frac{du_2}{du_0}$. Второй способ – reverse accumulation, PyTorch его как раз и использует (т.к. эффективный для backpropagation), $\frac{dy}{du_0} = \frac{dy}{du_1} \frac{du_1}{du_0}$. Подсчёт градиентов используется для генерации *adversarial examples* (или *adversarial attacks*). В computer vision, adversarial examples – это примеры, которые генерируются добавлением небольшого незаметного шума (возмущения) к входным данным так, что deep NNs неправильно их классифицируют. Эта тема вне конспекта, но её можете почитать сами.

13.4 Simplifying impls of common architectures via torch.nn

Перед тем как углубиться в `nn.Module`, давайте кратко глянем на другой подход для соединения этих слоёв через `nn.Sequential` ([ссылка](#)). В этом подходе слои соединены каскадным образом, посмотрим на пример с двумя densely (fully) connected слоями:

```

model = nn.Sequential(
    nn.Linear(4, 16),
    nn.ReLU(),
    nn.Linear(16, 32),
    nn.ReLU()
)
print(model)
# Sequential(
#   (0): Linear(in_features=4, out_features=16, bias=True)
#   (1): ReLU()
#   (2): Linear(in_features=16, out_features=32, bias=True)
#   (3): ReLU()
# )

```

Можно дальше конфигурировать эти слои, вот список того, что можно ещё добавить:

- Выбор функции активации ([ссылка](#)).
- Инициализация параметров слоёв через `nn.init` ([ссылка](#)).
- Применение L2 регуляризации к параметрам слоёв через параметр `weight_decay` из модуля `torch.optim` ([ссылка](#)).

- Применение L1 регуляризации, добавлением L1 члена к loss тензору, это реализуем дальше.

В следующем примере кода мы специфицируем распределение весов для первого слоя и добавим L1 член для весовой матрицы второго слоя:

```
nn.init.xavier_uniform_(model[0].weight)
l1_weight = 0.01
l1_penalty = l1_weight * model[2].weight.abs().sum()
```

Дальше можно выбрать `optimizer` (через `torch.optim`) и `loss` функцию. Теперь посмотрим как можно выбрать loss функцию. SGD и Adam – это два самых используемых алгоритма, а вот выбор функции потерь зависит от задачи, например, для регрессии нужно использовать MSE. Семейство cross-entropy loss функций применяется для задач классификации, которые мы рассмотрим в главе 14. Напишем две строчки кода, где будем использовать cross-entropy loss для бинарной классификации:

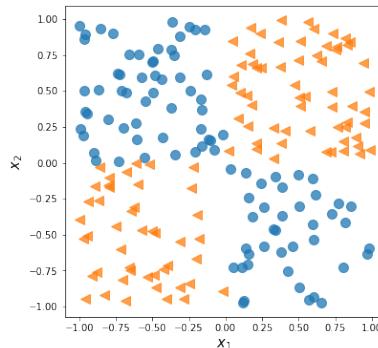
```
loss_fn = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Теперь посмотрим на более практическую задачу – задача XOR классификации – это классическая задача для анализа возможностей модели по улавливанию нелинейных границ. Мы сгенерируем датасет из 200 примеров с двумя фичами (x_0, x_1) из uniform distribution в полуинтервале $[-1, 1]$. А вот метки мы присвоим вот так:

$$y^{(i)} = \begin{cases} 0 & \text{if } x_0^{(i)} \times x_1^{(i)} < 0 \\ 1 & \text{otherwise} \end{cases}$$

Также мы разделим данные на train и validation, вот и код:

```
import matplotlib.pyplot as plt
import numpy as np
torch.manual_seed(1)
np.random.seed(1)
x = np.random.uniform(low=-1, high=1, size=(200, 2))
y = np.ones(len(x))
y[x[:, 0] * x[:, 1] < 0] = 0
n_train = 100
x_train = torch.tensor(x[:n_train], dtype=torch.float32)
y_train = torch.tensor(y[:n_train], dtype=torch.float32)
x_valid = torch.tensor(x[n_train:], dtype=torch.float32)
y_valid = torch.tensor(y[n_train:], dtype=torch.float32)
fig = plt.figure(figsize=(6, 6))
plt.plot(x[y == 0, 0], x[y == 0, 1], 'o', alpha=0.75, markersize=10)
plt.plot(x[y == 1, 0], x[y == 1, 1], '<', alpha=0.75, markersize=10)
plt.xlabel(r'$x_1$', size=15)
plt.ylabel(r'$x_2$', size=15)
plt.show()
```



Общее правило в том, что чем больше слоёв и нейронов в модели, тем лучше она приближает сложные данные, с другой стороны обучение идёт дольше и шанс переобучиться выше. На практике лучше начинать с простой модели, например, с однослоевой NN, logistic regression:

```
model = nn.Sequential(
    nn.Linear(2, 1),
    nn.Sigmoid()
)
model
```

Теперь объявим функцию потерь, оптимизатор и data loader:

```

loss_fn = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

from torch.utils.data import DataLoader, TensorDataset
train_ds = TensorDataset(x_train, y_train)
batch_size = 2
torch.manual_seed(1)
train_dl = DataLoader(train_ds, batch_size, shuffle=True)

```

Напишем тренировку модели:

```

torch.manual_seed(1)
num_epochs = 200
def train(model, num_epochs, train_dl, x_valid, y_valid):
    loss_hist_train = [0] * num_epochs
    accuracy_hist_train = [0] * num_epochs
    loss_hist_valid = [0] * num_epochs
    accuracy_hist_valid = [0] * num_epochs
    for epoch in range(num_epochs):
        for x_batch, y_batch in train_dl:
            pred = model(x_batch)[:, 0]
            loss = loss_fn(pred, y_batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            loss_hist_train[epoch] += loss.item()
            is_correct = ((pred >= 0.5).float() == y_batch).float()
            accuracy_hist_train[epoch] += is_correct.mean()
        loss_hist_train[epoch] /= n_train/batch_size
        accuracy_hist_train[epoch] /= n_train/batch_size
        pred = model(x_valid)[:, 0]
        loss = loss_fn(pred, y_valid)
        loss_hist_valid[epoch] = loss.item()
        is_correct = ((pred >= 0.5).float() == y_valid).float()
        accuracy_hist_valid[epoch] = is_correct.mean()
    return loss_hist_train, loss_hist_valid,
           accuracy_hist_train, accuracy_hist_valid

history = train(model, num_epochs, train_dl, x_valid, y_valid)

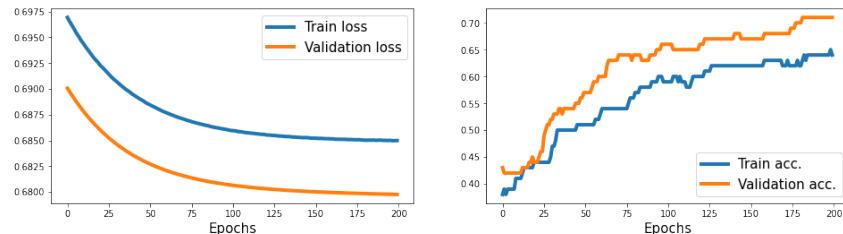
```

Напишем код для построения learning curves:

```

fig = plt.figure(figsize=(16, 4))
ax = fig.add_subplot(1, 2, 1)
plt.plot(history[0], lw=4)
plt.plot(history[1], lw=4)
plt.legend(['Train loss', 'Validation loss'], fontsize=15)
ax.set_xlabel('Epochs', size=15)
ax = fig.add_subplot(1, 2, 2)
plt.plot(history[2], lw=4)
plt.plot(history[3], lw=4)
plt.legend(['Train acc.', 'Validation acc.'], fontsize=15)
ax.set_xlabel('Epochs', size=15)

```



Отсутствие скрытых слоёв дало только линейную границу, а она очевидно нам не подходит. Нужно добавить один или несколько скрытых слоёв, соединённых через нелинейную функцию активации. Теорема об универсальной аппроксимации утверждает, что feedforward NN с одним скрытым слоем и относительно большим количеством hidden units может относительно хорошо аппроксимировать произвольные непрерывные функции. Так что можно добавить один скрытый слой и менять число нейронов в нём пока результаты не станут хорошими. С другой стороны можно увеличить число слоёв, это лучше т.к. нам нужно меньше параметров, чтобы достичь тех же результатов. Однако и такой подход имеет минусы, такие модели имеют проблемы vanishing и exploding градиентов, из-за чего их сложнее тренировать. Посмотрим на результаты с двумя скрытыми слоями:

```

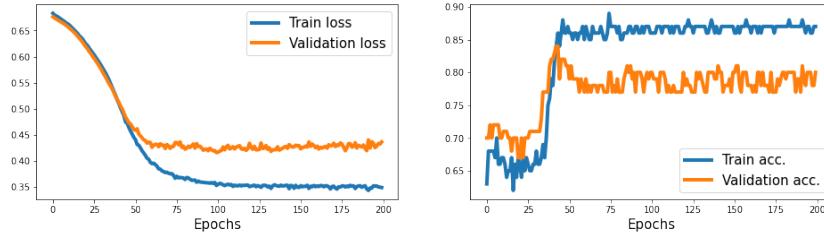
model = nn.Sequential(
    nn.Linear(2, 4),

```

```

        nn.ReLU(),
        nn.Linear(4, 4),
        nn.ReLU(),
        nn.Linear(4, 1),
        nn.Sigmoid()
    )
loss_fn = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
model

```



Рассмотрим гибкое построение моделей с помощью `nn.Module`. Мы использовали `Sequential` класс, но он не позволяет создавать сложные модели с несколькими входами и выходами или промежуточными ответвлениями, поэтому есть `nn.Module`. Код, который аналогичен выше написанному:

```

class MyModule(nn.Module):
    def __init__(self):
        super().__init__()
        l1 = nn.Linear(2, 10)
        a1 = nn.ReLU()
        l2 = nn.Linear(10, 10)
        a2 = nn.ReLU()
        l3 = nn.Linear(10, 1)
        a3 = nn.Sigmoid()
        l = [l1, a1, l2, a2, l3, a3]
        self.module_list = nn.ModuleList(l)

    def forward(self, x):
        for f in self.module_list:
            x = f(x)
        return x

model = MyModule()
print(model)
# MyModule(
#   (module_list): ModuleList(
#     (0): Linear(in_features=2, out_features=4, bias=True)
#     ...
#     (5): Sigmoid()))
# )

loss_fn = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
history = train(model, num_epochs, train_dl, x_valid, y_valid)

```

Мы будем использовать библиотеку `mlxtend`, чтобы визуализировать decision boundary, её можно установить так: `pip install mlxtend`. Чтобы нарисовать границу, нам нужно добавить метод `predict()` в класс `MyModule`:

```

class MyModule(nn.Module):
    ...
    def predict(self, x):
        x = torch.tensor(x, dtype=torch.float32)
        pred = self.forward(x)[:, 0]
        return (pred >= 0.5).float()

```

Напишем код для вывода границы и посмотрим на неё:

```

from mlxtend.plotting import plot_decision_regions
fig = plt.figure(figsize=(16, 4))

ax = fig.add_subplot(1, 3, 1)
plt.plot(history[0], lw=4)
plt.plot(history[1], lw=4)
plt.legend(['Train loss', 'Validation loss'], fontsize=15)
ax.set_xlabel('Epochs', size=15)

ax = fig.add_subplot(1, 3, 2)
plt.plot(history[2], lw=4)

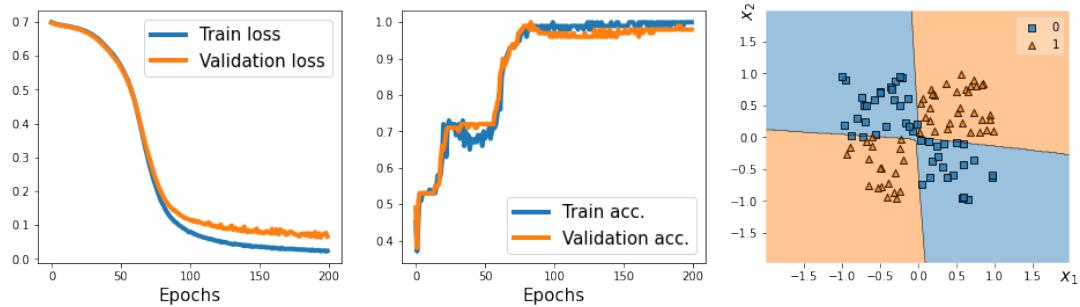
```

```

plt.plot(history[3], lw=4)
plt.legend(['Train acc.', 'Validation acc.'], fontsize=15)
ax.set_xlabel('Epochs', size=15)

ax = fig.add_subplot(1, 3, 3)
plot_decision_regions(X=x_valid.numpy(),
                      y=y_valid.numpy().astype(np.integer),
                      clf=model)
ax.set_xlabel(r'$x_1$', size=15)
ax.xaxis.set_label_coords(1, -0.025)
ax.set_ylabel(r'$x_2$', size=15)
ax.yaxis.set_label_coords(-0.025, 1)
plt.show()

```



Научимся писать кастомные слои в PyTorch. Если мы хотим создать свой слой (или изменить существующий), который ещё не поддерживается в PyTorch, то надо создать наследника от класса `nn.Module`. Например, мы хотим создать слой, который считает $w(x + \epsilon) + b$, где ϵ – это случайное число, переменная шума. В конструкторе мы создаём переменные и тензоры, которые нужны для нашего слоя. Правда, если нам не известно значение `input_size`, то инициализацию надо делать где-то в другом месте дальше. Напишем код:

```

class NoisyLinear(nn.Module):
    def __init__(self, input_size, output_size,
                 noise_stddev=0.1):
        super().__init__()
        w = torch.Tensor(input_size, output_size)
        # Это специальный тензор-параметр для наследников
        # nn.Module, он умеет чуть больше, чем просто тензор,
        # это можно прочитать в документации.
        self.w = nn.Parameter(w)
        nn.init.xavier_uniform_(self.w)
        b = torch.Tensor(output_size).fill_(0)
        self.b = nn.Parameter(b)
        self.noise_stddev = noise_stddev

    def forward(self, x, training=False):
        if training:
            noise = torch.normal(0.0, self.noise_stddev, x.shape)
            x_new = torch.add(x, noise)
        else:
            x_new = x
        return torch.add(torch.mm(x_new, self.w), self.b)

```

Есть некоторые методы, которые ведут себя по-разному для процесса тренировки и предсказаний, например, `Dropout`. Перед тем как вставить этот слой в нашу модель, давайте протестируем его на простом примере. Просто посмотрим на вывод слоя:

```

torch.manual_seed(1)
noisy_layer = NoisyLinear(4, 2)
x = torch.zeros((1, 4))
print(noisy_layer(x, training=True)) # [[ 0.1154, -0.0598]]
print(noisy_layer(x, training=True)) # [[ 0.0432, -0.0375]]
print(noisy_layer(x, training=False)) # [[0., 0.]]

```

Немного изменим код нашей модели:

```

class MyNoisyModule(nn.Module):
    def __init__(self):
        super().__init__()
        self.l1 = NoisyLinear(2, 10, 0.07)
        self.a1 = nn.ReLU()
        self.l2 = nn.Linear(10, 10)
        self.a2 = nn.ReLU()

```

```

    self.l3 = nn.Linear(10, 1)
    self.a3 = nn.Sigmoid()

def forward(self, x, training=False):
    x = self.l1(x, training)
    x = self.a3(self.l3(self.a2(self.l2(self.a1(x))))))
    return x

def predict(self, x):
    x = torch.tensor(x, dtype=torch.float32)
    pred = self.forward(x)[:, 0]
    return (pred >= 0.5).float()

torch.manual_seed(1)
model = MyNoisyModule()

```

Процесс тренировки такой же как и раньше за исключением одной строчки: `pred = model(x_batch, True)[:, 0]`.

13.5 Project one – predicting the fuel efficiency of a car

Мы будем предсказывать эффективность топлива в miles per gallon (MPG). Датасет Auto MPG, над которым мы будем работать, можно найти по [ссылке](#). Для начала сделаем стандартную подготовку данных:

```

import pandas as pd
url = 'http://archive.ics.uci.edu/ml/' \
      'machine-learning-databases/auto-mpg/auto-mpg.data'
column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower',
                'Weight', 'Acceleration', 'Model Year', 'Origin']

# na_values – набор символов, которые мы считаем пан
# sep – сепаратор, comment – комментарии (в нашем случае название авто)
# skipinitialspace – нужно ли пропускать пробелы после sep
df = pd.read_csv(url, names=column_names, na_values='?',
                  comment='\t', sep=' ', skipinitialspace=True)

df = df.dropna() # удаляем строки с NA
# This resets the index to the default integer index.
df = df.reset_index(drop=True)

import sklearn
import sklearn.model_selection
df_train, df_test = sklearn.model_selection.train_test_split(
    df, train_size=0.8, random_state=1
)
# выдаёт много статистической информации
train_stats = df_train.describe().transpose()
print(train_stats)
#           count      mean        std      min ...
# MPG          313.0    23.404153    7.666909     9.0 ...
# ...

numeric_column_names = ['Cylinders', 'Displacement', 'Horsepower',
                        'Weight', 'Acceleration']
df_train_norm, df_test_norm = df_train.copy(), df_test.copy()
for col_name in numeric_column_names:
    mean = train_stats.loc[col_name, 'mean']
    std = train_stats.loc[col_name, 'std']
    df_train_norm.loc[:, col_name] = \
        (df_train_norm.loc[:, col_name] - mean) / std
    df_test_norm.loc[:, col_name] = \
        (df_test_norm.loc[:, col_name] - mean) / std
print(df_train_norm.tail())

```

Давайте сложим параметр ModelYear в бакеты, чтобы облегчить задачу обучения модели. А вот и бакеты:

$$\text{bucket} = \begin{cases} 0 & \text{if year} < 73 \\ 1 & \text{if } 73 \leq \text{year} < 76 \\ 2 & \text{if } 76 \leq \text{year} < 79 \\ 3 & \text{if year} \geq 79 \end{cases}$$

Сначала мы сделаем три cut-off значения: [73, 76, 79], дальше мы воспользуемся функцией, которая выдаст индексы бакетов – `torch.bucketize`. См. код:

```
import torch
```

```

boundaries = torch.tensor([73, 76, 79])

v = torch.tensor(df_train_norm['Model Year'].values)
# right=True – находит последнее подходящее значение,
# т.е. мы делаем интервал справа, а не слева
df_train_norm['Model Year Bucketed'] = torch.bucketize(
    v, boundaries, right=True)
v = torch.tensor(df_test_norm['Model Year'].values)
df_test_norm['Model Year Bucketed'] = torch.bucketize(
    v, boundaries, right=True)

numeric_column_names.append('Model Year Bucketed')

```

Теперь поработаем с неупорядоченной категориальной фичей `Origin`. В PyTorch есть два способа с ними работать: использовать embedding layer через `nn.Embedding` или использовать уже знакомую технику one-hot-encoded векторы (indicators). Как работает второй способ мы уже знаем. Первый способ сопоставляет каждому числу вектор из случайных чисел типа `float` (на самом деле это более эффективная реализация one-hot encoding техники). Когда число категорий большое, использовать embedding слой с меньшим числом размерностей, чем кол-во категорий – это более эффективно. Напишем код:

```

from torch.nn.functional import one_hot
total_origin = len(set(df_train_norm['Origin']))

origin_encoded = one_hot(torch.from_numpy(
    df_train_norm['Origin'].values) % total_origin)
x_train_numeric = torch.tensor(
    df_train_norm[numeric_column_names].values)
x_train = torch.cat([x_train_numeric, origin_encoded], 1).float()

origin_encoded = one_hot(torch.from_numpy(
    df_test_norm['Origin'].values) % total_origin)
x_test_numeric = torch.tensor(
    df_test_norm[numeric_column_names].values)
x_test = torch.cat([x_test_numeric, origin_encoded], 1).float()

```

Осталось создать метки:

```

y_train = torch.tensor(df_train_norm['MPG'].values).float()
y_test = torch.tensor(df_test_norm['MPG'].values).float()

```

Теперь будем тренировать DNN regression модель. Создадим `DataLoader`:

```

from torch.utils.data import TensorDataset, DataLoader
train_ds = TensorDataset(x_train, y_train)
batch_size = 8
torch.manual_seed(1)
train_dl = DataLoader(train_ds, batch_size, shuffle=True)

```

Теперь сделаем два fully connected слоя с 8 и 4 hidden units:

```

import torch.nn as nn
hidden_units = [8, 4]
input_size = x_train.shape[1]
all_layers = []
for hidden_unit in hidden_units:
    layer = nn.Linear(input_size, hidden_unit)
    all_layers.append(layer)
    all_layers.append(nn.ReLU())
    input_size = hidden_unit
all_layers.append(nn.Linear(hidden_units[-1], 1))
model = nn.Sequential(*all_layers)
print(model)

```

Наконец-то напишем процедуру тренировки модели:

```

loss_fn = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

torch.manual_seed(1)
num_epochs = 200
log_epochs = 20
for epoch in range(num_epochs):
    loss_hist_train = 0
    for x_batch, y_batch in train_dl:
        pred = model(x_batch)[:, 0]
        loss = loss_fn(pred, y_batch)
        loss.backward()
        optimizer.step()

```

```

optimizer.zero_grad()
loss_hist_train += loss.item()
if epoch % log_epochs==0:
    print(f'Epoch {epoch} Loss : '
          f'{loss_hist_train/len(train_dl):.4f}')
# Epoch 0 Loss 536.1047
# ...
# Epoch 180 Loss 6.2156

```

Вычислим качество модели на тестовых данных:

```

with torch.no_grad():
    pred = model(x_test.float())[:, 0]
    loss = loss_fn(pred, y_test)
    print(f'Test MSE: {loss.item():.4f}')
    print(f'Test MAE: {nn.L1Loss()(pred, y_test).item():.4f}')
# Test MSE: 9.6133
# Test MAE: 2.1211

```

Мы видим, что mean absolute error (MAE) равно 2.1, что довольно хорошо. Сейчас мы работали над регрессией, а в следующей части поработаем над классификацией.

13.6 Project two – classifying MNIST handwritten digits

Загрузим датасет и сформируем data loader:

```

import torchvision
from torchvision import transforms
import torch
from torch.utils.data import DataLoader
image_path = './'
transform = transforms.Compose([
    transforms.ToTensor()
])
mnist_train_dataset = torchvision.datasets.MNIST(
    root=image_path, train=True,
    transform=transform, download=True
)
mnist_test_dataset = torchvision.datasets.MNIST(
    root=image_path, train=False,
    transform=transform, download=True
)
batch_size = 64
torch.manual_seed(1)
train_dl = DataLoader(mnist_train_dataset,
                      batch_size, shuffle=True)

```

Метод `ToTensor()` конвертирует фичи из пикселей в тензор, состоящий из чисел с плавающей точкой, также он нормализует фичи из диапазона [0, 255] в диапазон [0, 1]. Заметим, что можно получить доступ к сырым пикселям через атрибут `data`, правда тогда надо не забыть привести их в диапазон от 0 до 1. Пришло время написать NN модель:

```

import torch.nn as nn
hidden_units = [32, 16]
# первый полъ – номер примера
# второй полъ – наш тензор (на позиции 1 – метка)
image_size = mnist_train_dataset[0][0].shape
input_size = image_size[0] * image_size[1] * image_size[2]
all_layers = [nn.Flatten()]
for hidden_unit in hidden_units:
    layer = nn.Linear(input_size, hidden_unit)
    all_layers.append(layer)
    all_layers.append(nn.ReLU())
    input_size = hidden_unit
all_layers.append(nn.Linear(hidden_units[-1], 10))
all_layers.append(nn.Softmax(dim=1))
model = nn.Sequential(*all_layers)
print(model)

```

Пришло время обучить модель:

```

loss_fn = nn.CrossEntropyLoss()
# см. главу 14 – Adam
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
torch.manual_seed(1)
num_epochs = 20
for epoch in range(num_epochs):
    accuracy_hist_train = 0

```

```

for x_batch, y_batch in train_dl:
    pred = model(x_batch)
    loss = loss_fn(pred, y_batch)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
    accuracy_hist_train += is_correct.sum()
    accuracy_hist_train /= len(train_dl.dataset)
    print(f'Epoch {epoch} Accuracy {f'{accuracy_hist_train:.4f}}')
# Epoch 0 Accuracy 0.8277
# ...
# Epoch 19 Accuracy 0.9696

```

Посчитаем точность на test датасете:

```

pred = model(mnist_test_dataset.data / 255.)
is_correct = (torch.argmax(pred, dim=1) ==
              mnist_test_dataset.targets).float()
print(f'Test accuracy: {is_correct.mean():.4f}')
# Test accuracy: 0.9561

```

13.7 Higher-level PyTorch API – PyTorch-Lightning

В последние годы сообщество PyTorch разработало несколько библиотек и API поверх PyTorch. Самые яркие примеры: fastai, Catalyst, PyTorch Lightning, PyTorch-Ignite. Хотя Lightning фокусируется на простоте и гибкости, он также позволяет использовать фичи, такие как multi-GPU support и fast low-precision training, про которые можно почитать по [ссылке](#). В предыдущей части мы реализовали классификатор чисел, сейчас мы его напишем снова только уже с использованием Lightning. Установить можно одной строкой: `pip install pytorch-lightning`. Реализовать модель очень просто, код будет очень похожий, будем наследоваться от класса `LightningModule`. А вот и код:

```

import pytorch_lightning as pl
import torch
import torch.nn as nn

from torchmetrics import Accuracy

class MultiLayerPerceptron(pl.LightningModule):
    def __init__(self, image_shape=(1, 28, 28), hidden_units=(32, 16)):
        super().__init__()

        self.train_acc = Accuracy()
        self.valid_acc = Accuracy()
        self.test_acc = Accuracy()

        input_size = image_shape[0] * image_shape[1] * image_shape[2]
        all_layers = [nn.Flatten()]
        for hidden_unit in hidden_units:
            layer = nn.Linear(input_size, hidden_unit)
            all_layers.append(layer)
            all_layers.append(nn.ReLU())
            input_size = hidden_unit
        all_layers.append(nn.Linear(hidden_units[-1], 10))
        all_layers.append(nn.Softmax(dim=1))
        self.model = nn.Sequential(*all_layers)

    def forward(self, x):
        x = self.model(x)
        return x

    def training_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.train_acc.update(preds, y)
        self.log("train_loss", loss, prog_bar=True)
        return loss

    def training_epoch_end(self, outs):
        self.log("train_acc", self.train_acc.compute())

    def validation_step(self, batch, batch_idx):

```

```

x, y = batch
logits = self(x)
loss = nn.functional.cross_entropy(self(x), y)
preds = torch.argmax(logits, dim=1)
self.valid_acc.update(preds, y)
self.log("valid_loss", loss, prog_bar=True)
self.log("valid_acc", self.valid_acc.compute(), prog_bar=True)
return loss

def test_step(self, batch, batch_idx):
    x, y = batch
    logits = self(x)
    loss = nn.functional.cross_entropy(self(x), y)
    preds = torch.argmax(logits, dim=1)
    self.test_acc.update(preds, y)
    self.log("test_loss", loss, prog_bar=True)
    self.log("test_acc", self.test_acc.compute(), prog_bar=True)
    return loss

def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
    return optimizer

```

Модуль `torchmetrics` автоматически устанавливается вместе с Lightning, если возникает ошибка, то можно установить модуль отдельно: `pip install torchmetrics`. Метод `training_step` запускается на каждом отдельном batch во время обучения, а метод `training_epoch_end` вызывается в конце каждой эпохи для подсчёта точности во время тренировки. Теперь поговорим про то, как можно подготовить датасет для Lightning, есть три главных способа:

- Сделать датасет частью модели.
- Создать обычный data loader и передать его в `fit` метод для Lightning Trainer – Trainer мы обсудим в следующей части.
- Создать `LightningDataModule` класс.

Мы будем использовать третий подход как самый организованный, посмотрим на код:

```

from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torchvision.datasets import MNIST
from torchvision import transforms

class MnistDataModule(pl.LightningDataModule):
    def __init__(self, data_path='./'):
        super().__init__()
        self.data_path = data_path
        self.transform = transforms.Compose([transforms.ToTensor()])

    def prepare_data(self):
        MNIST(root=self.data_path, download=True)

    def setup(self, stage=None):
        # stage = 'fit' / 'validate' / 'test' / 'predict'
        mnist_all = MNIST(
            root=self.data_path,
            train=True,
            transform=self.transform,
            download=False
        )

        self.train, self.val = random_split(
            mnist_all, [55000, 5000], generator=torch.Generator() \
                .manual_seed(1)
        )

        self.test = MNIST(
            root=self.data_path,
            train=False,
            transform=self.transform,
            download=False
        )

    def train_dataloader(self):
        # num_workers - how many subprocesses to use for data loading
        # по умолчанию 0, т.е. главный процесс
        return DataLoader(self.train, batch_size=64, num_workers=4)

```

```

def val_dataloader(self):
    return DataLoader(self.val, batch_size=64, num_workers=4)

def test_dataloader(self):
    return DataLoader(self.test, batch_size=64, num_workers=4)

```

В конце концов создадим объект этого класса:

```

torch.manual_seed(1)
mnist_dm = MnistDataModule()

```

Сейчас обучим модель с помощью PyTorch Lightning Trainer класса. Lightning реализует класс **Trainer**, который заботится о всех промежуточных шагах (`zero_grad()`, `optimizer.step()`, ...), также очень просто указать один или больше GPU (если доступен). См. код:

```

mnist_classifier = MultiLayerPerceptron()
if torch.cuda.is_available():
    print('device=gpu')
    trainer = pl.Trainer(max_epochs=10, gpus=1)
else:
    print('device=cpu')
    trainer = pl.Trainer(max_epochs=10)
trainer.fit(model=mnist_classifier, datamodule=mnist_dm)

```

Во время обучения мы видим множество различной полезной информации, значения loss функции, оставшееся время, точность и многое другое. Оценим модель с помощью TensorBoard. Другая крутая фича Lightning – это возможности логирования (мы указывали `self.log` шаги), до и во время тренировки мы можем их визуализировать в TensorBoard, установим его: `pip install tensorboard`. По умолчанию Lightning записывает обучение в папке `lightning_logs`, чтобы визуализировать эти данные, нужно выполнить эту команду в терминале, которая откроет TensorBoard в браузере: `tensorboard --logdir lightning_logs/`. Если вы запускаете код в ноутбуке, то выполните этот код в ячейке, чтобы показать панель прямо в ноутбуке: `%load_ext tensorboard` и `%tensorboard --logdir lightning_logs/`. Если вы запустите код несколько раз, то будет несколько версий `version_i`. Если у вас всё же ничего не работает или возникает ошибка 403, то попробуйте отключить все cookie и всякие защиты, например, как в Firefox. Вот так выглядит эта панель:



Посмотрев на графики, можно предположить, что качество модели улучшится, если добавить ещё несколько эпох. Lightning позволяет загрузить checkpoint нашей модели и дотренировать её. А вот и код:

```

path = './lightning_logs/version_2/checkpoints/epoch=9-step=8600.ckpt'
if torch.cuda.is_available():
    trainer = pl.Trainer(max_epochs=15, resume_from_checkpoint=path, gpus=1)
else:
    trainer = pl.Trainer(max_epochs=15, resume_from_checkpoint=path)
trainer.fit(model=mnist_classifier, datamodule=mnist_dm)

```

Мы дотренировали ещё 5 эпох, это очень удобно, дальше можно опять вывести панель и посмотреть на графики. На графиках будет очень красиво достроены линии с 10 до 15 эпохи, мы видим, что тренировали не зря. Теперь оценим результат на тестовом датасете:

```

trainer.test(model=mnist_classifier, datamodule=mnist_dm)
# {'test_loss': 1.5140321254730225, 'test_acc': 0.9411739110946655}

```

Т.к. модель автоматически сохраняется, то мы можем её снова загрузить с помощью этого кода:

```

path = './lightning_logs/version_3/checkpoints/epoch=14-step=12900.ckpt'
model = MultiLayerPerceptron.load_from_checkpoint(path)

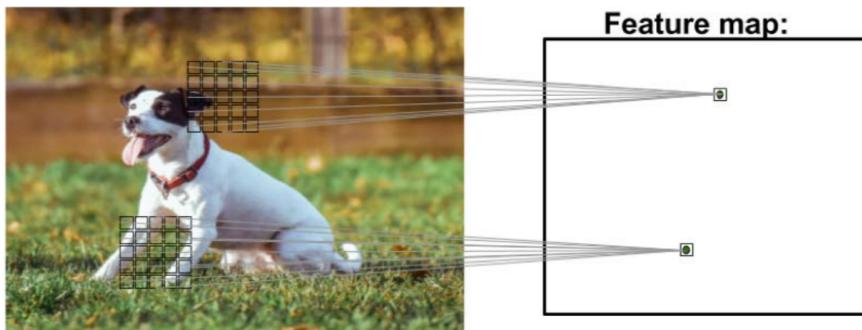
```

Следующая глава про convolutional neural network (CNN) архитектуру.

14 Classifying Images with Deep CNNs

14.1 The building blocks of CNNs

CNNs – это семейство алгоритмов, которые были придуманы на основе того, как работает зрительная кора нашего мозга. Учёные обнаружили, что разные нейроны по-разному реагируют при воздействии разных частей изображения. Было открыто несколько слоёв коры мозга, первый слой распознаёт линии и края, тогда как глубокие слои распознают сложные структуры. Мы поймём, почему CNNs часто описывают как “feature extraction layers”, также мы разберёмся в типах convolution операций и рассмотрим примеры. Успешное извлечение salient (relevant) features – это ключевая вещь для хорошего качества модели. Некоторые типы NNs, такие как CNNs, могут автоматически искать фичи из сырых данных, поэтому часто CNN слои рассматривают как feature extractors. Обычно первые слои выделяют low-level features из сырых данных, а уже глубокие слои (обычно fully connected layers) используют эти фичи, чтобы делать предсказания. Некоторые типы NNs, в частности CNNs, конструируют **feature hierarchy**, комбинируя low-level features послойно, чтобы сформировать high-level features. Например, в изображениях сначала выделяются линии и круги, а потом уже сложные формы. CNNs считают **feature maps**, посмотрим что это на картинке:



Этот локальный кусок пикселей называют **local receptive field**. CNNs очень хорошо показывают себя на изображениях, основная причина в двух идеях:

- **Sparse connectivity** – отдельный элемент в feature map связан с маленьким кол-вом других кусков пикселей.
- **Parameter sharing** – одни и те же веса используются для разных патчей изображения.

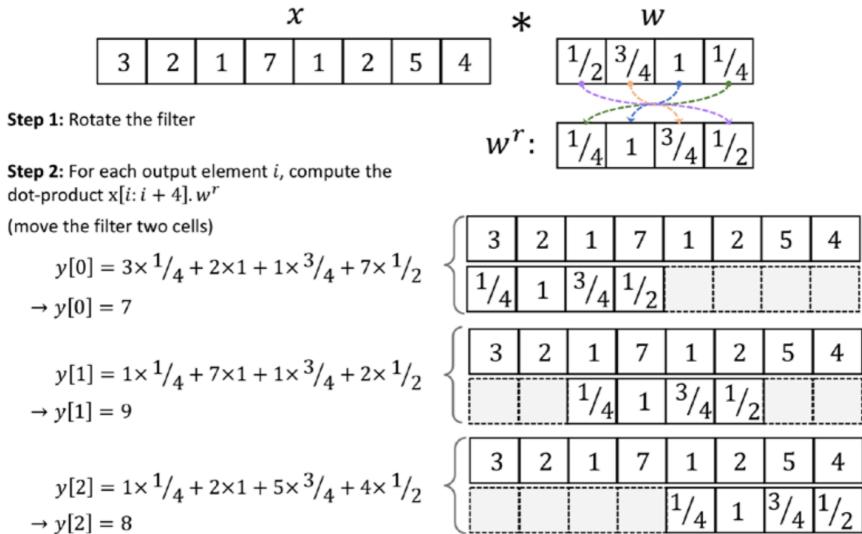
Как следствие из этих двух пунктов, convolution слой по сравнению с обычным слоем имеет намного меньше параметров, мы увидим улучшение в возможности улавливать значимые фичи. То есть соседние пиксели более релевантны, чем пиксели в разных углах. Обычно CNNs состоят из нескольких convolutional и subsampling слоёв, за которыми уже идут fully connected слои. Subsampling слои, обычно известные как **pooling layers**, не имеют никаких обучающих параметров. Однако два других типа слоёв в CNN имеют веса, которые мы как раз и учим. До конца этой секции мы будем разбираться с convolutional и pooling слоями. Чтобы понять как работают convolution операции, начнём с convolution в одной размерности, что иногда используют при работе, например, с текстом. Фундаментальная операция в CNNs – это **discrete convolution** (или просто **convolution**), есть несколько **naive** алгоритмов, чтобы посчитать эти свёртки. Мы будем использовать символ “*”, чтобы обозначить свёртку между двумя векторами или матрицами. Свёртка для двух векторов x и w записывается так: $y = x * w$, где x – это вход (или **signal**), а w – это **filter** или **kernel**. Свёртка записывается так:

$$y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]w[k]$$

В этой формуле две странные вещи – это $-\infty$ и $+\infty$, а также отрицательные индексы для x . Чтобы правильно посчитать значение формулы, предполагают, что x и w заполнены нулями до бесконечности. Т.к. на практике это бесполезно, вектор x окаймлён конечным числом нулей. Этот процесс называется **zero-padding** или просто **padding**. Количество нулей с каждой стороны обозначается как p . Пусть x и w имеют n и m элементов, $m \leq n$, так что вектор x^p имеет размер $n + 2p$. Практическая формула для свёртки такая:

$$y = x * w \rightarrow y[i] = \sum_{k=0}^{m-1} x^p[i+m-k]w[k]$$

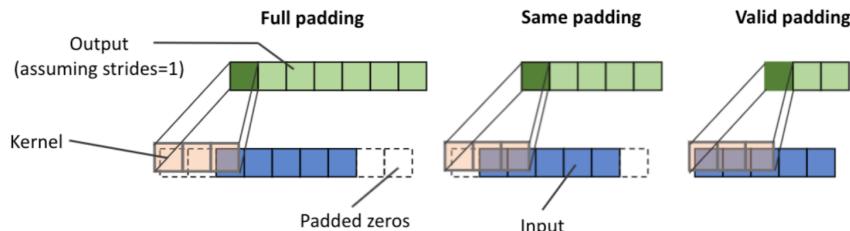
Индексы в сумме идут в разных направлениях, это эквивалентно случаю, где оба индекса идут в одном направлении, только один из векторов x или w перевёрнут. То есть $y[i] = x[i : i+m].w^r$, $x[i : i+m]$ – это как раз окно. Посмотрим на пример, см. картинку:



В этом примере $p = 0$. Заметим, что w^r двигается на две клетки на каждом шаге, это ещё один гиперпараметр свёртки – **stride**, s . Cross-correlation (или просто correlation) между входом и фильтром $y = x * w$ очень похожа на свёртку, которую мы прошли за тем отличием, что произведение идёт в одном направлении, а вот и формула:

$$y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i+k]w[k]$$

Большинство фреймворков реализует именно этот тип свёртки, но называют её так же, просто свёртка. Мы пока только использовали $p = 0$, технически можно взять любое $p \geq 0$. В зависимости от p граничные точки будут обрабатываться по-разному в отличие от центральных. Пусть $n = 5$ и $m = 3$, тогда, если $p = 0$, то крайние точки будут считаться ровно 1 раз, а если $p = 2$, то все точки будут считаться одинаково. Размер y также зависит от выбора p . Есть три режима padding: *full* ($p = m - 1$), *same* и *valid*. Full padding увеличивает размер выхода, поэтому он редко применяется в CNN. Same padding делает выход такого же размера как и вход, поэтому p считается в зависимости от размера фильтра и условия на одинаковый размер входа и выхода. В valid padding $p = 0$. См. картинку для лучшего понимания:



Самый часто используемый тип padding в CNN – это same padding, т.к. он сохраняет размер, что очень удобно при дизайне архитектуры сети. Главный минус valid padding по сравнению с остальными в том, что размер постоянно уменьшается, что приводит к плохому качеству сети. На практике нужно сохранять размер через same padding и уменьшать размер через pooling layers (или через convolutional layers с $stride = 2$). Full padding используется при обработке сигналов, т.к. очень важно минимизировать граничные эффекты. Поговорим про то, как определить размер выхода свёртки. Пусть размер входа n , размер фильтра m , тогда размер выхода свёртки $y = x * w$ с параметрами p и s равен:

$$o = \left\lceil \frac{n + 2p - m}{s} \right\rceil + 1$$

Теперь, чтобы лучше понять свёртку в одной размерности, мы напишем код и сравним результат с функцией `numpy.convolve`:

```
import numpy as np
def conv1d(x, w, p=0, s=1):
    w_rot = np.array(w[::-1])
    x_padded = np.array(x)
    if p > 0:
        zero_pad = np.zeros(shape=p)
        x_padded = np.concatenate([
```

```

        zero_pad, x_padded, zero_pad
    ])
res = []
for i in range(0, int(len(x_padded) - len(w_rot)) + 1, s):
    res.append(np.sum(x_padded[i:i+w_rot.shape[0]] * w_rot))
return np.array(res)

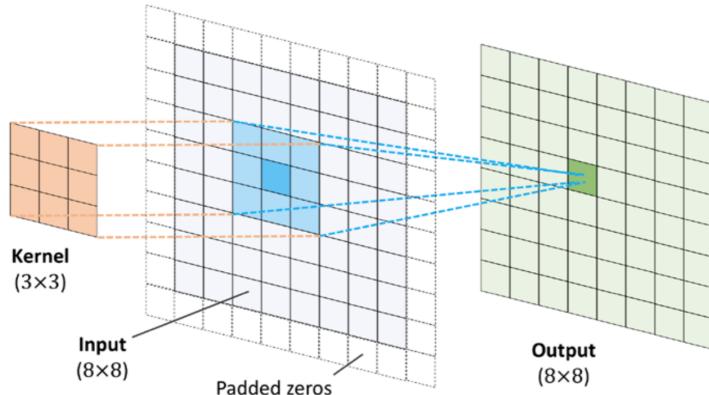
x = [1, 3, 2, 4, 5, 6, 1, 3]
w = [1, 0, 3, 1, 2]
print('Conv1d Implementation:', conv1d(x, w, p=2, s=1))
#[ 5. 14. 16. 26. 24. 34. 19. 22.]
print('NumPy Results:', np.convolve(x, w, mode='same'))
#[ 5 14 16 26 24 34 19 22]

```

Поговорим про свёртки в 2D. Пусть у нас есть матрица $X_{n_1 \times n_2}$ и фильтр $W_{m_1 \times m_2}$, где $m_1 \leq n_1$ и $m_2 \leq n_2$, тогда результат 2D свёртки выглядит так:

$$Y = X * W \rightarrow Y[i, j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i - k_1, j - k_2] W[k_1, k_2]$$

Всё, что мы обсудили (zero padding, разворот фильтра, strides), расширяется и на 2D случай по каждой размерности независимо. См. пример на картинке для лучшего понимания:



Пример разворота фильтра, мы просто отражаем матрицу относительно строк, потом относительно столбцов:

$$W = \begin{bmatrix} 0.5 & 0.7 & 0.4 \\ 0.3 & 0.4 & 0.1 \\ 0.5 & 1 & 0.5 \end{bmatrix} \Rightarrow W^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

Важно, что этот поворот – это не тоже самое, что и транспонирование. Чтобы получить развёрнутый фильтр в NumPy, можно написать так: `W_rot=W[::-1,::-1]`. Также напишем наивный алгоритм 2D свёртки и сравним результат с помощью функции `scipy.signal.convolve2d`:

```

import numpy as np
import scipy.signal
def conv2d(X, W, p=(0, 0), s=(1, 1)):
    W_rot = np.array(W)[::-1, ::-1]
    X_orig = np.array(X)
    n1 = X_orig.shape[0] + 2*p[0]
    n2 = X_orig.shape[1] + 2*p[1]
    X_padded = np.zeros(shape=(n1, n2))
    X_padded[p[0]:p[0]+X_orig.shape[0],
              p[1]:p[1]+X_orig.shape[1]] = X_orig

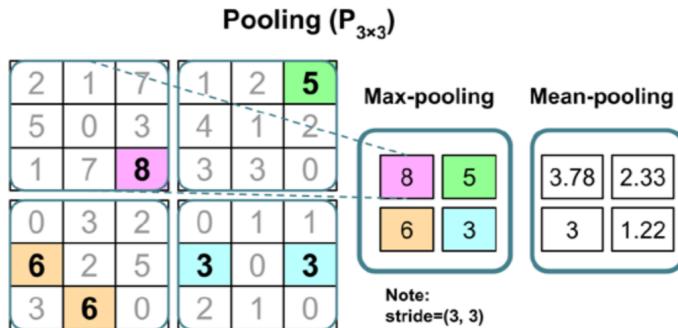
    res = []
    for i in range(0,
                   int((X_padded.shape[0] - W_rot.shape[0])/s[0])+1, s[0]):
        res.append([])
        for j in range(0, int((X_padded.shape[1] - \
                               W_rot.shape[1])/s[1])+1, s[1]):
            X_sub = X_padded[i:i+W_rot.shape[0],
                              j:j+W_rot.shape[1]]
            res[-1].append(np.sum(X_sub * W_rot))
    return np.array(res)

X = [[1, 3, 2, 4], [5, 6, 1, 3], [1, 2, 0, 2], [3, 4, 3, 2]]
W = [[1, 0, 3], [1, 2, 1], [0, 1, 1]]
print('Conv2d Implementation:\n', conv2d(X, W, p=(1, 1), s=(1, 1)))

```

```
# Conv2d Implementation:
# [[11. 25. 32. 13.]
# [19. 25. 24. 13.]
# [13. 28. 25. 17.]
# [11. 17. 14. 9.]]
print('SciPy Results:\n', scipy.signal.convolve2d(X, W, mode='same'))
```

Эти реализации свёртки очень неэффективны в смысле памяти и сложности вычислений. В большинстве случаев матрицу фильтра не поворачивают, в последний год были придуманы очень эффективные алгоритмы, которые используют Fourier transform. Важно заметить, что размер ядра обычно сильно меньше размера картинки, обычно размеры такие: 1×1 , 3×3 , 5×5 . Для этих размеров были отдельно придуманы быстрые алгоритмы, например, Winograd's minimal filtering алгоритм. Сейчас мы обсудим subsampling или pooling – другая важная операция. Subsampling обычно применяется в двух формах pooling операций в CNNs: **max-pooling** и **mean-pooling** (или **average-pooling**). Обозначается pooling layer так: $P_{n_1 \times n_2}$. Эти индексы обозначают размер области, на которой делается max или mean операция, этот размер называют **pooling size**. См. картинку для лучшего понимания:



Преимущества pooling двоякие:

- Pooling (max-pooling) вводит локальную инвариантность. Это значит, что маленькие изменения в окрестности не меняют результат max-pooling, поэтому генерируются фичи, которые больше всего устойчивы к шуму в данных.
- Pooling уменьшает размер фич, что повышает эффективность вычислений, а также это уменьшает степень переобучения.

Можно делать overlapping и non-overlapping pooling. Традиционно применяется второй способ, нужно просто поставить $s = (n_1, n_2)$ для $P_{n_1 \times n_2}$. Однако первый способ тоже иногда применяется, прочтайте про него самостоятельно. Хотя pooling layers часто применяются для CNNs, иногда для уменьшения размера фич берут convolutional layers с $s = 2$, про них можно думать как про pooling layers с весами для обучения.

14.2 Implementing a CNN

Рассмотрим работу с несколькими входами или цветовыми каналами. На вход свёрточному слою можно дать несколько 2D матриц, они называются каналами, то есть обычно на вход ожидается тензор ранга 3 – $X_{N_1 \times N_2 \times C_{in}}$, где C_{in} – это число входных каналов. Для цветных картинок $C_{in} = 3$, а для серых $C_{in} = 1$, т.к. есть ровно один канал с интенсивностью серого. Когда мы работаем с изображениями, то нужно использовать `uint8` тип для экономии памяти, который принимает значения от 0 до 255 (RGB изображения имеют тот же диапазон). Загрузим картинку через `torchvision` ([изображение](#)):

```
import torch
from torchvision.io import read_image
img = read_image('example-image.png')
print('Image shape:', img.shape) # [3, 252, 221]
print('Number of channels:', img.shape[0]) # 3
print('Image data type:', img.dtype) # torch.uint8
print(img[:, 100:102, 100:102])
```

Следующий вопрос в том, как можно объединить несколько каналов и операцию свёртки. Мы делаем свёртку для каждого канала отдельно, а потом просто складываем матрицы. Свёртка для каждого слоя (c) имеет свою матрицу ядра $W[:, :, c]$. Мы имеем итоговую feature map A . Обычно свёрточный слой в CNN имеет более одной feature map, тогда тензоры ядра становятся четырёхмерными $width \times height \times C_{in} \times C_{out}$, здесь $width \times height$ – это размер ядра. Вот основные шаги при значении C_{out} равном 1 и больше 1:

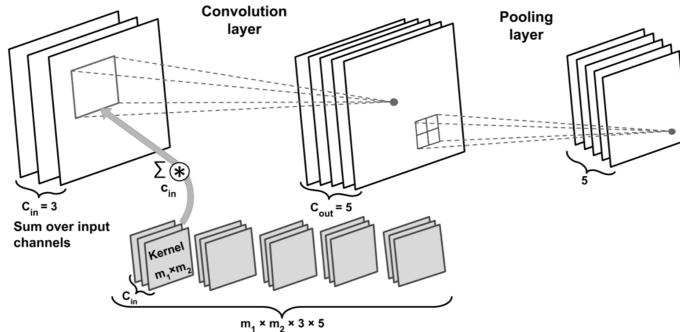
Given an example $X_{n_1 \times n_2 \times c_{in}}$,
a kernel matrix $W_{m_1 \times m_2 \times c_{in}}$,
and a bias value b

$$\Rightarrow \begin{cases} Z^{Conv} = \sum_{c=1}^{c_{in}} W[:, :, c] * X[:, :, c] \\ \text{Pre-activation: } Z = Z^{Conv} + b_c \\ \text{Feature map: } A = \sigma(Z) \end{cases}$$

Given an example $X_{n_1 \times n_2 \times c_{in}}$,
a kernel matrix $W_{m_1 \times m_2 \times c_{in} \times c_{out}}$,
and a bias vector $b_{c_{out}}$

$$\Rightarrow \begin{cases} Z^{Conv}[:, :, k] = \sum_{c=1}^{c_{in}} W[:, :, c, k] * X[:, :, c] \\ Z[:, :, k] = Z^{Conv}[:, :, k] + b[k] \\ A[:, :, k] = \sigma(Z[:, :, k]) \end{cases}$$

Посмотрим на пример для лучшего понимания, на нём 3 входных канала, тензор ядра четырёхмерный, каждая матрица ядра размером $m_1 \times m_2$, их 3 штуки для каждого канала, таких ядер всего пять для каждой выходной feature map. В конце есть pooling layer для subsampling feature maps. См. картинку:



Чтобы продемонстрировать плюсы свёртки, parameter sharing и sparse connectivity, посчитаем сколько параметров надо для обучения в нашей сети. У нас есть $m_1 \times m_2 \times 3 \times 5$ параметров, связанных с ядром, есть bias vector для каждой feature map, его размер 5. Pooling layers не имеют параметров для обучения, поэтому можно написать $m_1 \times m_2 \times 3 \times 5 + 5$. Если входной тензор имеет размер $n_1 \times n_2 \times 3$, то учитывая same padding, размер выходных feature maps равен $n_1 \times n_2 \times 5$. Если мы используем fully connected layer, то параметров больше, число параметров для матрицы весов должно быть $(n_1 \times n_2 \times 3) \times (n_1 \times n_2 \times 5) = (n_1 \times n_2)^2 \times 3 \times 5$, чтобы достичь того же кол-ва output units. В добавок, размер bias vector равен $n_1 \times n_2 \times 5$. Учитывая, что $m_1 < n_1$ и $m_2 < n_2$, мы видим, что разница в числе параметров значительная. Интересный факт, что свёртку можно расширить также на 3D для 3D датасетов. Теперь поговорим про регуляризацию с помощью L2 рег. и dropout. Огромные сети склонны к переобучению, сеть просто запомнит тренировочные данные и будет иметь супер хороший результат на нём, а вот на тестовых данных всё будет плохо, в реальных задачах размер сети заранее конечно неизвестен. Один из способов это решить в том, чтобы сделать модель чуть “сильнее”, чем нам надо, а потом применить одну или несколько регуляризаций, чтобы избежать переобучения. L2 рег. применяется чаще, чем L1. Чтобы реализовать L2 рег., нужно просто добавить штраф весов некоторого слоя к функции потерь. См. код:

```
import torch.nn as nn
loss_func = nn.BCELoss()
loss = loss_func(torch.tensor([0.9]), torch.tensor([1.0]))
l2_lambda = 0.001
conv_layer = nn.Conv2d(in_channels=3,
                      out_channels=5,
                      kernel_size=5)
l2_penalty = l2_lambda * sum([(p**2).sum() for p in conv_layer.parameters()])
loss_with_penalty = loss + l2_penalty
print(loss_with_penalty) # 0.1071
```

Альтернативный способ использовать L2 рег. – это поставить параметр `weight_decay` в PyTorch оптимизаторе, например:

```
optimizer = torch.optim.SGD(
    model.parameters(),
    weight_decay=l2_lambda,
    ...
)
```

Эти два способа разные, но для SGD они идентичны. В последний год **dropout** завоевал популярность для регуляризации моделей. Dropout чаще всего применяется для скрытых параметров в

последних уровнях. Идея такая: во время тренировки часть hidden units случайно отбрасывается на каждой итерации с вероятностью p_{drop} (или $p_{keep} = 1 - p_{drop}$), чаще всего берут $p = 0.5$. При отбрасывании доли входных нейронов, веса, связанные с оставшимися нейронами, масштабируются для учета отброшенных нейронов. Нейросеть не может полагаться на какое-то подмножество активаций скрытого слоя, так как они могут быть в любой момент выключены, из-за чего она начинает учить общие и надёжные шаблоны в данных. Эти случайные нейроны на каждой итерации выбираются случайно заново. Однако во время предсказания все нейроны будут участвовать в подсчёте pre-activations следующего слоя. Чтобы убедиться, что все активации одного масштаба и во время тренировки, и во время предсказания, активации активных нейронов нужно масштабировать соответственно (например, нужно разделить пополам, если $p = 0.5$). Однако это неудобно постоянно масштабировать активации во время предсказаний, поэтому PyTorch масштабирует активации во время тренировки, этот подход называется *inverse dropout*. Вообще dropout чем-то напоминает ensemble моделей. Тренировать несколько моделей в deep learning довольно дорого, а dropout даёт эффективный способ тренировать много моделей за раз и считать их среднее значение во время самого теста. Мы имеем новую модель для каждого mini-batch (так как каждый forward pass мы ставим случайные нули). То есть, проходя по этими batches, мы тренируем $M = 2^h$ моделей, где h – это число скрытых нейронов. Отличие от обычного ensembling в том, что мы делимся весами между всеми моделями. Подсчёт среднего значения по моделям, среднего значения по class-membership вероятностям выглядит так:

$$p_{\text{Ensemble}} = \left[\prod_{j=1}^M p^{\{i\}} \right]^{\frac{1}{M}}$$

Трюк dropout в том, что геометрическое среднее M моделей может быть приближено, масштабированием предсказаний финальной модели во время тренировки множителем $\frac{1}{1-p}$, что явно дешевле формулы выше (на самом деле, приближение точно равно данному среднему, если мы рассматриваем лин. модели). Поговорим про loss функции для классификации. Мы знаем разные функции активации, например, ReLU используется в скрытых слоях, sigmoid и softmax добавляются после последнего слоя для того, чтобы получить вероятности, иначе будут считаться просто logits. Мы рассматриваем классификацию, в зависимости от проблемы (бинарная или больше) и типа выхода (logits или вероятности) нужно выбирать разные loss функции. Binary cross-entropy используется, если выход один, а categorical cross-entropy – если выходов много. Для обоих случаев в `torch.nn` есть функции, см. на изображение, они могут принимать как logits, так и вероятности:

| Loss function | Usage | Example Using probabilities | Example Using logits |
|---|---------------------------|--|--|
| <code>BCELoss</code> or <code>BCEWithLogitsLoss</code> | Binary classification | <code>BCELoss</code> <code>y_true:</code> <code>y_pred:</code> | <code>BCEWithLogitsLoss</code> <code>y_true:</code> <code>y_pred:</code> |
| <code>NLLLoss</code> or <code>CrossEntropyLoss</code> | Multiclass classification | <code>NLLLoss</code> <code>y_true:</code> <code>y_pred:</code> | <code>CrossEntropyLoss</code> <code>y_true:</code> <code>y_pred:</code> |

Cross-entropy loss рекомендуется считать, передавая в параметры logits, т.к. это более стабильно с точки зрения численных вычислений. Для нескольких классов мы можем передать логарифмические вероятности в negative log-likelihood loss функцию `nn.NLLLoss()`. А вот и пример их использования:

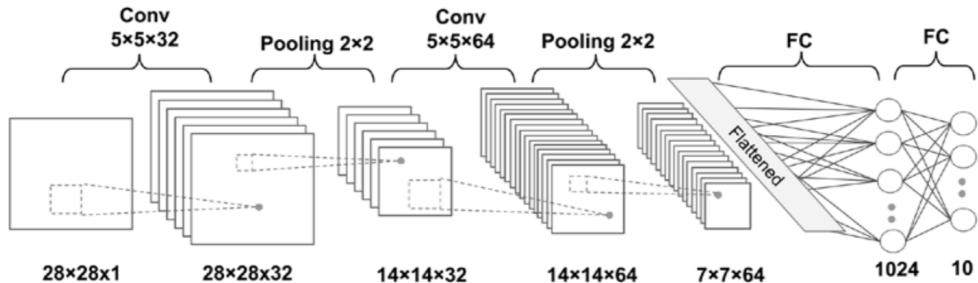
```
# Binary Cross-entropy
logits = torch.tensor([0.8])
probas = torch.sigmoid(logits)
target = torch.tensor([1.0])
bce_loss_fn = nn.BCELoss()
bce_logits_loss_fn = nn.BCEWithLogitsLoss()
print(f'BCE (w Probas): {bce_loss_fn(probas, target):.4f}') # 0.3711
print(f'BCE (w Logits): {bce_logits_loss_fn(logits, target):.4f}')


# Categorical Cross-entropy
logits = torch.tensor([[1.5, 0.8, 2.1]])
probas = torch.softmax(logits, dim=1)
target = torch.tensor([2])
cce_loss_fn = nn.NLLLoss()
cce_logits_loss_fn = nn.CrossEntropyLoss()
print(f'CCE (w Probas): {cce_loss_fn(probas, target):.4f}') # 0.5996
print(f'CCE (w Logits): {cce_logits_loss_fn(torch.log(probas), target):.4f}')
```

Мы привыкли, что, если у нас у сети один выход, то мы интерпретируем его как вероятность (второй выход не нужен, т.к. $p_{neg} = 1 - p_{pos}$), однако иногда бывает надо два выхода, тогда это считается уже не бинарной классификацией.

14.3 Implementing a deep CNN using PyTorch

В главе 13 мы распознавали рукописные цифры, мы достигли точности 95.6 процентов, в сети было 2 скрытых слоя. Сначала поговорим про архитектуру многослойной CNN сети. На входе у нас серые изображения размера 28×28 . Входные тензоры будут иметь размер $batchsize \times 28 \times 28 \times 1$. См. архитектуру сети на изображении:



Для свёрточных ядер мы используем `stride=1`, чтобы сохранять изначальные размеры изображений. Сейчас мы загрузим и предобработаем данные. А вот и код:

```
import torchvision
from torchvision import transforms
image_path = './'
transform = transforms.Compose([
    transforms.ToTensor()
])
mnist_dataset = torchvision.datasets.MNIST(
    root=image_path, train=True,
    transform=transform, download=True
)

from torch.utils.data import Subset
mnist_valid_dataset = Subset(mnist_dataset,
                             torch.arange(10000))
mnist_train_dataset = Subset(mnist_dataset,
                             torch.arange(10000,
                                         len(mnist_dataset)))

mnist_test_dataset = torchvision.datasets.MNIST(
    root=image_path, train=False,
    transform=transform, download=False
)
```

Теперь создадим data loaders:

```
from torch.utils.data import DataLoader
batch_size = 64
torch.manual_seed(1)
train_dl = DataLoader(mnist_train_dataset,
                      batch_size,
                      shuffle=True)
valid_dl = DataLoader(mnist_valid_dataset,
                      batch_size,
                      shuffle=False)
```

Чтобы соединять слои, мы будем использовать `torch.nn.Sequential` класс. А вот и классы для слоёв: `nn.Conv2d`, `nn.MaxPool2d`, `nn.AvgPool2d` и `nn.Dropout`. Обычно первая размерность тензора для изображения – это кол-во каналов, это так называемый NCHW формат (кол-во картинок в batch и каналов, высота и ширина). `Conv2D` слой требует именно такой формат, TensorFlow использует его же. Если всё такие каналы в последней размерности, то придётся сделать транспонирование. У `dropout` один параметр `p_drop`. Во время вызова этого слоя, его поведение можно контролировать через `model.train()` и `model.eval()` методы, важно правильно вызывать эти методы. Создадим модель, см. код:

```
model = nn.Sequential()
model.add_module(
    'conv1',
    nn.Conv2d(
        in_channels=1, out_channels=32,
```

```

        kernel_size=5, padding=2
    )
)
model.add_module('relu1', nn.ReLU())
model.add_module('pool1', nn.MaxPool2d(kernel_size=2))
model.add_module(
    'conv2',
    nn.Conv2d(
        in_channels=32, out_channels=64,
        kernel_size=5, padding=2
    )
)
model.add_module('relu2', nn.ReLU())
model.add_module('pool2', nn.MaxPool2d(kernel_size=2))

```

В pooling слое, если значение **stride** явно не указано, то оно равно размеру ядра. Проверим размер выхода:

```

x = torch.ones((4, 1, 28, 28))
model(x).shape
# torch.Size([4, 64, 7, 7]) - OK

```

Нужно выровнить слои в один слой:

```

model.add_module('flatten', nn.Flatten())
x = torch.ones((4, 1, 28, 28))
model(x).shape
# torch.Size([4, 3136]) - OK

```

Добавим полные слои и dropout между ними:

```

model.add_module('fc1', nn.Linear(3136, 1024))
model.add_module('relu3', nn.ReLU())
model.add_module('dropout', nn.Dropout(p=0.5))
model.add_module('fc2', nn.Linear(1024, 10))

```

Обычно мы ещё в конце добавляем softmax функцию, однако она уже находится внутри реализации метода **CrossEntropyLoss**, именно поэтому мы явно ничего в конце не добавляем. Ещё немного кода:

```

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

```

Adam оптимизатор – это надёжный gradient-based метод, который подходит для невыпуклых оптимизаций. Adam получился из двух других популярных методов оптимизации: **RMSProp** и **AdaGrad**. Ключевое преимущество Adam заключается в выборе размера шага обновления, полученного из текущего среднего значения моментов градиента. Напишем тренировку модели:

```

def train(model, num_epochs, train_dl, valid_dl):
    loss_hist_train = [0] * num_epochs
    accuracy_hist_train = [0] * num_epochs
    loss_hist_valid = [0] * num_epochs
    accuracy_hist_valid = [0] * num_epochs
    for epoch in range(num_epochs):
        model.train()
        for x_batch, y_batch in train_dl:
            pred = model(x_batch)
            loss = loss_fn(pred, y_batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            loss_hist_train[epoch] += loss.item() * y_batch.size(0)
            is_correct = (
                torch.argmax(pred, dim=1) == y_batch
            ).float()
            accuracy_hist_train[epoch] += is_correct.sum()
        loss_hist_train[epoch] /= len(train_dl.dataset)
        accuracy_hist_train[epoch] /= len(train_dl.dataset)

        model.eval()
        with torch.no_grad():
            for x_batch, y_batch in valid_dl:
                pred = model(x_batch)
                loss = loss_fn(pred, y_batch)
                loss_hist_valid[epoch] += loss.item() * y_batch.size(0)
                is_correct = (
                    torch.argmax(pred, dim=1) == y_batch
                ).float()
                accuracy_hist_valid[epoch] += is_correct.sum()
            loss_hist_valid[epoch] /= len(valid_dl.dataset)

```

```

accuracy_hist_valid[epoch] /= len(valid_dl.dataset)

print(f'Epoch {epoch+1} accuracy: ',
      f'{accuracy_hist_train[epoch]:.4f} val_accuracy: ',
      f'{accuracy_hist_valid[epoch]:.4f}')

return loss_hist_train, loss_hist_valid, \
       accuracy_hist_train, accuracy_hist_valid

```

Вызов нужного метода `train` или `eval` будет ставить нужный режим dropout слою, а также масштабировать нейроны так, как нам нужно. Пришло время тренировки:

```

torch.manual_seed(1)
num_epochs = 20
hist = train(model, num_epochs, train_dl, valid_dl)
# Epoch 1 accuracy: 0.9502 val_accuracy: 0.9798
# ...
# Epoch 20 accuracy: 0.9984 val_accuracy: 0.9890

```

Такая тренировка будет проходить довольно долго, поэтому, чтобы её ускорить, нужно переключиться на гпу. Для начала определим устройство:

```

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)

```

В коде функции для тренировки поменяем тип устройства у тензоров:

```
x_batch, y_batch = x_batch.to(device), y_batch.to(device)
```

Последним шагом поменяем устройство у модели:

```
model.to(device)
```

Тренировка должна занять несколько минут вместо часа с лишним, это очень круто. Теперь нарисуем learning curves:

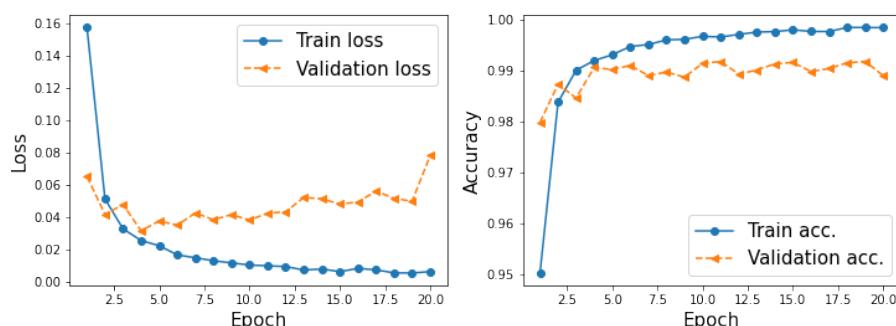
```

hist = [torch.tensor(h).to(torch.device('cpu')).numpy()
        for h in hist]

import matplotlib.pyplot as plt
x_arr = np.arange(len(hist[0])) + 1
fig = plt.figure(figsize=(12, 4))
ax = fig.add_subplot(1, 2, 1)
ax.plot(x_arr, hist[0], '-o', label='Train loss')
ax.plot(x_arr, hist[1], '--<', label='Validation loss')
ax.legend(fontsize=15)
ax.set_xlabel('Epoch', size=15)
ax.set_ylabel('Loss', size=15)

ax = fig.add_subplot(1, 2, 2)
ax.plot(x_arr, hist[2], '-o', label='Train acc.')
ax.plot(x_arr, hist[3], '--<', label='Validation acc.')
ax.legend(fontsize=15)
ax.set_xlabel('Epoch', size=15)
ax.set_ylabel('Accuracy', size=15)
plt.show()

```



Теперь вычислим точность на тестовых данных:

```

model.to(torch.device('cpu'))
pred = model(mnist_test_dataset.data.unsqueeze(1) / 255.)
is_correct = (
    torch.argmax(pred, dim=1) == mnist_test_dataset.targets
).float()
print(f'Test accuracy: {is_correct.mean():.4f}')
# Test accuracy: 0.9918

```

Возникает вопрос, зачем мы применили метод `unsqueeze`, он нужен, чтобы добавить 1 цветовой канал, см. код:

```
mnist_test_dataset.data.shape
# torch.Size([10000, 28, 28])
mnist_test_dataset.data.unsqueeze(1).shape
# torch.Size([10000, 1, 28, 28])
```

Выведем цифры и их метки:

```
fig = plt.figure(figsize=(12, 4))
for i in range(12):
    ax = fig.add_subplot(2, 6, i+1)
    ax.set_xticks([]); ax.set_yticks([])
    img = mnist_test_dataset[i][0][0, :, :]
    # NCHW:
    pred = model(img.unsqueeze(0).unsqueeze(1))
    y_pred = torch.argmax(pred)
    ax.imshow(img, cmap='gray_r')
    # transform=ax.transAxes - нормальные
    # математические оси
    ax.text(0.9, 0.1, y_pred.item(),
            size=15, color='blue',
            horizontalalignment='center',
            verticalalignment='center',
            transform=ax.transAxes)
```

Сейчас без картинки, и так понятно, что на ней будет.

14.4 Smile classification from face images using a CNN

В этой части мы напишем классификацию улыбок из CelebA датасета. Он содержит 202,599 изображений лиц, а также 40 атрибутов для каждого изображения, например, наличие улыбки и возраст (старый или молодой человек). Мы для простоты будем использовать только 16,000 примеров, чтобы ускорить вычисления. Однако, чтобы улучшить качество модели и уменьшить переобучение на таком маленьком датасете, мы будем использовать **data augmentation** технику. Сначала как всегда загрузим датасет:

```
! pip install wget
import wget
url = 'https://drive.google.com/u/0/uc?id=1m8-EBPgi5MRubrm6iQjafK2QMHDMSfJ&'
      '&export=download&confirm=t&uuid=c48ede51-1a7c-4a6c-8f91-043fd4171646'
wget.download(url)

import zipfile
with zipfile.ZipFile('./celeba.zip', 'r') as zip_ref:
    zip_ref.extractall('./')
    with zipfile.ZipFile('./celeba/img_align_celeba.zip', 'r') as zip_ref:
        zip_ref.extractall('./celeba')

import torchvision
image_path = './'
celeba_train_dataset = torchvision.datasets.CelebA(
    image_path, split='train',
    target_type='attr', download=False
)
celeba_valid_dataset = torchvision.datasets.CelebA(
    image_path, split='valid',
    target_type='attr', download=False
)
celeba_test_dataset = torchvision.datasets.CelebA(
    image_path, split='test',
    target_type='attr', download=False
)
print('Train set:', len(celeba_train_dataset)) # 162770
print('Validation set:', len(celeba_valid_dataset)) # 19867
print('Test set:', len(celeba_test_dataset)) # 19962
```

Безусловно можно поставить `download=True`, но тогда могут возникнуть ошибки разного рода, поэтому я скачивал датасет прямо с google диска. Можно просто скачать архив на компьютер явно и потом загрузить его в ноутбук, но это может быть довольно долго. Есть ещё вариант использовать Kaggle API, но для этого нужно будет регистрироваться и создавать файл с json вручную. Теперь обсудим **data augmentation**, техника для улучшения качества deep NNs. Data augmentation обобщает широкий набор техник для работы со случаями, где размер тренировочных данных ограничен. Некоторые техники позволяют изменять данные, а другие даже могут синтезировать новые данные. Эта техника не только для изображений, но есть некоторые, которые применяются только для

картинок, например, обрезка деталей изображения, изменение контрастности, яркости и насыщенности, переворачивание. Часть из этих трансформаций доступна в модуле `torchvision.transforms`. В следующем коде мы посмотрим на них в действии, размер изображений (218, 178). А вот и код:

```
import matplotlib.pyplot as plt
from torchvision import transforms
fig = plt.figure(figsize=(16, 8.5))

## Column 1: cropping to a bounding-box
ax = fig.add_subplot(2, 5, 1)
img, attr = celeba_train_dataset[0]
ax.set_title('Crop to a \nbounding-box', size=15)
ax.imshow(img)
ax = fig.add_subplot(2, 5, 6)
img_cropped = transforms.functional.crop(img, 50, 20, 128, 128)
ax.imshow(img_cropped)

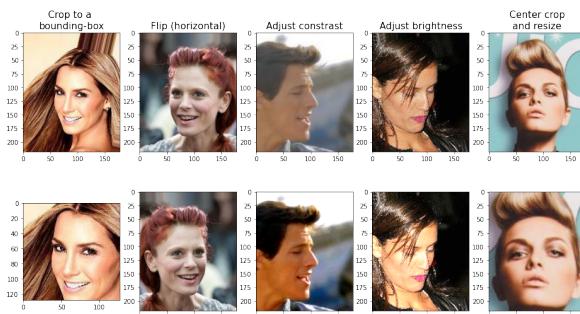
## Column 2: flipping (horizontally)
...
img_flipped = transforms.functional.hflip(img)
ax.imshow(img_flipped)

## Column 3: adjust contrast
...
img_adj_contrast = transforms.functional.adjust_contrast(
    img, contrast_factor=2)
ax.imshow(img_adj_contrast)

## Column 4: adjust brightness
...
img_adj_brightness = transforms.functional.adjust_brightness(
    img, brightness_factor=1.3)
ax.imshow(img_adj_brightness)

## Column 5: cropping from image center
...
img_center_crop = transforms.functional.center_crop(
    img, [0.7 * 218, 0.7 * 178])
img_resized = transforms.functional.resize(
    img_center_crop, size=(218, 178))
ax.imshow(img_resized)

plt.show()
```



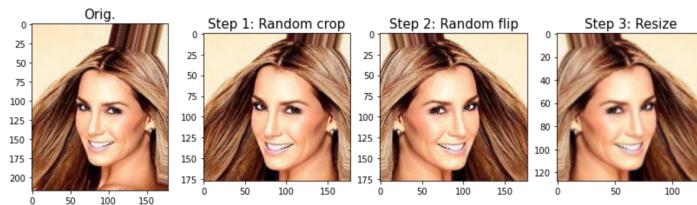
Напомню, что начало координат у картинки находится в левом верхнем углу. Мы сделали детерминированные трансформации, однако во время тренировки их лучше рандомизировать, например, случайные координаты левого верхнего угла, случайное переворачивание, `contrast_factor` можно выбирать из uniform distribution, также мы можем сделать pipeline из этих трансформаций. А вот и пример кода:

```
import torch
torch.manual_seed(1)
fig = plt.figure(figsize=(14, 12))
for i, (img, attr) in enumerate(celeba_train_dataset):
    ax = fig.add_subplot(3, 4, i*4+1)
    ax.imshow(img)
    if i == 0:
        ax.set_title('Orig.', size=15)
    ax = fig.add_subplot(3, 4, i*4+2)
    img_transform = transforms.Compose([
        transforms.RandomCrop([178, 178])
    ])
    img_cropped = img_transform(img)
```

```

ax.imshow(img_cropped)
if i == 0:
    ax.set_title('Step 1: Random crop', size=15)
ax = fig.add_subplot(3, 4, i*4+3)
img_transform = transforms.Compose([
    transforms.RandomHorizontalFlip()
])
img_flip = img_transform(img_cropped)
ax.imshow(img_flip)
if i == 0:
    ax.set_title('Step 2: Random flip', size=15)
ax = fig.add_subplot(3, 4, i*4+4)
img_resized = transforms.functional.resize(
    img_flip, size=(128, 128)
)
ax.imshow(img_resized)
if i == 0:
    ax.set_title('Step 3: Resize', size=15)
if i == 2:
    break
plt.show()

```



Я вставил только первый ряд изображений, остальные аналогичные. Давайте объявим лямбду, которая будет возвращать smile метку:

```
get_smile = lambda attr: attr[18]
```

Напишем pipeline для случайных изображений:

```

transform_train = transforms.Compose([
    transforms.RandomCrop([178, 178]),
    transforms.RandomHorizontalFlip(),
    transforms.Resize([64, 64]),
    transforms.ToTensor(),
])

```

Для тестовых и валидационных данных код другой:

```

transform = transforms.Compose([
    transforms.CenterCrop([178, 178]),
    transforms.Resize([64, 64]),
    transforms.ToTensor(),
])

```

Посмотрим на эти функции в действии:

```

from torch.utils.data import DataLoader
celeba_train_dataset = torchvision.datasets.CelebA(
    image_path, split='train',
    target_type='attr', download=False,
    transform=transform_train, target_transform=get_smile
)
torch.manual_seed(1)
data_loader = DataLoader(celeba_train_dataset, batch_size=2)
fig = plt.figure(figsize=(15, 6))
num_epochs = 5
for j in range(num_epochs):
    img_batch, label_batch = next(iter(data_loader))
    img = img_batch[0] # первый пример из двух в batch
    ax = fig.add_subplot(2, 5, j+1)
    ax.set_xticks([]); ax.set_yticks([])
    ax.set_title(f'Epoch {j}:', size=15)
    # здесь цветовой канал последний:
    ax.imshow(img.permute(1, 2, 0))

    img = img_batch[1]
    ax = fig.add_subplot(2, 5, j+6)
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(img.permute(1, 2, 0))
plt.show()

```



Плюсы от таких трансформаций очевидны. Теперь применим функцию `transform` к оставшимся двум частям датасета:

```
celeba_valid_dataset = torchvision.datasets.CelebA(
    image_path, split='valid',
    target_type='attr', download=False,
    transform=transform, target_transform=get_smile
)
celeba_test_dataset = torchvision.datasets.CelebA(
    image_path, split='test',
    target_type='attr', download=False,
    transform=transform, target_transform=get_smile
)
```

Теперь возьмём подмножество этих датасетов по причинам, которые мы описали в начале этой части:

```
from torch.utils.data import Subset
celeba_train_dataset = Subset(celeba_train_dataset,
                             torch.arange(16000))
celeba_valid_dataset = Subset(celeba_valid_dataset,
                             torch.arange(1000))
print('Train set:', len(celeba_train_dataset)) # 16000
print('Validation set:', len(celeba_valid_dataset)) # 1000
```

Последним шагом нужно создать data loaders:

```
batch_size = 32
torch.manual_seed(1)
train_dl = DataLoader(celeba_train_dataset,
                     batch_size, shuffle=True)
valid_dl = DataLoader(celeba_valid_dataset,
                     batch_size, shuffle=False)
test_dl = DataLoader(celeba_test_dataset,
                     batch_size, shuffle=False)
```

Пришло время тренировать CNN smile classifier. На вход мы подаём изображения размера $3 \times 64 \times 64$, они идут через свёрточные слои, чтобы сделать 32, 64, 128 и 256 feature maps, используя фильтры размера 3×3 и $p = 1$ (same padding), есть max-pooling слои $P_{2 \times 2}$ и dropout. См. код:

```
import torch.nn as nn
model = nn.Sequential()
model.add_module(
    'conv1',
    nn.Conv2d(
        in_channels=3, out_channels=32,
        kernel_size=3, padding=1
    )
)
model.add_module('relu1', nn.ReLU())
model.add_module('pool1', nn.MaxPool2d(kernel_size=2))
model.add_module('dropout1', nn.Dropout(p=0.5))

model.add_module(
    'conv2',
    nn.Conv2d(
        in_channels=32, out_channels=64,
        kernel_size=3, padding=1
    )
)
model.add_module('relu2', nn.ReLU())
model.add_module('pool2', nn.MaxPool2d(kernel_size=2))
model.add_module('dropout2', nn.Dropout(p=0.5))

model.add_module(
    'conv3',
    nn.Conv2d(
        in_channels=64, out_channels=128,
```

```

        kernel_size=3, padding=1
    )
)
model.add_module('relu3', nn.ReLU())
model.add_module('pool3', nn.MaxPool2d(kernel_size=2))

model.add_module(
    'conv4',
    nn.Conv2d(
        in_channels=128, out_channels=256,
        kernel_size=3, padding=1
    )
)
model.add_module('relu4', nn.ReLU())

x = torch.ones((4, 3, 64, 64))
model(x).shape
# torch.Size([4, 256, 8, 8]) - OK

```

Если мы добавим fc слой, то нам нужно 16,384 входов. Альтернативно, рассмотрим новый слой *global average-pooling*, который считает среднее значение каждой feature map отдельно, тем самым сокращая число нейронов до 256. Теперь уже можно добавить fc слой. Этот слой – это просто особый случай average-pooling, в котором pooling size равен размеру input feature maps. После этого слоя важно сжать (*squeeze*) размерности, нам нужно *batchsize* × 8, а иначе было бы *batchsize* × 8 × 1 × 1, т.к. размерность 64 × 64 стала бы просто 1 × 1. Напишем код для этого:

```

model.add_module('pool4', nn.AvgPool2d(kernel_size=8))
model.add_module('flatten', nn.Flatten())
x = torch.ones((4, 3, 64, 64))
model(x).shape
# torch.Size([4, 256]) - OK

model.add_module('fc', nn.Linear(256, 1))
model.add_module('sigmoid', nn.Sigmoid())
x = torch.ones((4, 3, 64, 64))
model(x).shape
# torch.Size([4, 1]) - OK

```

model

Функция потерь и оптимизатор:

```

loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

```

Переключим модель на gpu:

```

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
print(device)

```

Напишем функцию для тренировки, она почти такая же как и в предыдущей части, есть небольшие изменения:

```

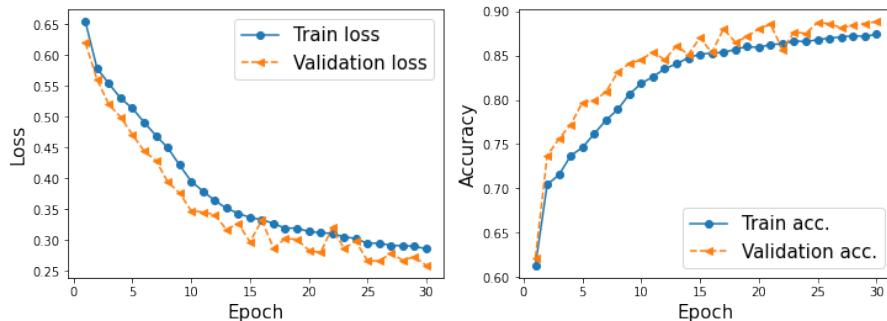
def train(model, num_epochs, train_dl, valid_dl):
    ...
    for epoch in range(num_epochs):
        model.train()
        for x_batch, y_batch in train_dl:
            x_batch, y_batch = x_batch.to(device), y_batch.to(device)
            pred = model(x_batch)[:, 0]
            loss = loss_fn(pred, y_batch.float())
            ...
            is_correct = ((pred >= 0.5).float() == y_batch).float()
            accuracy_hist_train[epoch] += is_correct.sum()
            ...
            with torch.no_grad():
                for x_batch, y_batch in valid_dl:
                    x_batch, y_batch = x_batch.to(device), y_batch.to(device)
                    pred = model(x_batch)[:, 0]
                    loss = loss_fn(pred, y_batch.float())
                    ...
                    is_correct = ((pred >= 0.5).float() == y_batch).float()
                    accuracy_hist_valid[epoch] += is_correct.sum()
                    loss_hist_valid[epoch] /= len(valid_dl.dataset)
                    ...

```

Пришло время тренировки:

```
torch.manual_seed(1)
num_epochs = 30
hist = train(model, num_epochs, train_dl, valid_dl)
# Epoch 1 accuracy: 0.6131 val_accuracy: 0.6220
# ...
# Epoch 30 accuracy: 0.8739 val_accuracy: 0.8880
```

Посмотрим теперь на learning curves, код ровно такой же как и в предыдущей части:

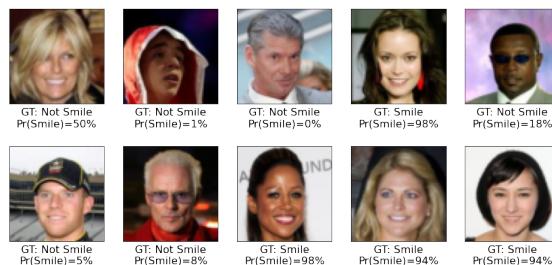


Теперь вычислим точность на тестовых данных:

```
accuracy_test = 0
model.eval()
model.to(device)
with torch.no_grad():
    for x_batch, y_batch in test_dl:
        x_batch, y_batch = x_batch.to(device), y_batch.to(device)
        pred = model(x_batch)[:, 0]
        is_correct = ((pred >= 0.5).float() == y_batch).float()
        accuracy_test += is_correct.sum()
accuracy_test /= len(test_dl.dataset)
print(f'Test accuracy: {accuracy_test:.4f}')
# Test accuracy: 0.8728
```

Вот примеры:

```
model.to(torch.device('cpu'))
pred = model(x_batch.to(torch.device('cpu'))[:, 0] * 100
fig = plt.figure(figsize=(15, 7))
for j in range(10, 20):
    ax = fig.add_subplot(2, 5, j-10+1)
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(x_batch[j].to(torch.device('cpu')).permute(1, 2, 0))
    label = 'Smile' if y_batch[j] == 1 else 'Not Smile'
    ax.text(
        0.5, -0.15,
        f'GT: {label}\nPr(Smile)={pred[j]:.0f}%', 
        size=16, horizontalalignment='center',
        verticalalignment='center',
        transform=ax.transAxes)
plt.show()
```

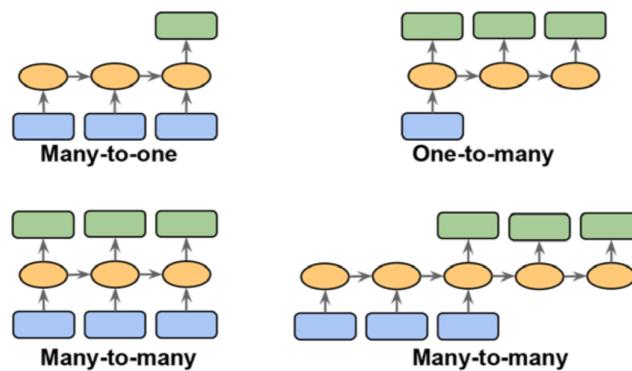


В следующей главе мы пройдём **recurrent neural networks (RNNs)**. RNNs используются для изучения структуры sequence data, они имеют великолепные применения, включая языковой перевод и заголовки к изображениям.

15 Modeling Sequential Data Using RNNs

15.1 Introducing sequential data

Мы начнём наше обсуждение RNNs с изучения природы sequential data, эти данные ещё называют sequence data или **sequences**. Мы посмотрим на уникальные свойства sequences, которые отличают эти данные от других. Уникальность sequences заключается в том, что элементы в sequence идут в определённом порядке и они не независимы друг от друга. Типичные алгоритмы машинного обучения для supervised learning предполагают, что вход – это **independent and identically distributed (IID)** data. Примером зависимых данных может служить прогнозирование рыночной стоимости конкретной акции. То есть, если у нас есть некоторая зависимость чего-то от времени, то важно давать данные в хронологическом, а не в случайном порядке. Time series data – это специальный тип sequential data, где каждый пример связан со временем. Примером этих данных может служить запись голоса человека. Однако далеко не все данные имеют зависимость от времени, например, текст или DNA sequences, мы будем применять RNNs для NLP. Последовательность обозначается так: $\langle x^{(1)}, x^{(2)}, \dots, x^{(T)} \rangle$. Говорят, что модель, которая не требует порядка данных, не имеет памяти. RNNs же запоминают информацию, которая уже прошла. Sequence modeling имеет много применений, например, перевод текста, заголовки изображений, генерация текста. Виды sequence modeling задач, которые можно выделить, зависят от взаимоотношений между входными и выходными данными. А вот и эти взаимоотношения:

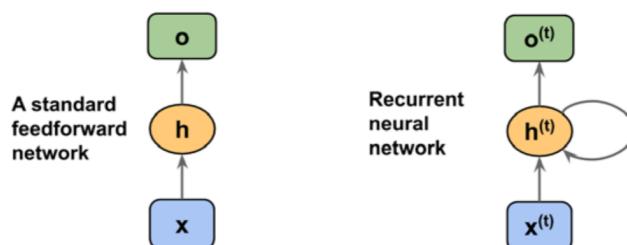


Обсудим их:

- **Many-to-one:** Входные данные – это последовательность, а выход – это число, или вектор фиксированного размера. Например, movie reviews, которые мы уже обсуждали.
- **One-to-many:** Входные данные – это обычные данные, а выход – это последовательность. Например, заголовки, на входе изображение, а на выходе какая-то фраза.
- **Many-to-many:** И вход, и выход – это последовательность. Этую группу можно дальше разделить на основе того, что вход и выход синхронизированы или нет. Пример синхронизированных данных – это классификация видео, где каждый кадр в видео помечен. Пример *delayed many-to-many* задачи моделирования – это перевод одного языка в другой. Всё предложение должно быть прочитано и обработано перед тем, как алгоритм выдаст переведённый текст.

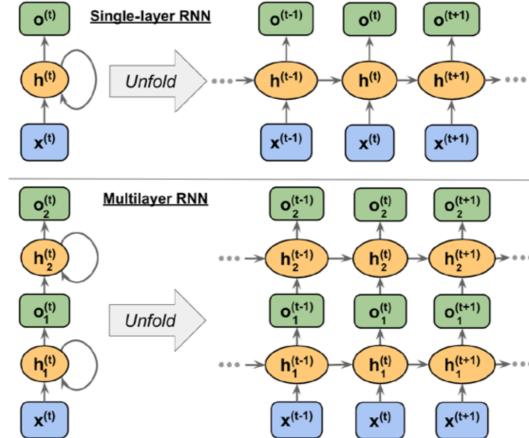
15.2 RNNs for modeling sequences

В этой части мы обсудим основные концепции RNNs, посмотрим, как считаются активации нейронов, также мы обсудим **LSTM** и **gated recurrent units (GRUs)**. Для начала поймём как идёт поток данных в RNNs. См. картинку, чтобы понять структуру:

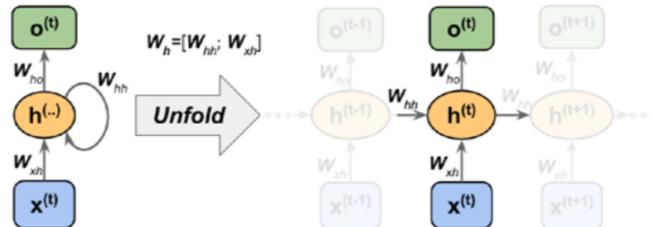


Обе сети имеют один скрытый слой, в этих обозначениях units не обозначаются, но мы предполагаем, что все 3 слоя имеют много units. Эта общая RNN архитектура может привести к двум типам

sequence modeling. Рекуррентный слой может вернуть $\langle o^{(0)}, o^{(1)}, \dots, o^{(T)} \rangle$ (many-to-many) или просто $o^{(T)}$ (many-to-one), если мы используем только последнее значение. В RNN скрытый слой получает информацию из входа на текущем шаге и из скрытого слоя с предыдущего шага. **Recurrent edge** позволяет RNN иметь память. Вот картинка с общей идеей:



Из входного слоя мы получаем preactivation, а с предыдущего шага – activation скрытого слоя $h^{(t-1)}$. Каждый рекуррентный слой должен получать последовательность на вход, поэтому все такие слои кроме последнего должны возвращать последовательность на выходе (`return_sequences=True`). Поведение последнего слоя зависит от проблемы. Поговорим про подсчёт активаций в RNNs. Каждое ориентированное ребро графа имеет матрицу весов, она не зависит от времени. W_{xh} – матрица между входом $x^{(t)}$ и скрытым слоем h , W_{hh} – матрица по рекуррентному ребру, W_{ho} – матрица между скрытым слоем и выходом. См. картинку для лучшего понимания:



Формула для подсчёта net input: $z_h^{(t)} = W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h$, активации скрытого слоя в момент времени t считаются так: $h^{(t)} = \sigma_h(z_h^{(t)})$. Если $W_h = [W_{xh}; W_{hh}]$, то

$$h^{(t)} = \sigma_h \left([W_{xh}; W_{hh}] \begin{bmatrix} x^{(t)} \\ h^{(t-1)} \end{bmatrix} + b_h \right)$$

Активации выходного слоя считаются так: $o^{(t)} = \sigma_o(W_{ho}h^{(t)} + b_o)$. RNNs тренируются с помощью алгоритма backpropagation through time (BPTT). Производные градиентов довольно сложные, но общая идея в том, что общие потери L – это сумма потерь всех функций потерь во времена от $t=1$ до $t=T$:

$$L = \sum_{t=1}^T L^{(t)}$$

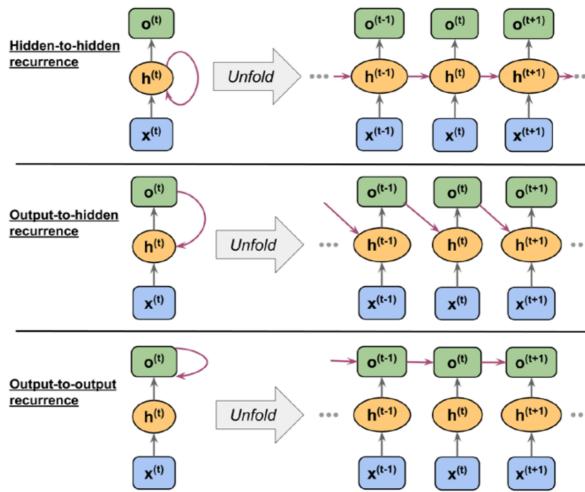
Так как потери во время t зависят от скрытых нейронов во все предыдущие времена $1:t$, градиент можно посчитать так:

$$\frac{\partial L^{(t)}}{\partial W_{hh}} = \frac{\partial L^{(t)}}{\partial o^{(t)}} \times \frac{\partial o^{(t)}}{\partial h^{(t)}} \times \left(\sum_{k=1}^t \frac{\partial h^{(t)}}{\partial h^{(k)}} \times \frac{\partial h^{(k)}}{\partial W_{hh}} \right)$$

Здесь $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ считается так:

$$\frac{\partial h^{(t)}}{\partial h^{(k)}} = \prod_{i=k+1}^t \frac{\partial h^{(i)}}{\partial h^{(i-1)}}$$

Сравним hidden recurrence и output recurrence. Мы рассматривали RNN сети, где скрытые слои имеют рекуррентное свойство. Есть другая модель, где рекуррентные рёбра идут из выходных слоёв, в этом случае o^{t-1} можно добавить в два места, к $h^{(t)}$ или к $o^{(t)}$. См. картинку для лучшего понимания:



Чтобы посмотреть, как это работает, давайте посчитаем forward pass для одного из типов RNN сетей. В `torch.nn` модуле есть слой RNN, это hidden-to-hidden рекурсия. Для начала создадим слой:

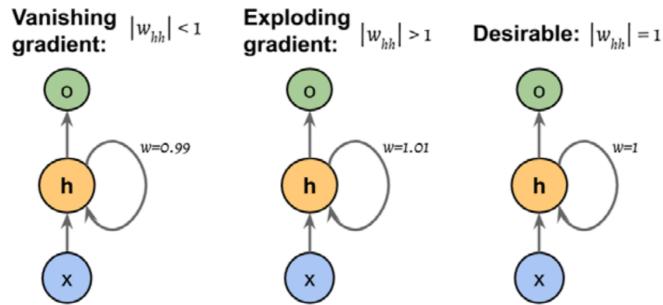
```
import torch
import torch.nn as nn
torch.manual_seed(1)
rnn_layer = nn.RNN(input_size=5, hidden_size=2,
                    num_layers=1, batch_first=True)
w_xh = rnn_layer.weight_ih_0
w_hh = rnn_layer.weight_hh_0
b_xh = rnn_layer.bias_ih_0
b_hh = rnn_layer.bias_hh_0
print('W_xh shape:', w_xh.shape) # [2, 5]
print('W_hh shape:', w_hh.shape) # [2, 2]
print('b_xh shape:', b_xh.shape) # [2]
print('b_hh shape:', b_hh.shape) # [2]
```

Входной размер для этого слоя – это (`batch_size`, `sequence_length`, 5), где первая размерность появилась из-за `batch_first=True`, вторая размерность для sequence, а последняя для фич. Если на вход подать sequence размера 3, то на выходе будет $\langle o^{(0)}, o^{(1)}, o^{(2)} \rangle$. По умолчанию RNN использует один слой, но можно поставить больше через `num_layers`, чтобы сформировать stacked RNN. Теперь уже посчитаем выход слоя:

```
x_seq = torch.tensor([[1.0]*5, [2.0]*5, [3.0]*5]).float()
# hn - это последний выход
output, hn = rnn_layer(torch.reshape(x_seq, (1, 3, 5)))
out_man = []
for t in range(3):
    xt = torch.reshape(x_seq[t], (1, 5))
    print(f'Time step {t} =>')
    print('Input : ', xt.numpy())
    ht = torch.matmul(xt, torch.transpose(w_xh, 0, 1)) + b_xh
    print('Hidden : ', ht.detach().numpy())
    if t > 0:
        prev_h = out_man[t-1]
    else:
        prev_h = torch.zeros((ht.shape))
    ot = ht + torch.matmul(prev_h, torch.transpose(w_hh, 0, 1)) + b_hh
    ot = torch.tanh(ot)
    out_man.append(ot)
    print('Output (manual) : ', ot.detach().numpy())
    print('RNN output : ', output[:, t].detach().numpy())
    print()
# Time step 0 =>
# Input : [[1. 1. 1. 1. 1.]]
# Hidden : [[-0.4701929  0.58639044]]
# Output (manual) : [[-0.3519801  0.52525216]]
# RNN output : [[-0.3519801  0.52525216]]
# ...
```

Поговорим про проблемы обучения параметров на больших расстояниях. ВРТТ приносит новые трудности из-за произведения множителей в частном $\frac{\partial h^{(t)}}{\partial h^{(k)}}$, появляются так называемые проблемы **vanishing** и **exploding** градиентов. Это частное содержит $t-k$ умножений, поэтому значение w^{t-k} становится либо очень маленьким, либо очень большим. Long-range dependencies

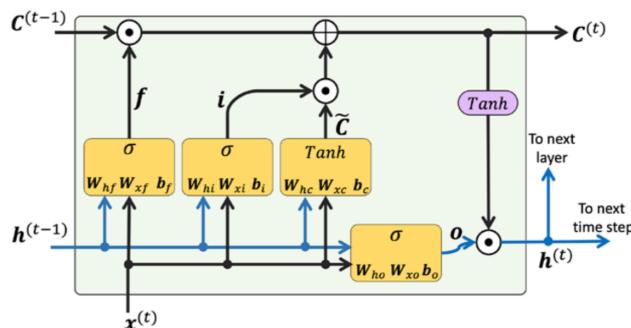
– это как раз разница $t - k$. Наивное решение можно сделать, если $|w| = 1$. См. картинку для лучшего понимания:



На практике есть как минимум три решения этой проблемы:

- Gradient clipping. Мы указываем cut-off или threshold значение, дальше мы присваиваем эти значения градиентам, если они вылезают за эти рамки.
- Truncated backpropagation through time (TBPTT). TBPTT ограничивает число временных шагов, которое сигнал может backpropagate после каждого forward pass. То есть, если последовательность имеет 100 шагов, то в вычислении могут участвовать только последние 20.
- LSTM.

Минус первых двух пунктов в том, что мы ограничиваем число эффективных шагов для обновления модели. LSTM – это более успешный алгоритм для этих проблем при моделировании дальнодействующих зависимостей за счет использования ячеек памяти. Давайте обсудим в деталях long short-term memory cells (длинные ячейки кратковременной памяти). Изначально LSTMs были придуманы для проблемы исчезания градиентов, строительный блок алгоритма – это **memory cell**, который по сути заменяет стандартный слой в RNNs. В каждой такой ячейке есть рекуррентное ребро с весом $w = 1$, значения, связанные с этим ребром называют **cell state**. Структура современной LSTM ячейки выглядит так:



Заметим, что состояние ячейки с прошлого шага $C^{(t-1)}$ меняется без прямого умножения на весовой множитель, чтобы получить состояние $C^{(t)}$. Поток информации в этой ячейке контролируется несколькими computation units (часто называют gates). На рисунке \odot обозначает поэлементное произведение, а \oplus обозначает поэлементное сложение. $x^{(t)}$ обозначает вход в момент времени t , а $h^{(t-1)}$ обозначает скрытые units в момент $t - 1$. Четыре ячейки показывают функцию активации, набор весов, это просто линейная операция. Блоки с функцией σ как раз и называются gates. В LSTM ячейке есть три типа gates: forget gate, input gate и output gate. **Forget gate** (f_t) позволяет сбросить состояние ячейки памяти без неограниченного роста. Эти ворота решают какой информации идти дальше, а какую информацию подавить:

$$f_t = \sigma(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f)$$

Изначально этого типа gates не было, его добавили через пару лет. **Input gate** (i_t) и **candidate value** (\tilde{C}_t) ответственны за обновление cell state. Считают их так:

$$i_t = \sigma(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i)$$

$$\tilde{C}_t = \text{Tanh}(W_{xc}x^{(t)} + W_{hc}h^{(t-1)} + b_c)$$

Cell state в момент t считают так:

$$C^{(t)} = (C^{(t-1)} \odot f_t) \oplus (i_t \odot \tilde{C}_t)$$

Output gate (o_t) решает как обновить значения скрытых units:

$$o_t = \sigma(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_o)$$

На основе этого, hidden units на текущем шаге считают так:

$$h^{(t)} = o_t \odot \tanh(C^{(t)})$$

PyTorch уже реализовал всё в оптимизированных функциях обёртки, что позволяет нам объявить нашу LSTM ячейку очень просто. Мы применим и RNNs, и LSTMs к реальному датасету позже в этой главе. Скажем пару слов про другие продвинутые RNN модели. LSTMs дают базовый подход для моделирования long-range dependencies в sequences. В литературе есть много вариаций LSTMs. Есть более новый подход, **gated recurrent unit (GRU)**. GRUs имеют более простую структуру, чем LSTMs, поэтому они вычислительно более эффективны в то время как их производительность в некоторых задачах, таких как моделирование полифонической музыки, сравнима с LSTM.

15.3 Implementing RNNs for sequence modeling in PyTorch

Мы применим RNNs к двум частым задачам:

1. Sentiment analysis.
2. Language modeling.

Первый проект – прогнозирование настроений в обзорах фильмов IMDb. Мы реализуем многослойную RNN для sentiment analysis, используя many-to-one архитектуру. Чуть позже мы реализуем many-to-many RNN для language modeling, это имеет много интересных применений, например, чатботы. Для начала приготовим movie review датасет. В главе 8 мы препроцессировали и очистили review датасет, сейчас мы сделаем тоже самое. Скачаем датасет:

```
! pip install torchtext
from torchtext.datasets import IMDB
train_dataset = IMDB(split='train')
test_dataset = IMDB(split='test')
```

Мы будем предсказывать либо **neg**, либо **pos** значение для каждого текста. Нам нужно разбить датасет на train и validation, найти уникальные слова и промаппить текст массиву индексов уникальных слов, разделить всё на mini-batches. Первый шаг:

```
from torch.utils.data import random_split
torch.manual_seed(1)
train_dataset, valid_dataset = random_split(
    list(train_dataset), [20000, 5000])
```

Сделаем шаги 2 и 3, для этого сначала нужно найти уникальные слова в train датасете. Для поиска уникальных токенов будет эффективнее использовать класс **Counter** из пакета **collections**. В этом применении нам нужен только набор уникальных слов, а не их кол-во, в отличие от bag-of-words модели. Мы будем переиспользовать **tokenizer** функцию из главы 8. См. код:

```
import re
from collections import Counter, OrderedDict

def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall(
        '(?:[:|]=)(?:-)?(?:\\)\\(|D|P)', text.lower())
    text = re.sub('[\\W]+', ' ', text.lower()) + \
        ''.join(emoticons).replace('-', ' ')
    tokenized = text.split()
    return tokenized

token_counts = Counter()
for label, line in train_dataset:
    tokens = tokenizer(line)
    token_counts.update(tokens)
print('Vocab-size:', len(token_counts))
# Vocab-size: 69023
```

Дальше надо сопоставить каждое уникальное слово уникальному числу. Это можно сделать просто обычным словарём, но в пакете **torchtext** уже есть класс **Vocab**, который мы можем использовать для такого маппинга и закодировать весь датасет. Мы создадим **vocab** объект, передавая ему отсортированный словарь. Также мы добавим два специальных токена – padding и unknown токены. См. код:

```

from torchtext.vocab import vocab
sorted_by_freq_tuples = sorted(
    token_counts.items(), key=lambda x: x[1], reverse=True
)
# Dictionary that remembers insertion order:
ordered_dict = OrderedDict(sorted_by_freq_tuples)
vocab = vocab(ordered_dict)
# don't tokens:
vocab.insert_token("<pad>", 0)
vocab.insert_token("<unk>", 1)
# если в vocab нет такого токена:
vocab.set_default_index(1)

```

Пример использования:

```

print([vocab[token] for token in \
      ['this', 'is', 'an', 'example']])
# [11, 7, 35, 457]

```

В valid и test датасетах есть токены, которых нет в train датасете, поэтому им будет присвоено число 1. Ещё есть *padding token* со значением 0, он нужен для настройки длины последовательности. Когда мы будем строить RNN модель, мы рассмотрим этот токен подробнее. Объявим функции для трансформации датасета:

```

text_pipeline = \
    lambda x: [vocab[token] for token in tokenizer(x)]
label_pipeline = lambda x: 1. if x == 'pos' else 0.

```

Напишем функцию для конвертации примеров в вектора чисел, которую передадим в `DataLoader`:

```

def collate_batch(batch):
    label_list, text_list, lengths = [], [], []
    for _label, _text in batch:
        label_list.append(label_pipeline(_label))
        processed_text = torch.tensor(text_pipeline(_text),
                                       dtype=torch.int64)
        text_list.append(processed_text)
        lengths.append(processed_text.size(0))
    label_list = torch.tensor(label_list)
    lengths = torch.tensor(lengths)
    # мы хотим, чтобы все тексты были одного размера,
    # поэтому маленькие массивы мы дополняем нулями
    padded_text_list = nn.utils.rnn.pad_sequence(
        text_list, batch_first=True
    )
    return padded_text_list, label_list, lengths

from torch.utils.data import DataLoader
dataloader = DataLoader(train_dataset, batch_size=4,
                       shuffle=False, collate_fn=collate_batch)

```

Хотя в общем случае RNNs могут поддерживать последовательности разных размеров, мы всё равно делаем так, чтобы все последовательности в mini-batch были одного размера, чтобы хранить его эффективно в тензоре. Посмотрим на то, как это работает:

```

text_batch, label_batch, length_batch = next(iter(dataloader))
print(text_batch)
print(label_batch)
print(length_batch) # tensor([165, 86, 218, 145])
print(text_batch.shape) # torch.Size([4, 218])

```

Создадим data loaders для уже реальных датасетов:

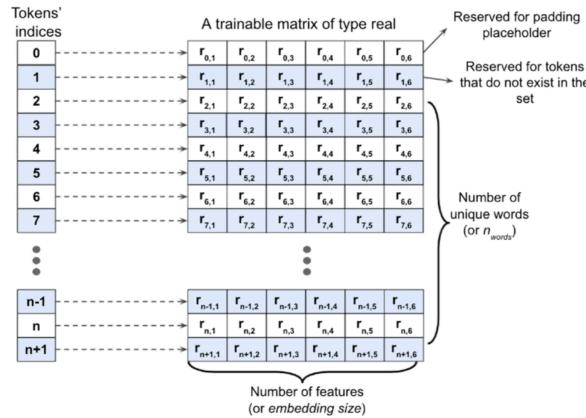
```

batch_size = 32
train_dl = DataLoader(train_dataset, batch_size=batch_size,
                      shuffle=True, collate_fn=collate_batch)
valid_dl = DataLoader(valid_dataset, batch_size=batch_size,
                      shuffle=False, collate_fn=collate_batch)
test_dl = DataLoader(test_dataset, batch_size=batch_size,
                     shuffle=False, collate_fn=collate_batch)

```

Обсудим feature **embedding** слои для кодирования текстов, это опциональный, но рекомендуемый шаг для уменьшения размерности вектора слов. Индексы уникальных слов в векторах могут быть конвертированы во входные фичи несколькими путями. Один наивный способ – это конвертировать индексы с помощью one-hot encoding. Дальше каждому слову сопоставим вектор того же размера, как и кол-во уникальных фич. Это кол-во может быть порядка $10^4\text{--}10^5$, из-за чего модель может страдать от curse of dimensionality. Также эти фичи очень разреженные. Более элегантный подход

в том, чтобы сопоставить каждому слову вектор фиксированного размера с вещественными значениями. Мы можем извлечь бесконечно много чисел из отрезка, например из $[-1, 1]$. Embedding – это feature-learning техника, которую можно здесь использовать, чтобы автоматически выбрать значимые фичи, чтобы представить слова. Нужно выбрать $embedding_dim \ll n_{unique\ words}$, чтобы представить весь словарь как входные фичи. Преимущества этого подхода по сравнению с one-hot encoding в том, что он сокращает размерность, также он выбирает значимые фичи, т.к. может быть обучен и оптимизирован. А вот и идея на картинке:



Если размер множества слов n , то кол-во токенов будет $n + 2$ (из-за того, что 0 – это padding, а 1 – для несуществующих), поэтому embedding matrix будет размера $(n + 2) \times embedding_dim$. Посмотрим на пример использования слова:

```
embedding = nn.Embedding(
    num_embeddings=10, # число уникальных токенов
    embedding_dim=3, # итоговая размерность
    padding_idx=0)
text_encoded_input = torch.LongTensor([[1, 2, 4, 5], [4, 3, 2, 0]])
print(embedding(text_encoded_input))
# tensor ([[ 0.8810,  0.4093,  2.3804],
#           [-0.0137,  1.0193,  0.8326],
#           [-1.1266,  0.8366,  0.1196],
#           [ 1.3220,  0.4102,  1.1531]],
#           ...], grad_fn=<EmbeddingBackward0>)
```

`padding_idx` равен индексу токена для padding (здесь 0), он не будет участвовать в обновлении градиентов во время обучения. После обучения слова, на значение 0, будет даваться вектор, состоящий полностью из нулей. Теперь наконец-то построим RNN модель. Для рекуррентных слоёв можно использовать любую из этих реализаций:

- RNN: обычный RNN слой, fc recurrent слой
- LSTM: long short-term memory RNN, полезен для выявления зависимостей на дальних расстояниях
- GRU: рекуррентный слой с GRU, альтернатива LSTM

Вот пример кода для многослойной RNN модели:

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.rnn = nn.RNN(input_size, hidden_size, num_layers=2,
                          batch_first=True)
        # можно попробовать nn.GRU или nn.LSTM вместо nn.RNN
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        # output – содержит выходные фичи  $h_t$  из
        # последнего слоя RNN для каждого момента  $t$ 
        # hidden – содержит финальное скрытое состояние
        # для каждого элемента в batch
        _, hidden = self.rnn(x)
        # мы используем финальное скрытое состояние
        # из последнего скрытого слоя как вход для fc layer
        out = hidden[-1, :, :]
        out = self.fc(out)
        return out
```

```

model = RNN(64, 32)
print(model)
print(model(torch.randn(5, 3, 64)))
# tensor([[-0.1540],
#         [ 0.0253],
#         [-0.1185],
#         [-0.3075],
#         [-0.1923]], grad_fn=<AddmmBackward0>)

Немного поясню этот код. На вход мы подаём batch из 5 примеров, каждый пример – это последовательность из 3х элементов, каждый элемент – это какие-то 64 значения. output содержит значения h_t для каждого момента времени в каждом примере, поэтому имеет размер [5, 3, 32]. hidden содержит выходы сети для каждого из 5 примеров, при этом выходы из обоих 2 слоёв, поэтому он имеет размер [2, 5, 32]. Теперь уже построим RNN модель для sentiment analysis задачи. Так как мы имеем очень длинные последовательности, мы будем использовать LSTM слой, начнём мы с embedding слоя с размерностью 20, дальше будут идти рекуррентный слой и два fc слоя. См. код:

class RNN(nn.Module):
    def __init__(self, vocab_size, embed_dim, rnn_hidden_size,
                 fc_hidden_size):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size,
                                      embed_dim,
                                      padding_idx=0)
        self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
                           batch_first=True)
        self.fc1 = nn.Linear(rnn_hidden_size, fc_hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(fc_hidden_size, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, text, lengths):
        out = self.embedding(text)
        # Упаковывает тензор, содержащий дополненные
        # последовательности переменной длины.
        # enforce_sorted=True – ожидается, что входные последовательности
        # отсортированы по длине в порядке убывания.
        out = nn.utils.rnn.pack_padded_sequence(
            out, lengths.cpu().numpy(), enforce_sorted=False, batch_first=True
        )
        # cell – содержит финальное состояние ячейки
        # для каждого элемента в последовательности
        out, (hidden, cell) = self.rnn(out)
        out = hidden[-1, :, :]
        out = self.fc1(out)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.sigmoid(out)
        return out

vocab_size = len(vocab)
embed_dim = 20
rnn_hidden_size = 64
fc_hidden_size = 64
torch.manual_seed(1)
model = RNN(vocab_size, embed_dim,
            rnn_hidden_size, fc_hidden_size)
print(model)

```

Теперь нужно написать функцию `train` для тренировки одной эпохи:

```

def train(dataloader):
    model.train()
    total_acc, total_loss = 0, 0
    for text_batch, label_batch, lengths in dataloader:
        optimizer.zero_grad()
        pred = model(text_batch, lengths)[:, 0]
        loss = loss_fn(pred, label_batch)
        loss.backward()
        optimizer.step()
        total_acc += (
            (pred >= 0.5).float() == label_batch
        ).float().sum().item()
        total_loss += loss.item() * label_batch.size(0)
    return total_acc/len(dataloader.dataset), \
           total_loss/len(dataloader.dataset)

```

Аналогично напишем `evaluate` функцию:

```

def evaluate(dataloader):
    model.eval()
    total_acc, total_loss = 0, 0
    with torch.no_grad():
        for text_batch, label_batch, lengths in dataloader:
            pred = model(text_batch, lengths)[:, 0]
            loss = loss_fn(pred, label_batch)
            total_acc += (
                (pred >= 0.5).float() == label_batch
            ).float().sum().item()
            total_loss += loss.item() * label_batch.size(0)
    return total_acc/len(dataloader.dataset), \
           total_loss/len(dataloader.dataset)

```

Создадим loss функцию и оптимизатор:

```

loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

```

Обучим модель:

```

num_epochs = 10
torch.manual_seed(1)
for epoch in range(num_epochs):
    acc_train, loss_train = train(train_dl)
    acc_valid, loss_valid = evaluate(valid_dl)
    print(f'Epoch {epoch} accuracy: {acc_train:.4f}', 
          f' val_accuracy: {acc_valid:.4f}')
# Epoch 0 accuracy: 0.5969 val_accuracy: 0.5946
# ...
# Epoch 9 accuracy: 0.9469 val_accuracy: 0.8378

```

Не забудем ускорить модель с помощью гри:

```

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
print(device)

text_batch, label_batch, lengths = text_batch.to(device), \
                                    label_batch.to(device), \
                                    lengths.to(device)

```

Вычислим точность на тестовых данных:

```

acc_test, _ = evaluate(test_dl)
print(f'test_accuracy: {acc_test:.4f}')
# test_accuracy: 0.8364

```

Чтобы это заработало, нужно внести небольшое изменение:

```

test_dl = DataLoader(list(test_dataset), batch_size=batch_size,
                     shuffle=False, collate_fn=collate_batch)

```

Мы просто показали, как работает RNN, результат далеко не самый лучший. Обсудим немного bidirectional RNN. Напоследок мы поставим параметр `bidirectional=True` у LSTM, что позволит рекуррентному слою проходить по входным последовательностям в обоих направлениях, с начала в конец и с конца в начало. А вот и код, здесь несколько маленьких изменений:

```

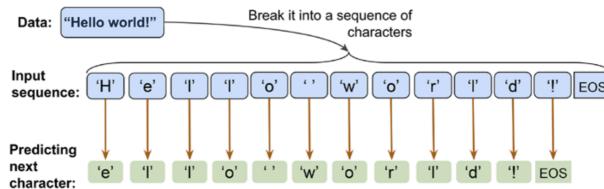
class RNN(nn.Module):
    def __init__(self, vocab_size, embed_dim, rnn_hidden_size,
                 fc_hidden_size):
        ...
        self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
                           batch_first=True, bidirectional=True)
        self.fc1 = nn.Linear(rnn_hidden_size * 2, fc_hidden_size)
        ...

    def forward(self, text, lengths):
        ...
        _, (hidden, cell) = self.rnn(out)
        # Если bidirectional=True, то h_n будет содержать конкатенацию
        # финальных forward и reverse скрытых состояний.
        out = torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1)
        ...

```

Bidirectional RNN слой делает два прохода по каждой входной последовательности: forward pass и reverse (или backward) pass. Итоговые скрытые состояния обычно конкatenируют в одно скрытое состояние, но можно сложить их, умножить или усреднить. Если обучить такую модель, то на тестовых данных можно получить примерно 85%. Теперь напишем вторую задачу – character-level

language modeling in PyTorch. Моделирование языка – это прекрасное применение, которое позволяет машинам выполнять задачи, связанные с нашим языком, например, генерация предложений на английском. В модели, которую мы сейчас сделаем, вход – это текстовый документ, наша задача – сделать модель для генерации текста, который по стилю похож на входной. Пример входа – это книга или код программы на каком-то языке. В character-level language моделировании вход разбит на последовательность символов, которые мы по одному отдаём нашей сети. Сеть будет обрабатывать каждый новый символ в соединении с памятью о предыдущих символах, чтобы предсказывать следующий. Идея такая:



Реализацию этой задачи можно разбить на 3 части: подготовка данных, построение RNN модели и предсказание следующих символов (+ генерация новых текстов). Начнём с первой части. Нам нужен какой-то большой текст для обучения, поэтому можно скачать книгу в простом текстовом формате с помощью команды `curl -O` – сохранять контент в файл с именем страницы или файла на сервере. Вот команда: `curl -O https://www.gutenberg.org/files/1268/1268-0.txt`. Загрузим книгу в сессию Python как обычный текст, мы уберём часть описательного лишнего текста до и после самого текста книги, дальше мы создадим переменную `char_set`, в ней будет сет уникальных символов в тексте. См. код:

```
import numpy as np

with open('1268-0.txt', 'r', encoding='utf8') as book:
    text = book.read()
start_indx = text.find('THE MYSTERIOUS ISLAND')
end_indx = text.find('End of the Project Gutenberg')
text = text[start_indx:end_indx]
char_set = set(text)

print('Total Length:', len(text)) # 1,130,711
print('Unique Characters:', len(char_set)) # 85
```

Большинство NN библиотек требуют вход в виде чисел, поэтому нужно конвертировать буквы в цифры. Заведём простой map `char2int`, который маппит буквы в числа, также нам нужен и обратный map, чтобы уметь перегнать результаты модели в текст. Конечно можно завести ещё один map, но проще сделать так: если букве 'a' соответствует число 2, то в массиве на позицию 2 нужно положить букву 'a'. А вот и код:

```
chars_sorted = sorted(char_set)
char2int = {ch: i for i, ch in enumerate(chars_sorted)}
char_array = np.array(chars_sorted)
text_encoded = np.array([
    char2int[ch] for ch in text],
    dtype=np.int32
)
print('Text encoded shape:', text_encoded.shape) # (1130711,)
print(text[:15], '== Encoding ==>', text_encoded[:15])
print(text_encoded[15:21], '== Reverse ==>',
      ''.join(char_array[text_encoded[15:21]]))
```

Ещё пару строк кода для примера:

```
for ex in text_encoded[:5]:
    print('{} -> {}'.format(ex, char_array[ex]))
```

Для генерации текста можно рассмотреть задачу как задачу классификации. У нас есть какой-то неполный текст или предложение, наша задача – это уметь предсказывать следующий символ. Пример: имеем `Hello`, `wor` ⇒ предсказываем `l`, имеем `ello`, `worl` ⇒ предсказываем `d`, и т.д. Мы ограничим длину последовательности 40 элементами. Это значит, что входной тензор состоит из 40 токенов. На практике, от длины последовательности зависит качество модели, при большей длине мы имеем предложения с более глубоким смыслом, при меньшей длине модель фокусируется больше на отдельных словах. Однако при большой длине нужно использовать LSTM по причине длинных зависимостей. Мы будем решать задачу multi классификации, например, если мы имеем на входе `nikit`, то выдадим мы `ikita`, только в нашем случае длина 40. Как мы видим x и y отстоят всего на один символ, поэтому сделаем chunks из 41 элемента, первые 40 – это x , последние 40 – это y . Создадим класс для датасета:

```

import torch
from torch.utils.data import Dataset
seq_length = 40
chunk_size = seq_length + 1
text_chunks = [text_encoded[i:i+chunk_size]
               for i in range(len(text_encoded)-chunk_size)]
class TextDataset(Dataset):
    def __init__(self, text_chunks):
        self.text_chunks = text_chunks

    def __len__(self):
        return len(self.text_chunks)

    def __getitem__(self, idx):
        text_chunk = self.text_chunks[idx]
        return text_chunk[:-1].long(), text_chunk[1:].long()

seq_dataset = TextDataset(torch.tensor(text_chunks))

for i, (seq, target) in enumerate(seq_dataset):
    print('Input (x): ', repr(''.join(char_array[seq])))
    print('Target (y): ', repr(''.join(char_array[target])))
    print()
    if i == 1:
        break
# Input (x): 'THE MYSTERIOUS ISLAND ***|n|n|n|n|nTHE MYSTER'
# Target (y): 'HE MYSTERIOUS ISLAND ***|n|n|n|n|nTHE MYSTERI'

# Input (x): 'HE MYSTERIOUS ISLAND ***|n|n|n|n|nTHE MYSTERI'
# Target (y): 'E MYSTERIOUS ISLAND ***|n|n|n|n|nTHE MYSTERO'

```

Последнее, что нужно сделать – это создать mini-batches:

```

from torch.utils.data import DataLoader
batch_size = 64
torch.manual_seed(1)
seq_dl = DataLoader(seq_dataset, batch_size=batch_size,
                    shuffle=True, drop_last=True)

```

Теперь создадим класс нашей character-level RNN модели:

```

import torch.nn as nn
class RNN(nn.Module):
    def __init__(self, vocab_size, embed_dim, rnn_hidden_size):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.rnn_hidden_size = rnn_hidden_size
        self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
                           batch_first=True)
        self.fc = nn.Linear(rnn_hidden_size, vocab_size)

    def forward(self, x, hidden, cell):
        # мы подаём в качестве x столбик из batch (размер - 64)
        # на вход нужно подать размер:
        # (длина послед., размер батча, кол-во фич) -> [64, 1, 256]
        out = self.embedding(x).unsqueeze(1)
        out, (hidden, cell) = self.rnn(out, (hidden, cell))
        out = self.fc(out).reshape(out.size(0), -1)
        return out, hidden, cell

    def init_hidden(self, batch_size):
        # размер: (кол-во слоёв, размер батча, размер скрытого слоя)
        hidden = torch.zeros(1, batch_size, self.rnn_hidden_size)
        cell = torch.zeros(1, batch_size, self.rnn_hidden_size)
        return hidden, cell

vocab_size = len(char_array)
embed_dim = 256
rnn_hidden_size = 512
torch.manual_seed(1)
model = RNN(vocab_size, embed_dim, rnn_hidden_size)
print(model)

```

Функция потерь и оптимизатор:

```

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

```

Теперь можно и тренировать:

```

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
model.to(device)
num_epochs = 10000
torch.manual_seed(1)
for epoch in range(num_epochs):
    hidden, cell = model.init_hidden(batch_size)
    seq_batch, target_batch = next(iter(seq_dl))
    optimizer.zero_grad()
    loss = 0
    for c in range(seq_length):
        # на выходе мы имеем logits
        seq_batch = seq_batch.to(device)
        hidden = hidden.to(device)
        cell = cell.to(device)
        target_batch = target_batch.to(device)
        pred, hidden, cell = model(seq_batch[:, c], hidden, cell)
        loss += loss_fn(pred, target_batch[:, c])
    loss.backward()
    optimizer.step()
    loss = loss.item()/seq_length
    if epoch % 500 == 0:
        print(f'Epoch {epoch} loss: {loss:.4f}')
# Epoch 0 loss: 4.4368
# Epoch 500 loss: 1.4892
# ...
# Epoch 9500 loss: 1.1037

```

Поговорим про этап оценки – создание новых отрывков текста. Logits, которые мы получили, можно превратить в вероятности с помощью softmax функции. Вместо того, чтобы выбирать каждый раз самый вероятный символ, мы хотим брать случайные буквы, иначе модель всё время будет давать один и тот же текст. Уже есть класс `torch.distributions.categorical.Categorical`, который берёт случайные примеры из категориального распределения. Вот пример кода:

```

from torch.distributions.categorical import Categorical
torch.manual_seed(1)
logits = torch.tensor([[1.0, 1.0, 1.0]])
print('Probabilities:', nn.functional.softmax(logits, dim=1).numpy()[0])
m = Categorical(logits=logits)
samples = m.sample((10,))
print(samples.numpy())
# 0,0,0,0,1,0,1,2,1,1

```

Эти цифры имеют равное распределение, однако, если, например, увеличить logits цифры 2 до 3, то она будет появляться намного чаще. Объявим функцию `sample()`, которая получает короткую строку и генерирует новую. Процесс получения сгенерированной последовательности на вход моделей для генерации новых элементов называется **autoregression**. А вот и код этой функции:

```

model.to(torch.device('cpu'))
def sample(model, starting_str,
          len_generated_text=500,
          scale_factor=1.0):
    encoded_input = torch.tensor(
        [char2int[s] for s in starting_str])
    # добавим размерность: [x] -> [[x]]
    encoded_input = torch.reshape(
        encoded_input, (1, -1))
    generated_str = starting_str

    model.eval()
    hidden, cell = model.init_hidden(1)
    for c in range(len(starting_str)-1):
        # view() меняет размер тензора без копирования памяти
        _, hidden, cell = model(
            last_char.view(1), hidden, cell)

    last_char = encoded_input[:, -1]
    for i in range(len_generated_text):
        logits, hidden, cell = model(
            last_char.view(1), hidden, cell)
        )
        # убираем первую размерность
        logits = torch.squeeze(logits, 0)
        scaled_logits = logits * scale_factor
        m = Categorical(logits=scaled_logits)
        last_char = m.sample()

```

```

generated_str += str(char_array[last_char])

return generated_str

torch.manual_seed(1)
print(sample(model, starting_str='It was a sunny day, I was sitting ,
'on a bench and talking with my friends .'))
```

Как мы видим, модель генерирует в основном правильные слова, а предложения иногда даже осмысленные. Конечно, можно дальше менять модель и её параметры для улучшения качества текста. При уменьшении множителя α растёт случайность и понижается качество, а при увеличении – наоборот. Посмотрим на пример:

```

logits = torch.tensor([[1.0, 1.0, 3.0]])
print('Probabilities before scaling:',
      nn.functional.softmax(logits, dim=1).numpy()[0])
# [0.10650698 0.10650698 0.78698605]
print('Probabilities after scaling with 0.5:',
      nn.functional.softmax(0.5*logits, dim=1).numpy()[0])
# [0.21194156 0.21194156 0.57611686]
print('Probabilities after scaling with 0.1:',
      nn.functional.softmax(0.1*logits, dim=1).numpy()[0])
# [0.3104238 0.3104238 0.37915248]
```

То есть, если мы поставим $\alpha = 2$, то получим довольно осмысленный текст, а если $\alpha = 0.5$, то более случайный. Мы поработали с генерацией текстов, это sequence-to-sequence (seq2seq) задача моделирования. Можно теперь, например, сделать чат бота для общения с людьми. В следующей главе мы посмотрим как можно усилить RNN с помощью attention механизма, что поможет моделировать long-range зависимости в задачах перевода. Дальше мы представим новую архитектуру глубокого обучения *transformer*, которая улучшила NLP область.

Продолжение будет в следующей части этого конспекта **ml-2**.

Краткое содержание.

1. Типы машинного обучения, терминология, общая идея, идея cross validation, generalized error
2. Perceptron, Adaline algorithms, градиентный спуск, feature scaling, SGD, OvA техника
3. Overfitting, scikit-learn, logistic regression, sigmoid function, lbfgs и другие, regularization, SVM, svm kernels (linear, rbf), decision tree, IG, random forests, KNN
4. Dealing with missing data, handling categorical data, one-hot encoding, feature scaling, selecting meaningful features (l_1 , l_2 reg.), sequential feature selection algorithms (SBS), оценка с помощью random forests
5. Principal component analysis (PCA), linear discriminant analysis (LDA), t-distributed stochastic neighbor embedding (t-SNE), uniform manifold approximation and projection (UMAP)
6. Pipelines, holdout и k-fold cross-validations (а также LOOCV, stratified cv), learning и validation curves, grid search, randomized search, hyperparameter search with successive halving (grid и randomized), TPE (Bayesian optimization), hyperopt-sklearn, nested cross-validation, precision, recall, F1 score, MCC, error, accuracy, TPR, FPR, receiver operating characteristic (ROC), ROC AUC, micro (и macro) averaging, class imbalance, SMOTE
7. Суть ensemble methods, majority voting, реализация majority vote classifier, GridSearchCV для ensemble (с помощью get_params), stacking, bagging, boosting и его реализация AdaBoost, другой вариант концепции boosting – gradient boosting и его реализация XGBoost, разные реализации gradient boosting: GradientBoostingClassifier, LightGBM, CatBoost и новая реализация в scikit-learn – HistGradientBoostingClassifier
8. Natural language processing (NLP), sentiment analysis (opinion mining), Python Progress Indicator (PyPrind), загрузка и распаковка архива, bag-of-words model, tf, tf-idf, idf, N-gram models, cleaning text data (regex), tokenizing, word stemming, NLTK, Snowball stemmer, lemmatization, Lancaster stemmer, stop word removal, итоговая реализация для IMDb, naïve Bayes classifier, online algorithms, out-of-core learning, пару слов о recurrent neural networks, HashingVectorizer, 32-bit MurmurHash3 функция, word2vec, topic modeling, expectation-maximization (EM) алгоритм, latent Dirichlet allocation (LDA)
9. Simple linear regression, multiple lin. reg., exploratory data analysis (EDA), heatmap, scatterplot и correlation матрицы, Pearson product-moment correlation coefficient (Pearson's r), ordinary least squares (OLS) модель, LAPACK, statsmodels библиотека, normal equation, RANdom SAmple Consensus (RANSAC), median absolute deviation (MAD), residual plots, MSE, mean absolute error (MAE), coefficient of determination (R^2), ridge regression, least absolute shrinkage and selection operator (LASSO), elastic net, polynomial regression, decision tree regression, random forest regression
10. Prototype-based clustering, k-means, k-means++, elbow и silhouette plots техники, hard и soft (или fuzzy) clustering, fuzzy C-means (FCM), hierarchical (agglomerative и divisive) и density-based clustering, dendograms, linkages (single, complete, average, Ward's), linkage matrix, heat map, DBSCAN, пару слов о graph-based clustering (самый популярный член – spectral clustering)
11. Neural networks (NNs), multilayer perceptron (MLP), forward/back propagation, MNIST датасет, полная реализация MLP, обсуждение существования разных техник улучшения NN, automatic differentiation
12. Пару слов про CPUs, GPUs и XLA устройства (TPUs), PyTorch и его установка, базовые операции с тензорами, Dataset и DataLoader, Image (из PIL) и transforms (из torchvision), датасеты из torchvision.datasets, torch.nn, torch.optim, nn.Module, написание NN с нуля и с помощью torch.nn и torch.optim, сохранение и загрузка модели, hyperbolic tangent, logistic, softmax, rectified linear unit (ReLU) функции активации
13. Computation graphs, requires_grad, torch.nn.init, Xavier (или Glorot) инициализация, automatic differentiation, понятие adversarial attacks, nn.Sequential, задача XOR классификации, библиотека mlxtend, написание кастомных слоёв, nn.Embedding, torch.bucketize, PyTorch-Lightning (и другие API), tensorboard
14. Convolution и pooling операции, четырёхмерное ядро, L2 регуляризация, dropout, loss функции, работа с gpu в PyTorch, data augmentation, CNN smile classifier

15. Sequences, IID data, типы sequence modeling, LSTM ячейки, идея RNNs и алгоритм BPTT, gradient clipping, TBPTT, GRUs, bidirectional RNNs, character-level language моделирование, два проекта: IMDb фильмы и генерация текстов