Линейный классификатор

Ограничения алгоритма k-nearest neighbors (kNN)

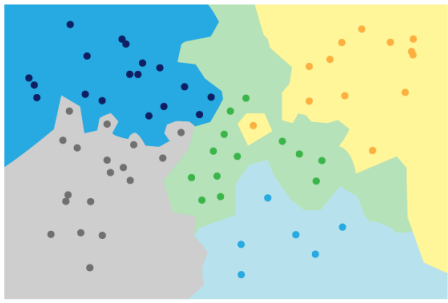
Продолжим с того места, где мы остановились в предыдущей лекции - классификация методом ближайших соседей (kNN).

Метрики расстояния

В практических заданиях EX01 мы использовали kNN для классификации изображений CIFAR10 (набор фотографий разделенных на 10 классов). Мы выяснили, что точность метода классификации с помощью kNN с использованем расстояний L1 (*Manhatten distance* - сумма абсолютных разностей между пикселями) оставляет желать лучшего, и попробовали применить L2 (*Euclidian distance*) в надежде эту точность повысить. С метриками L1 и L2 мы будем сталкиваться часто: и в качестве loss функции, и в качестве регуляризации, поэтому познакомиться с ними полезно.

L1 (Manhattan) distance

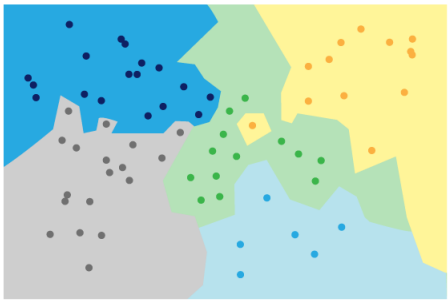
$$d_1(I_1, I_2) = \sum_p \left| I_1^p - I_2^p \right|$$



K = 1

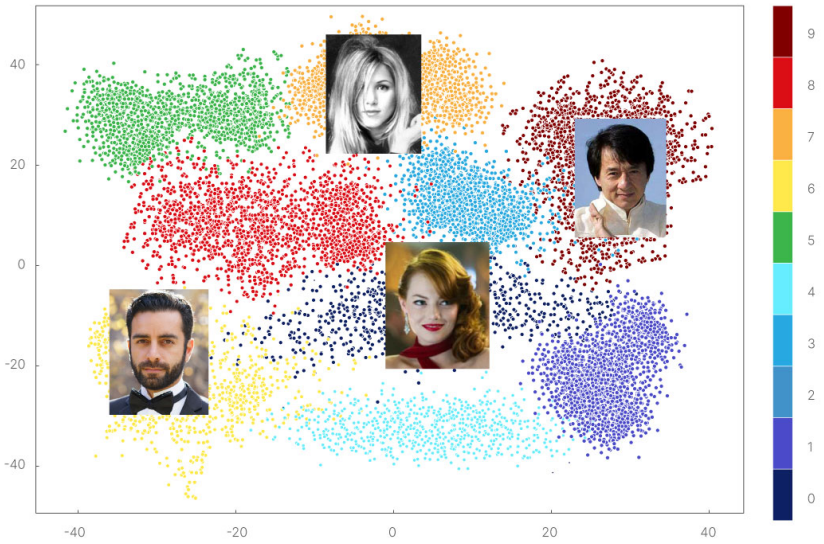
L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p \left(I_1^p - I_2^p \right)^2}$$



K = 1

kNN для классификации



На практике, метод ближайших соседей для классификации используется крайне редко. Давайте разберемся почему. Проблема заключается в следующем: предположим, что точность классификации нас устраивает. Теперь давайте применим kNN на больших данных (e.g. миллион картинок). Для определения класса каждой из картинок, нам нужно сравнить ее со всеми другими картинками в базе данных, такие расчеты, даже в супер оптимизированном виде, занимают много времени. Мы хотим, чтобы обученная модель работала быстро.

Тем не менее, метод ближайших соседей используется в других задачах, где без него обойтись сложно. Например, в задаче распознавания лиц. Представим, что у нас у нас есть большая база данных с фотографиями лиц (например, по 5 разных фотографий всех сотрудников, которые работают в офисном здании, как на примере выше) и есть камера, установленная на входе в это здание. Мы хотим узнать, кто и во сколько пришел на работу. Для того чтобы понять кто прошел перед камерой, нам нужно зафиксировать лицо этого человека и сравнить его со всеми фотографиями лиц в базе. В такой формулировке мы не пытаемся определить конкретный класс фотографии, а всего лишь определяем “похож-не похож”. Мы смотрим на k ближайших соседей и, например, если из k соседей, 5 - это фотографии Джеки Чана, то, скорее всего, под камерой прошел именно он. В таких случаях kNN метод вполне полезен. Похожим образом работает и поиск дубликатов в базах данных.

Примеры эффективной реализации метода на основе kNN:

- Facebook AI Research Similarity Search – разработка команды Facebook AI Research для быстрого поиска ближайших соседей и кластеризации в векторном пространстве. Высокая скорость поиска позволяет работать с очень большими данными – до нескольких миллиардов векторов.
- Алгоритм поиска ближайших соседей Hierarchical Navigable Small World.

Практические аспекты работы с классификаторами

Нормализация данных

Загрузим датасет с образцами здоровой и раковой ткани. Датасет состоит из 569 примеров, где каждой строчке из 30 признаков, соответствует класс `1` злокачественной (*malignant*) или `0` доброкачественной (*benign*) ткани. Задача состоит в том, чтобы по 30 признакам обучить модель определять тип ткани (злокачественная или доброкачественная).

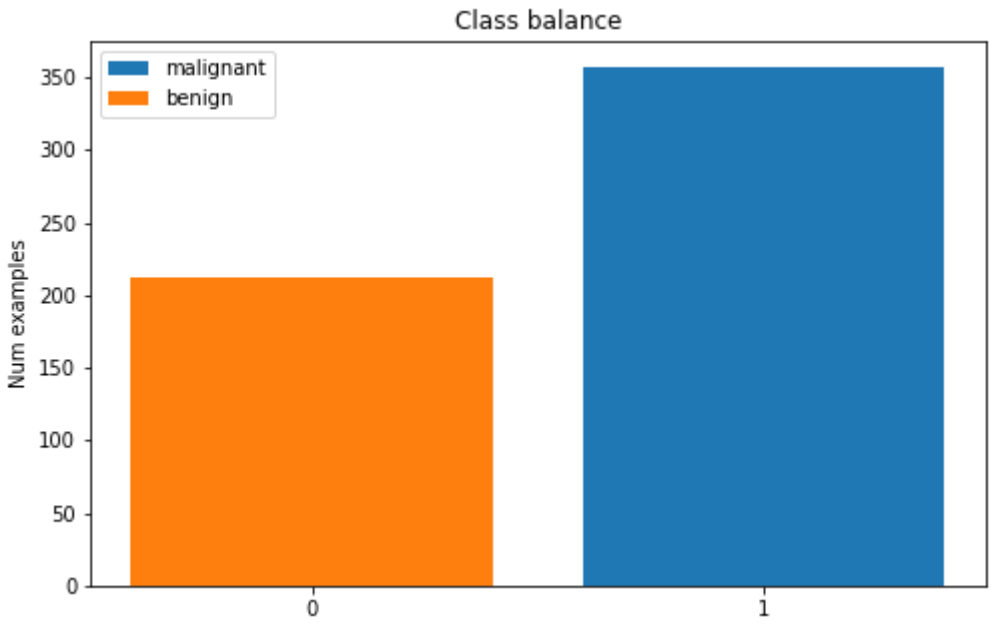
Можно иметь сколь угодно хороший алгоритм для классификации - но до тех пор, пока данные на входе - мусор, на выходе из нашего чудесного классификатора мы тоже будем получать мусор (*garbage in - garbage out*). Давайте разберемся, что конкретно надо сделать, чтобы kNN реально заработал.

```
import sklearn.datasets

cancer = sklearn.datasets.load_breast_cancer() # load data
X = cancer.data # features
Y = cancer.target # labels(classes)
print(f'X shape: {X.shape}, Y shape: {Y.shape}')
print(f'X[0]: \n {X[0]}')
print(f'Y[0]: \n {Y[0]}')
```

```
X shape: (569, 30), Y shape: (569,)
X[0]:
 [1.799e+01 1.038e+01 1.228e+02 1.001e+03 1.184e-01 2.776e-01 3.001e-01
 1.471e-01 2.419e-01 7.871e-02 1.095e+00 9.053e-01 8.589e+00 1.534e+02
 6.399e-03 4.904e-02 5.373e-02 1.587e-02 3.003e-02 6.193e-03 2.538e+01
 1.733e+01 1.846e+02 2.019e+03 1.622e-01 6.656e-01 7.119e-01 2.654e-01
 4.601e-01 1.189e-01]
Y[0]:
 0
```

Посмотрим сколько данных в классе `0` и сколько данных в классе `1`

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8,5)) # set fig size
plt.bar(1,Y[Y==1].shape, label=cancer.target_names[0]) # 1 label
plt.bar(0,Y[Y==0].shape, label=cancer.target_names[1]) # 0 label
plt.title('Class balance')
plt.ylabel('Num examples')
plt.xticks(ticks=[1,0], labels=['1','0'])
plt.legend(loc='upper left')
plt.show()
```



Теперь давайте посмотрим на сами данные. У нас есть 569 строк в каждой из которой по 30 колонок. Такие колонки называют признаками или *features*. Попробуем математически описать все эти признаки (mean, std, min и тд)

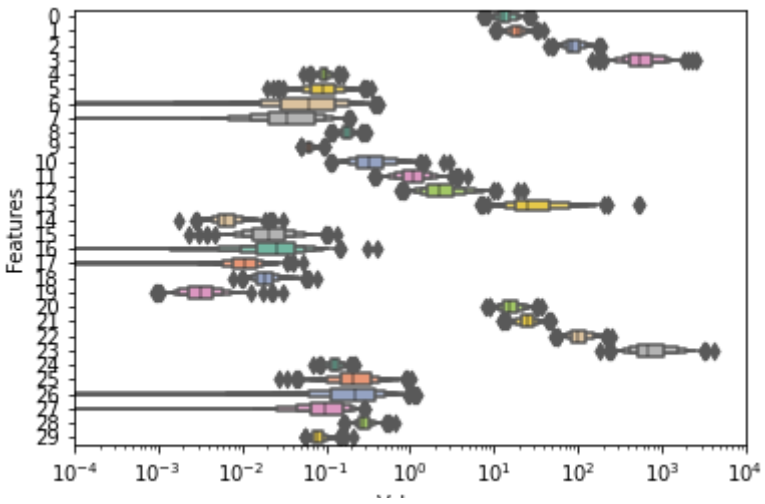
```
import pandas as pd
pd.DataFrame(X).describe()
```

	0	1	2	3	4	5	
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000
mean	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079
min	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000
25%	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029
50%	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061
75%	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426

То же самое, но в виде графика. Видно, что у фич совершенно разные значения.

```
import seaborn as sns

ax = sns.boxenplot(data=pd.DataFrame(X), orient="h", palette="Set2")
```

```
ax.set(xscale='log', xlim=(1e-4, 1e4), xlabel='Values', ylabel='Features')
plt.show()
```



Чтобы адекватно сравнить данные между собой нам следует использовать нормализацию.

Нормализация, выбор Scaler

Нормализация — это преобразование данных к неким безразмерным единицам. Ключевая цель нормализации — приведение различных данных в самых разных единицах измерения и диапазонах значений к единому виду, который позволит сравнивать их между собой.

Главное условие правильной нормализации — все признаки должны быть равны в возможностях своего влияния.

Например, у нас есть данные по группе людей: *возраст* (в годах) и *размер дохода* (в рублях). Возраст может измениться в диапазоне от 18 до 70 (интервал 70-18 = 52). А доход от 30 000 р до 500 000 р (интервал 500 000 - 30 000 = 470 000). В таком варианте разница в возрасте имеет меньшее влияние, чем разница в доходе. Получается, что доход становится более важным признаком, изменения в котором влияют больше при сравнении схожести двух людей.

Должно быть так, чтобы максимальные изменения любого признака в «основной массе объектов» были одинаковы. Тогда потенциально все признаки будут равноценны.

Осталось определиться с выбором инструмента, часто используют следующие варианты: `MinMaxScaler`, `StandardScaler`, `RobustScaler`

Сравним `MinMaxScaler`, `StandardScaler`, `RobustScaler` для признака `data[:,0]`. **Обратите внимание на ось X**

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
import numpy as np

np.random.seed(42)  # setting the initialization parameter for random values

# generate random values from 1 to 255, shape (30,1)
test = X[:,0].reshape(-1,1)

plt.figure(1, figsize=(30, 5))
plt.subplot(141)  # set location
plt.scatter(test, range(len(test)), c=Y)
plt.ylabel("Num examples", fontsize=15)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.title("Non scaled data", fontsize=18)

# scale data with MinMaxScaler
test_scaled = MinMaxScaler().fit_transform(test)
plt.subplot(142)
plt.scatter(test_scaled, range(len(test)), c=Y)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.title("MinMaxScaler", fontsize=18)

# scale data  with StandardScaler
test_scaled = StandardScaler().fit_transform(test)
plt.subplot(143)
plt.scatter(test_scaled, range(len(test)), c=Y)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.title("StandardScaler", fontsize=18)

# scale data  with RobustScaler
test_scaled = RobustScaler().fit_transform(test)
plt.subplot(144)
plt.scatter(test_scaled, range(len(test)), c=Y)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.title("RobustScaler", fontsize=18)
plt.show()
```

Идея `MinMaxScaler` заключается в том, что он преобразует данные в диапазоне от 0 до 1. Может быть полезно, если нужно выполнить преобразование, в котором отрицательные значения не допускаются (e.g., масштабирование RGB пикселей)

$$z = \frac{X_i - X_{min}}{X_{max} - X_{min}}$$

X_{min} и X_{max} задаются как минимальное и максимальное допустимое значение, по умолчанию: $X_{min} = 0$ и $X_{max} = 1$

Идея `StandardScaler` заключается в том, что он преобразует данные таким образом, что распределение будет иметь среднее значение 0 и стандартное отклонение 1. Большинство значений будет в диапозоне от -1 до 1. Это стандартная трансформация, и она применима во многих ситуациях.

$$z = \frac{X - u}{s}$$

u — среднее значение (или 0 при `with_mean=False`) и s — стандартное отклонение (или 0 при `with_std=False`)

И StandardScaler и MinMaxScaler очень чувствительны к наличию выбросов. `RobustScaler` использует медиану и основан на *процентилях*. Процентиль — мера, в которой процентное значение общих значений равно этой мере или меньше ее. Например, 90 % значений данных находятся ниже 90-го процентиля, а 10 % значений данных находятся ниже 10-го процентиля. Соответственно, RobustScaler не зависит от небольшого числа очень больших предельных выбросов (outliers). Следовательно, результирующий диапазон преобразованных значений признаков больше, чем для предыдущих скэйлеров и, что более важно, примерно одинаков.

$$z = \frac{X - X_{median}}{IQR}$$

X_{median} — значение медианы, IQR — межквартильный диапазон равный разнице между 75-ым и 25-ым процентилями

Для нашей задачи по определению раковых опухолей обработаем наши 30 признаков с помощью StandardScaler.

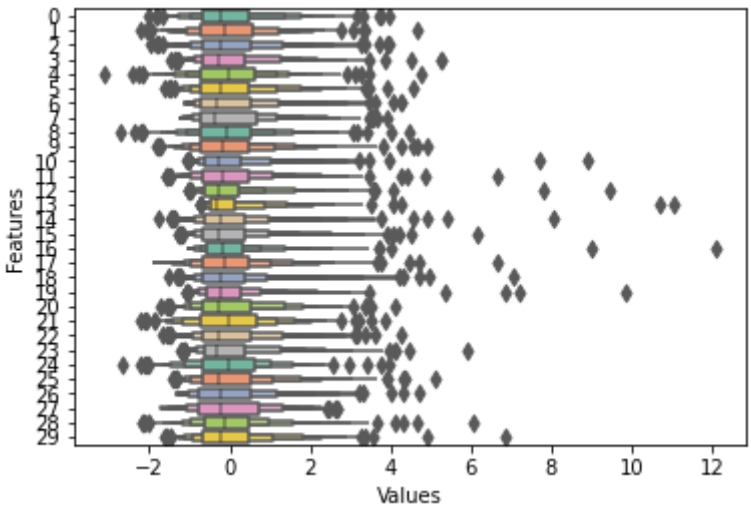
```
X_norm = StandardScaler().fit_transform(X)  # scaled data
```

Видим что они стали намного более сравнимы между собой.

```
pd.DataFrame(X_norm).describe()
```

	0	1	2	3	4	
count	5.690000e+02	5.690000e+02	5.690000e+02	5.690000e+02	5.690000e+02	5.690
mean	-3.162867e-15	-6.530609e-15	-7.078891e-16	-8.799835e-16	6.132177e-15	-1.120
std	1.000880e+00	1.000880e+00	1.000880e+00	1.000880e+00	1.000880e+00	1.000
min	-2.029648e+00	-2.229249e+00	-1.984504e+00	-1.454443e+00	-3.112085e+00	-1.610
25%	-6.893853e-01	-7.259631e-01	-6.919555e-01	-6.671955e-01	-7.109628e-01	-7.470
50%	-2.150816e-01	-1.046362e-01	-2.359800e-01	-2.951869e-01	-3.489108e-02	-2.219
75%	4.693926e-01	5.841756e-01	4.996769e-01	3.635073e-01	6.361990e-01	4.938
max	3.971288e+00	4.651889e+00	3.976130e+00	5.250529e+00	4.770911e+00	4.568

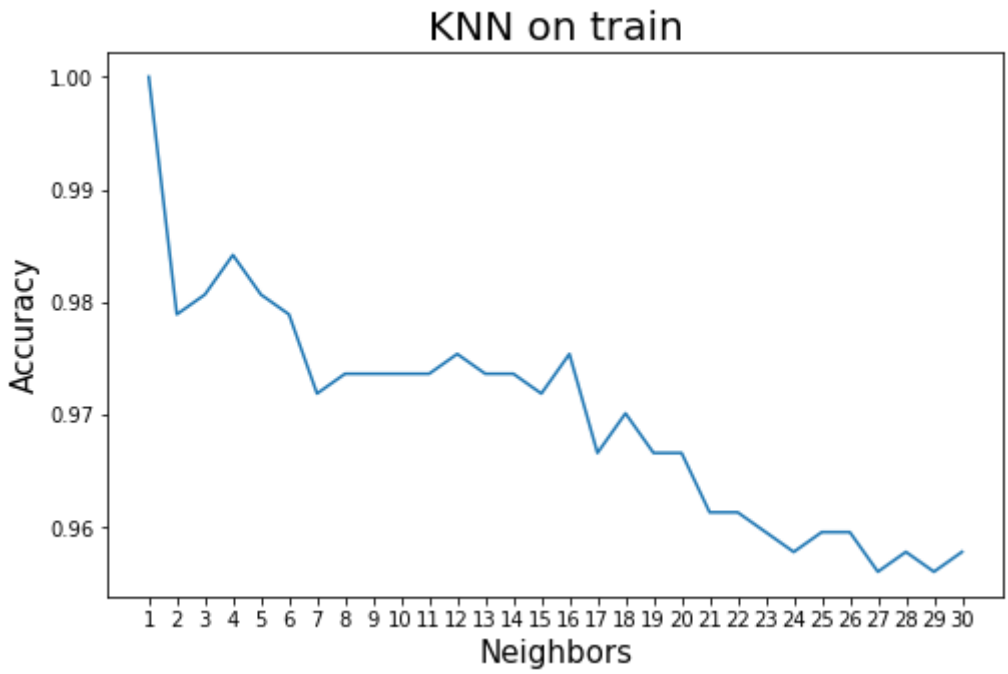
```
ax = sns.boxenplot(data=pd.DataFrame(X_norm),
                   orient="h",
                   palette="Set2")
ax.set(xlabel='Values', ylabel='Features')
plt.show()
```



Переобучение

Теперь обучим kNN для общей выборки данных, при разном значении количества соседей.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

n_nei_rng = np.arange(1, 31)  # array of the number of neighbors

quality = np.zeros(
    n_nei_rng.shape[0]
)

for ind in range(n_nei_rng.shape[0]):  # for all elements
    # create knn for all num neighbors
    knn = KNeighborsClassifier(
        n_neighbors=n_nei_rng[ind]
    )
    knn.fit(X_norm, Y)
    q = accuracy_score(y_pred=knn.predict(X_norm), y_true=Y)  # accuracy
    quality[ind] = q  # fill quality

plt.figure(figsize=(8, 5))
plt.title("KNN on train", size=20)
plt.xlabel("Neighbors", size=15)
plt.ylabel("Accuracy", size=15)
plt.plot(n_nei_rng, quality)
plt.xticks(n_nei_rng)
plt.show()
```

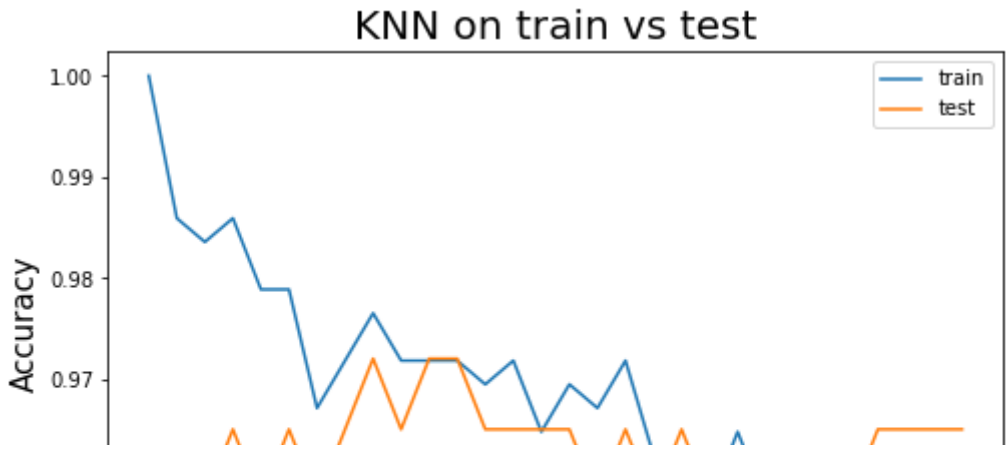


Видим, что качество на 1 соседе - самое лучшее. Но это и понятно - ближайшим соседом элемента из обучающей выборки будет сам объект. Мы просто **запомнили** все объекты.

Если теперь мы попробуем взять какой-то новый образец опухоли и классифицировать его - у нас скорее всего ничего не получится. В таких случаях мы говорим, что наша модель не умеет обобщать (*generalization*).

Для того, чтобы знать заранее обобщает ли наша модель или нет, мы можем разбить все имеющиеся у нас данные на 2 части. Но одной части мы будем обучать классификатор (*train set*), а на другой тестировать насколько хорошо он работает (*test set*).

```
from sklearn.model_selection import train_test_split

# split data to train/test
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=42)

scaler = StandardScaler()
scaler.fit(X_train)
X_train_norm = scaler.transform(X_train)  # scaling data
X_test_norm = scaler.transform(X_test)  # scaling data

n_nei_rng = np.arange(1, 31)
train_quality = np.zeros(n_nei_rng.shape[0])  # quality on train data
test_quality = np.zeros(n_nei_rng.shape[0])  # quality on test data

for ind in range(n_nei_rng.shape[0]):
    knn = KNeighborsClassifier(n_neighbors=n_nei_rng[ind])
    knn.fit(X_train_norm, Y_train)

    # accuracy on train data
    trq = accuracy_score(y_pred=knn.predict(X_train_norm), y_true=Y_train)
    train_quality[ind] = trq

    # accuracy on test data
    teq = accuracy_score(y_pred=knn.predict(X_test_norm), y_true=Y_test)
    test_quality[ind] = teq

# accuracy plot  on train and test data
plt.figure(figsize=(8, 5))
plt.title("KNN on train vs test", size=20)
plt.plot(n_nei_rng, train_quality, label="train")
plt.plot(n_nei_rng, test_quality, label="test")
plt.legend()
plt.xticks(n_nei_rng)
plt.xlabel("Neighbors", size=15)
plt.ylabel("Accuracy", size=15)
plt.show()
```



Вот, теперь мы видим, что 1 сосед был "ложной тревогой". Такие случаи мы называем *переобучением*. Чтобы действительно предсказывать что-то полезное нам надо выбирать число соседей начиная минимум с 3

Neighbors

Кросс-валидация

Каждая модель имеет как ряд **параметров**, которые она меняет в процессе обучения (например, веса модели), так и ряд **гиперпараметров**, которые влияют на то, каким способом модель меняет параметры в процессе обучения.

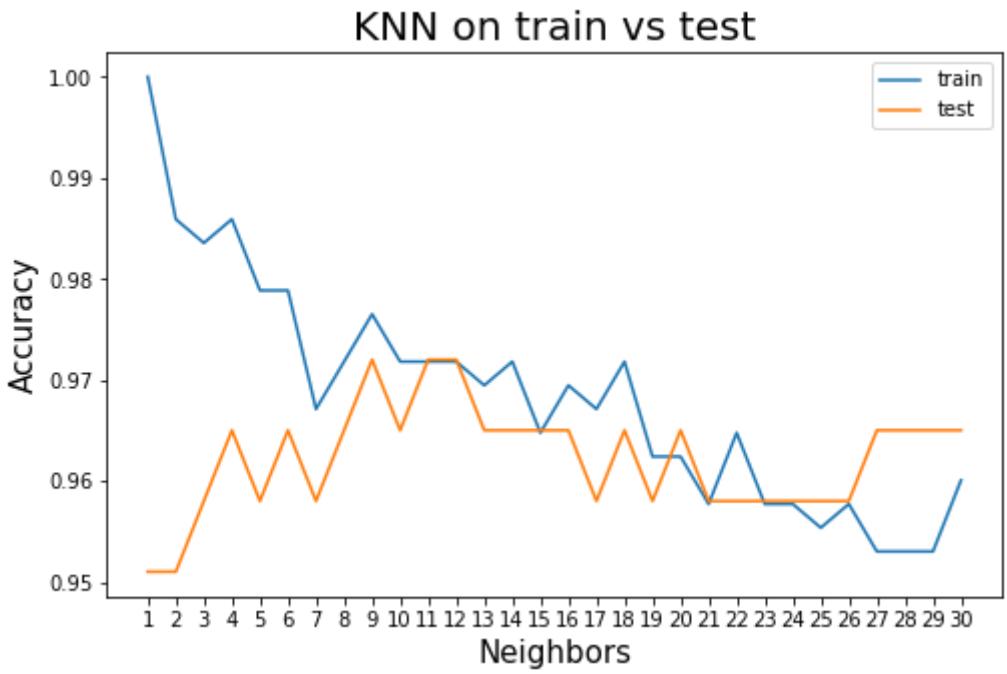
В случае kNN параметры, строго говоря, отсутствует - модель просто запоминает объекты обучающей выборки. Особо упорные могут считать их параметрами.

А вот гиперпараметры есть, даже несколько групп. Какие?

1. число соседей
2. функция, которой считаем расстояние между объектами (L2=eucledian, L1=manhattan)
3. веса, с которыми складываем метки ближайших соседей
4. признаки! (но об этом с вами поговорим позже)
5. сама модель - мы могли выбрать не kNN, а нагуглить что-нибудь другое

Посмотрим еще раз на график, который нарисовали на прошлом шаге. Какое число соседей считать оптимальным? Метрика явно скачет?

```
plt.figure(figsize=(8, 5))
plt.title("KNN on train vs test", size=20)
plt.plot(n_nei_rng, train_quality, label="train")
plt.plot(n_nei_rng, test_quality, label="test")
plt.xlabel("Neighbors", size=15)
plt.ylabel("Accuracy", size=15)
plt.legend()
plt.xticks(n_nei_rng)
plt.show()
```



Не понятно, насколько результат зависит от того, как нам повезло или не повезло с разбиением данных на обучение и тест. Может оказаться так, что для конкретного разбиения хорошо выбрать k=5, а для другого - k=7.

Кроме того, опять же - фактически, мы сами выступаем в роли модели, которая учит гиперпараметры (а не параметры) под видимую ей выборку.

Представим себе, что у нас есть 10000 моделей, полученных подкручиванием разных гиперпараметров (в том числе, выбором просто разного типа модели). Представим, что все эти модели не работают. Вообще. Представим так же, что каждая модель угадывает класс в задаче разделения на два класса с вероятностью 0.5 (и будем считать, что классы у нас сбалансированны - то есть 50% одного класса и 50% другого). Опять же, понятно, что классификация такими моделями ничем не лучше подбрасывания монетки.

```
def guess_model(Y_real):
    # array with values True/False, for random choice by mask
    guessed = np.random.choice(
        [True,False],
        size=Y_real.shape[0],
        replace=True,
    )
    Y_pred = np.zeros_like(Y_real)  # zeros array, shape Y_real

    # with mask 'guessed' assign values
    Y_pred[guessed] = Y_real[guessed]
    Y_pred[~guessed] = 1 - Y_real[~guessed]
    return Y_pred
```

```
models_num = 10000  # num of experiments
best_quality = 0.5  # порог качества по точности
```

```

# array with random values in range 0, 1
Y_real = np.random.choice(
    [0, 1], size=250, replace=True
)

for i in range(models_num):  # for all expirements
    Y_pred = guess_model(Y_real)  # predicted values
    q = accuracy_score(y_pred=Y_pred, y_true=Y_real)  # accuracy
    if q > best_quality:
        best_quality = q
print(f"Best result {best_quality}")
```

```
Best result 0.608
```

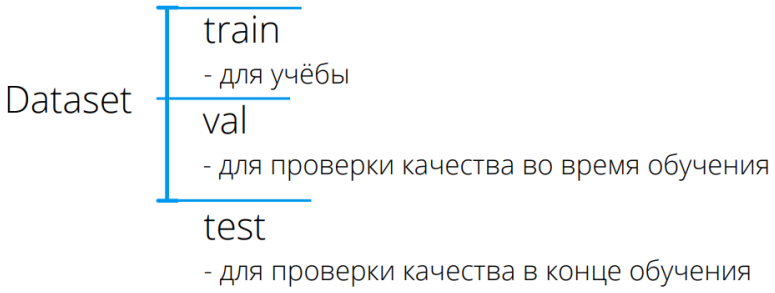
То есть мы перебором всех возможных моделей вполне можем получить для абсолютно бесполезной модели приемлемое качество

Получается, что если подбирать гиперпараметры модели на *train set*, то:

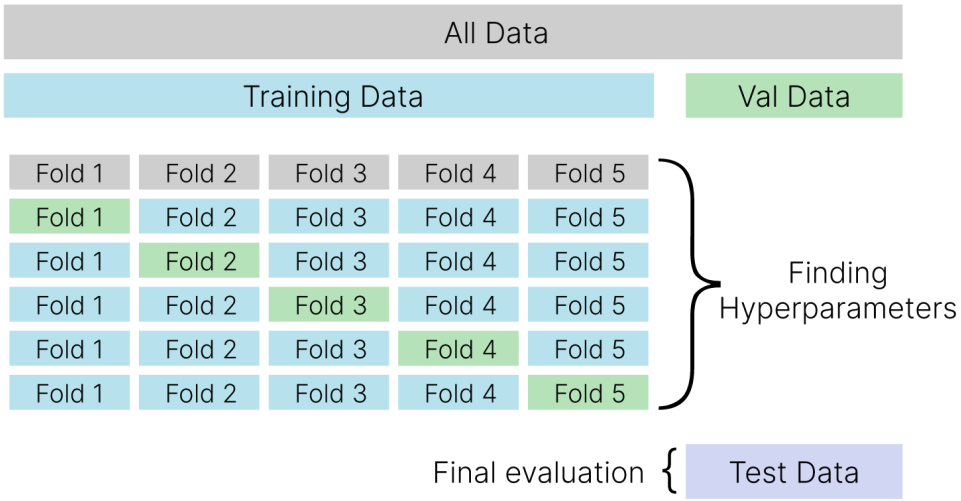
1. можно переобучиться, просто на более "высоком" уровне. Особенно если гиперпараметров у модели много и все они разнообразны
2. нельзя быть уверенным, что выбор параметров не зависит от разбиения на обучение и тест

Поэтому мы:

1. подбираем гиперпараметры моделей на отдельном датасете, называмым валидационным. Получаем мы его разбиением обучающего датасета на собственно обучающий и валидационный



2. чаще всего делаем несколько таких разбиений по какой-то схеме, чтобы получить уверенность оценок качества для моделей с разными гиперпараметрами - **кросс-валидация**



Часто применяется следующий подход, называемый [K-Fold кросс-валидацией](#):

Берется тренировочная часть датасета, разбивается на части - блоки. Дальше мы будем использовать для проверки первую часть (Fold 1), а на остальных учиться. И так последовательно для всех частей. В результате у нас будут информация о точности для разных фрагментов данных и уже на основании этого можно понять, насколько значение этого параметра, который мы проверяем, зависит или не зависит от данных. То есть если у нас от разбиения точность при одном и том же K меняться не будет, значит мы подобрали правильное K. Если она будет сильно меняться в зависимости от того, на каком куске данных мы проводим тестирование, значит, надо попробовать другое K и если ни при каком не получилось - то это такие данные.

Подберем параметры для модели с помощью **GridSearchCV**.

GridSearchCV – это инструмент для автоматического подбирания параметров для моделей машинного обучения. GridSearchCV находит наилучшие параметры, путем обычного перебора: он создает модель для каждой возможной комбинации параметров

```
from sklearn.model_selection import GridSearchCV, KFold
from IPython.display import clear_output
"""
Parameters for GridSearchCV:
estimator – model
cv – num of fold to cross-validation splitting
param_grid – parameters names
scoring – metrics
n_jobs - number of jobs to run in parallel, -1 means using all processors.
"""
model = GridSearchCV(
    estimator=KNeighborsClassifier(),
    cv=KFold(3, shuffle=True, random_state=42),
    param_grid={
        "n_neighbors": np.arange(1, 31),
        "metric": ["euclidean", "manhattan"],
```

```
        "weights": ["uniform", "distance"],
    },
    scoring="accuracy",
    n_jobs=-1,
)
model.fit(X_train_norm, Y_train)
clear_output()
```

Выведем лучшие гиперпараметры для модели, которые подобрали:

```
print("Metric:", model.best_params_["metric"])
print("Num neighbors:", model.best_params_["n_neighbors"])
print("Weigths:", model.best_params_["weights"])
```

```
Metric: manhattan
Num neighbors: 4
Weigths: distance
```

Объект GridSearchCV можно использовать как обычную модель

```
from sklearn.metrics import balanced_accuracy_score

Y_pred = model.predict(X_test_norm)
print(f"Percent correct predictions {np.round(accuracy_score(y_pred=Y_pred, y_true=Y_te
print(f"Percent correct predictions(balanced classes) {np.round(balanced_accuracy_score
```

```
Percent correct predictions 96.5 %
Percent correct predictions(balanced classes) 96.46 %
```

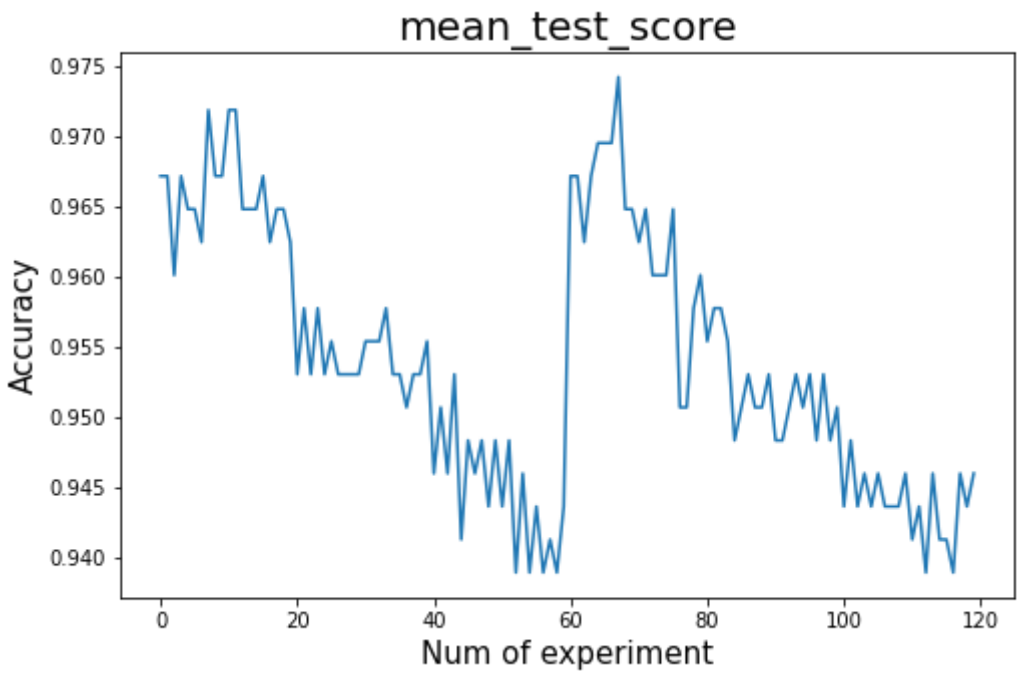
Мы можем извлечь дополнительные данные о кроссвалидации и по ключу обратиться к результатам всех моделей

```
list(model.cv_results_.keys())
```

```
['mean_fit_time',
 'std_fit_time',
 'mean_score_time',
 'std_score_time',
 'param_metric',
 'param_n_neighbors',
 'param_weights',
 'params',
 'split0_test_score',
 'split1_test_score',
 'split2_test_score',
 'mean_test_score',
 'std_test_score',
 'rank_test_score']
```

Выведем для примера mean_test_score:

```
plt.figure(figsize=(8, 5))
plt.plot(model.cv_results_["mean_test_score"])
plt.title("mean_test_score", size=20)
plt.xlabel("Num of experiment", size=15)
plt.ylabel("Accuracy", size=15)
plt.show()
```



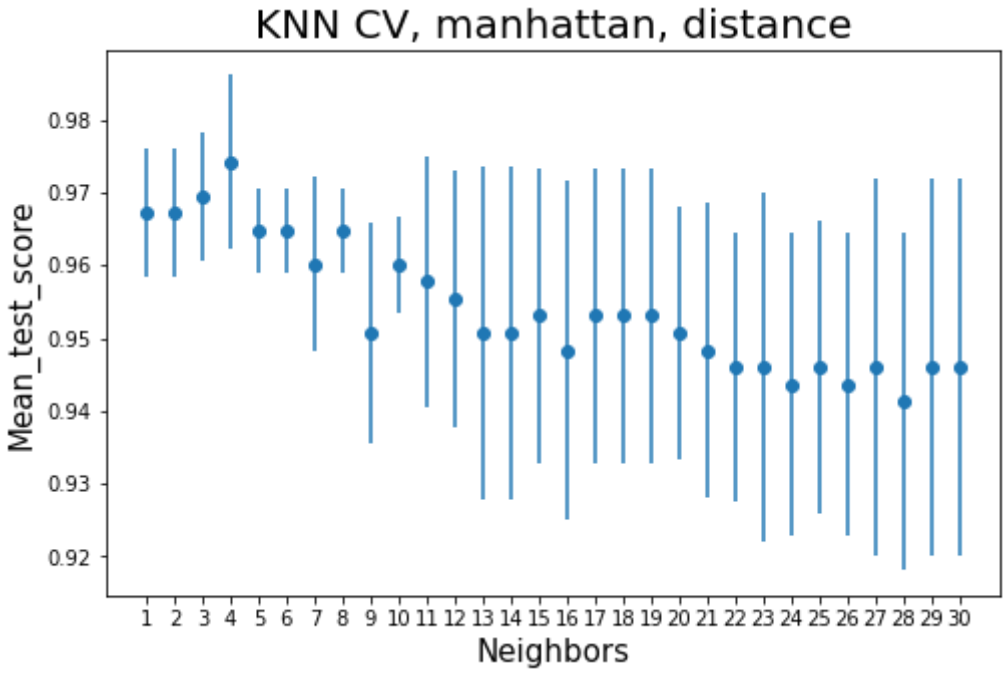
Построим, например, при фиксированных остальных параметрах (равных лучшим параметрам), качество модели на валидации в зависимости от числа соседей

```
selected_means = []
selected_std = []
n_nei = []
for ind, params in enumerate(model.cv_results_["params"]):
    if (
        params["metric"] == model.best_params_["metric"]
        and params["weights"] == model.best_params_["weights"]
    ):
        n_nei.append(params["n_neighbors"])
        selected_means.append(model.cv_results_["mean_test_score"][ind])
        selected_std.append(model.cv_results_["std_test_score"][ind])
```

Построим error bar, для сравнения разброса ошибки при разном количестве соседей Neighbors

```
plt.figure(figsize=(8, 5))
plt.title(f"KNN CV, {params['metric']}, {params['weights']}", size=20)
plt.errorbar(n_nei, selected_means, yerr=selected_std, linestyle="None", fmt="-o")
plt.xticks(n_nei)
plt.ylabel("Mean_test_score", size=15)
```

```
plt.xlabel("Neighbors", size=15)

plt.show()
```



Видим, что на самом деле большой разницы в числе соседей и нет.

Можно ли делать только кросс-валидацию (без теста)?

Нет, нельзя. Кросс-валидация не до конца спасает от подгона параметров модели под выборку, на которой она проводится. Оценка конечного качества модели должно производиться на отложенной тестовой выборке. Если у вас очень мало данных, можно рассмотреть вложенную кросс-валидацию, Речь об этом пойдет позже, в последующих лекциях. Но даже в этом случае придется анализировать поведение модели, чтобы показать, что она учит что-то разумное. Кстати, вложенную кросс-валидацию можно использовать, чтобы просто получить более устойчивую оценку поведения модели на тесте.

Линейный классификатор

Тестовое изображение

56	32	10	18
90	23	128	133
24	26	178	200
2	0	255	220

VS

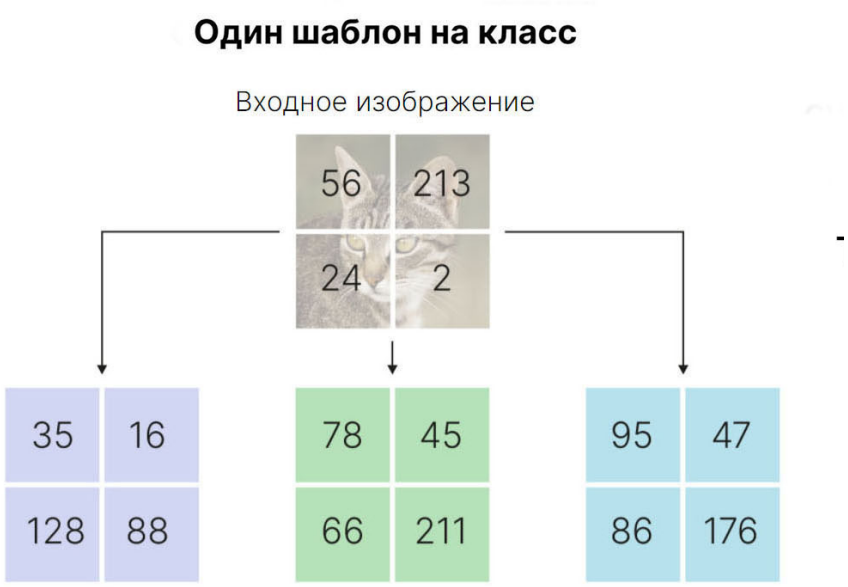
Тренировочное изображение

10	20	24	17
8	10	89	100
12	16	178	170
4	32	233	112

Давайте подумаем, как избавиться от проблемы со скоростью K-Nearest Neighbors. Реализуя метод ближайшего соседа, мы сравнивали одно изображение со всеми, при этом мы предполагали, что изображения из одного класса будут чем-то похожи друг на друга, и поэтому разница будет меньше. Метод как-то работал.

Чтобы ускорить этот процесс, мы можем взять весь блок изображений (справа), который относится, например, к машинам и усредним. Таким образом, получим некоторый шаблон для класса “автомобиль”. Скорее всего, работать такой подход будет не слишком здорово, но зато вместо тысяч изображений (в данном случае 50000) появится одно. Возможно, это поможет сильно сэкономить время.

Переход к сравнению с шаблоном



Идея в следующем: вместо того, чтобы сравнивать каждое изображение со всеми остальными по очереди, будем сравнивать изображения с шаблоном для каждого класса. Их будет всего 10 для данного датасета. Этот подход позволит радикально увеличить скорость.

Проверим, что получится из этой идеи.

скачиваем CIFAR10 в архиве

```
from IPython.display import clear_output
!wget https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
!tar -xzf cifar-10-python.tar.gz
clear_output()
```

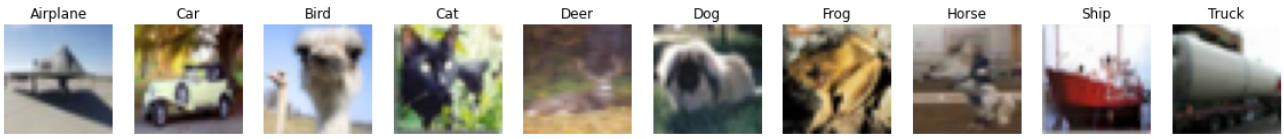
Загрузили архив в память целиком, используя для этого фрагмент кода с сайта CIFAR10.

```
import numpy as np
import pickle

def unpickle(file):
    with open(file, "rb") as fo:
        dict = pickle.load(fo, encoding="bytes")
    return dict

path = '/content/cifar-10-batches-py/'

X_train = np.zeros((0, 3072))
Y_train = np.array([])
for i in range(1, 6):
    raw = unpickle(f"{path}/data_batch_{i}")
    X_train = np.append(X_train, np.array(raw[b"data"]), axis=0)
```

```
    Y_train = np.append(Y_train, np.array(raw[b"labels"]), axis=0)

test = unpickle(f"{path}/test_batch")
X_test = np.array(test[b"data"])
Y_test = np.array(test[b"labels"])

labels_eng = [
    "Airplane",
    "Car",
    "Bird",
    "Cat",
    "Deer",
    "Dog",
    "Frog",
    "Horse",
    "Ship",
    "Truck",
]

print(f"X_train shape: {X_train.shape}, Y_train shape: {Y_train.shape}")
print(f"X_train shape : {X_test.shape}, Y_test shape: {Y_test.shape}")
```

```
X_train shape: (50000, 3072), Y_train shape: (50000,)
X_train shape : (10000, 3072), Y_test shape: (10000,)
```

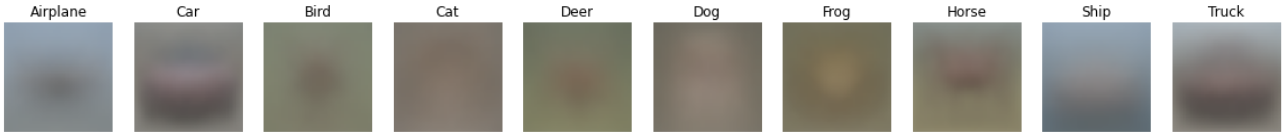
```
from matplotlib.pyplot import imshow
import matplotlib.pyplot as plt

plt.figure(figsize = (20.0, 20.0))
for i in range(10):  # for all classes(0 to 9)
    # get indexes for each class
    label_indexes = np.where(Y_train == i)[0]
    index = np.random.choice(label_indexes)
    # reshape for vizualization
    img = X_train[index].reshape(
        32, 32, 3, order="F"
    )
    img = np.rot90(img, k=3)
    plt.subplot(1, 10, i + 1)
    plt.title(labels_eng[i])
    plt.axis("off")
    imshow(img.astype("uint8"))
```



Для начала создадим шаблоны и визуализируем их.

```
templates = []
meta = unpickle(f"{path}/batches.meta")  # load data
labels = meta[b"label_names"]

for i in range(len(labels)):
    indexes = np.where(Y_train == i)  # index for each class
    mask = np.zeros(len(Y_train), dtype=bool)  # mask with False values
    mask[indexes, ] = True  # change values to True
    imgs = X_train[mask]  # all images for each class
    mn = np.mean(imgs, 0)  # mean values for all images for each class
    templates.append(mn)
```

```
plt.figure(figsize = (20.0, 2.0))

def show_templates(templates, labels):
    for i, template in enumerate(templates):
        img = template.reshape(3, 32, 32).transpose(1, 2, 0).astype(int)
        plt.subplot(1, len(labels), i + 1)
        plt.title(labels_eng[i])
        plt.axis("off")
        imshow(img)

show_templates(templates, labels)
```



Напишем классификатор, который будет сравнивать изображение с шаблоном, который генерируется во время обучения.

```
class TemplateBasedClassifier():
    def __init__(self):
        self.templates = []
        meta = unpickle(f"{path}/batches.meta")
        self.labels = meta[b"label_names"]

    def fit(self, X_train, Y_train):
        self.templates = []
        for label_num in range(len(self.labels)):
            indexes = np.where(Y_train == label_num)
            mask = np.zeros(len(Y_train), dtype=bool)
            mask[indexes, ] = True
            imgs = X_train[mask]
            mn = np.mean(imgs, 0)
            self.templates.append(mn)

    def forward(self, x):
```

```
        distances = np.sum(
            np.abs(self.templates - x), axis=1)  # compute distance score
        return np.argmin(distances)  # return minimum score index
```

Теперь у нас есть 10 шаблонов, с которыми мы будем сравнивать изображение и на основании этого делать предсказание. На этих шаблонах можно увидеть очертания объектов.

Рассмотрим сравнение на конкретном примере с кошкой. Построим изображения картинки, шаблона, а затем посчитаем расстояние L1 между ними. Чем желтее цвет - тем больше изображение похоже на шаблон

```
from mpl_toolkits.axes_grid1 import make_axes_locatable

fig,ax = plt.subplots(ncols=3, figsize=(10,5))

indexes = np.where(Y_train == 3) # class 3(cat) indexes
mask = np.zeros(len(Y_train), dtype=bool) # mask with False values
mask[indexes, ] = True # assign True for 3 class
imgs = X_train[mask] # images class 3(cat)

img = imgs[0].reshape(3, 32, 32).transpose(1, 2, 0).astype(int)
template = templates[3].reshape(3, 32, 32).transpose(1, 2, 0).astype(int)
residual = np.mean(np.abs(img-template), axis = -1) # calculate the difference  and ave

ax[0].imshow(img)
ax[1].imshow(template)
r_plot = ax[2].imshow(residual, cmap='inferno_r')

divider = make_axes_locatable(ax[2])
cax = divider.append_axes('right', size='5%', pad=0.05)
fig.colorbar(r_plot, cax=cax, orientation='vertical', label='L1')

ax[0].set_title('Image')
ax[1].set_title('Template')
ax[2].set_title('D = Image - Template')

for a in ax:
    a.axis('off')
```

Сделаем предсказание и замерим время

```
model = TemplateBasedClassifier()
model.fit(X_train, Y_train)
```

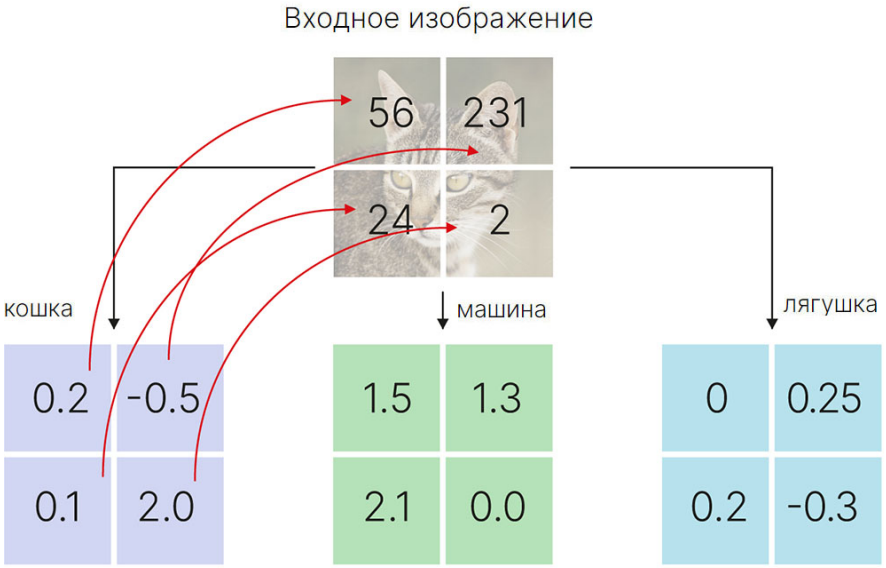
```
import time

def validate(model, X_test, Y_test, noprint=False):
    correct = 0  # num of correct predictions
    for i, img in enumerate(X_test):  # for all images
        index = model.forward(img)  # predict class
        correct += (
            1 if index == Y_test[i] else 0
        )
        if noprint is False:
            if i > 0 and i % 1000 == 0:
                print(
                    "Accuracy {:.3f}".format(correct / i)
                )
    return correct / len(Y_test)

start = time.perf_counter()
accuracy= validate(model, X_test, Y_test)
tm = time.perf_counter() - start
print(
    "\nAccuracy {:.2f} Train {:d} /test {:d} in {:.1f} sec. speed {:.2f} samples per se
        accuracy, len(X_train), len(X_test), tm, len(X_test) / tm
    )
)
```

```
Accuracy 0.286
Accuracy 0.280
Accuracy 0.279
Accuracy 0.275
Accuracy 0.274
Accuracy 0.272
Accuracy 0.273
Accuracy 0.272
Accuracy 0.270

Accuracy 0.27 Train 50000 /test 10000 in 2.6 sec. speed 3897.04 samples per second.
```

Если бы мы обучали KNN на таком количестве данных, это заняло бы у нас 2 часа.

Переход к весам

Умножение вместо вычитания - для чего?



$$\sum \quad 0{,}2\times56-0{,}5\times231+0{,}1\times24+2\times2 = \mathbf{-97.9}$$

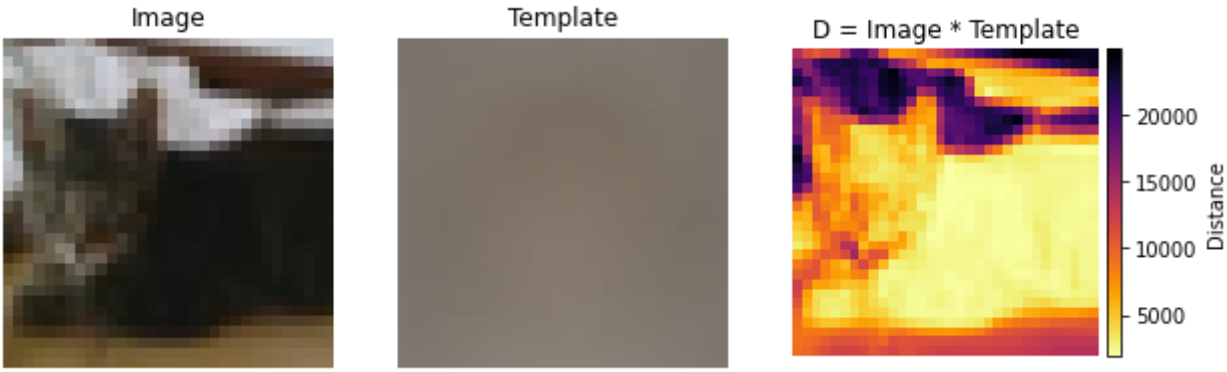
В сравнении с шаблонами, мы вычитали нашу картинку из шаблона и таким образом смотрели, насколько они похожи (L1). Теперь давайте сделаем следующий, пока ничем не обоснованный, ход: оставим всё, как было, но заменим вычитание умножением. Логика в том, что не все пиксели одинаково важны. Вероятно, если на изображении совпадут какие-то пиксели, которые отвечают, например, за глаза кошки, это будет намного важнее, чем фон, который может быть точно таким же у собаки. Здесь будут какие-то важные особенности, которым можно придать больший вес.

Если мы распишем в виде формул наложение шаблона на картинку таким образом с умножением, то получается, что мы скалярно перемножаем два вектора. Подробнее про скалярное произведение векторов. Для того, чтобы получить вектор из изображения размерностью 32x32x3, достаточно "выпрямить" его, получим вектор размерностью 1x3072.

Опять рассмотрим сравнение на конкретном примере с кошкой. Построим изображения картинки, шаблона, а затем посчитаем расстояние между ними, используя наш новый метод перемножения. Чем желтее цвет - тем больше изображение похоже на шаблон

```
fig,ax = plt.subplots(ncols=3, figsize=(10,5))

indexes = np.where(Y_train == 3)
mask = np.zeros(len(Y_train), dtype=bool)
mask[indexes, ] = True
imgs = X_train[mask]

img = imgs[0].reshape(3, 32, 32).transpose(1, 2, 0).astype(int)
template = templates[3].reshape(3, 32, 32).transpose(1, 2, 0).astype(int)
```

```
residual = np.mean(np.abs(img*template),-1)

ax[0].imshow(img)
ax[1].imshow(template)
r_plot = ax[2].imshow(residual, cmap='inferno_r')

divider = make_axes_locatable(ax[2])
cax = divider.append_axes('right', size='5%', pad=0.05)
fig.colorbar(r_plot, cax=cax, orientation='vertical', label='Distance')

ax[0].set_title('Image')
ax[1].set_title('Template')
ax[2].set_title('D = Image * Template')

for a in ax:
    a.axis('off')
```

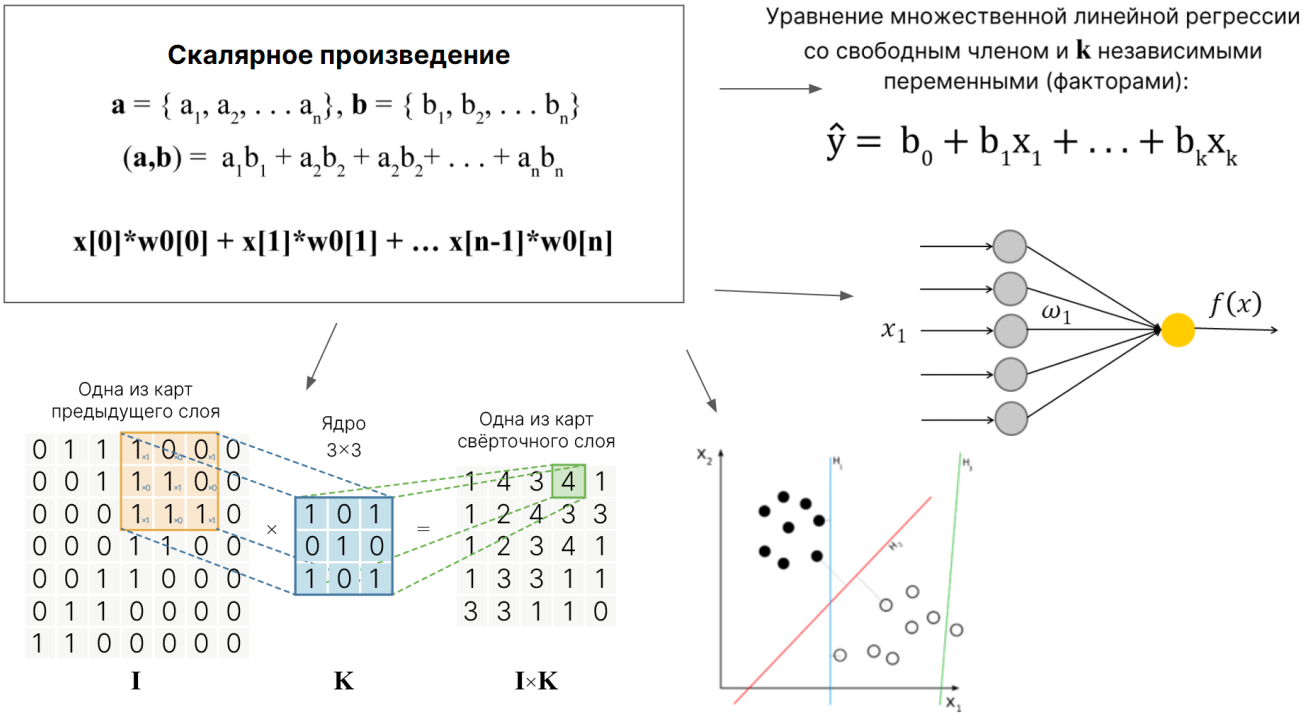


Математическая запись

Обозначим входное изображение как x, а шаблон для первого из классов как w_0.

Элементы пронумеруем подряд 1,2,3 ... n. То есть развернем матрицу пикселей изображения в вектор.

Тогда результат сравнения изображения с этим шаблоном будет вычисляться по формуле: $x[0] * w0[0] + x[1] * w0[1] + \ldots x[n-1] * w0[n-1]$



Эта простая модель лежит в основе практически всех сложных, которые мы будем рассматривать дальше. Внутри мы будем также пользоваться скалярным произведением.

В дальнейшем мы будем проходить сверточные сети, они работают очень похоже: мы тоже накладываем шаблон на некоторую матрицу и перемножаем элементы, затем складываем. Единственное отличие – обычно ядро свертки меньше, чем размер самого изображения.

Support Vector Machine (метод опорных векторов)

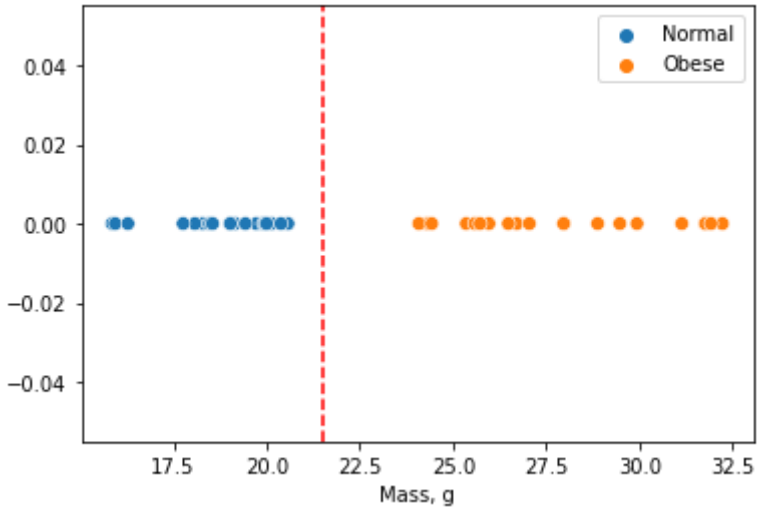
[Отличное видео про SVM от Stat Quest которое все объясняет]

1D классификация

Рассмотрим одномерный пример. У нас есть данные по массе мышей. Часть из них определена как нормальные, а часть как мыши с ожирением. Чтобы их отделить друг от

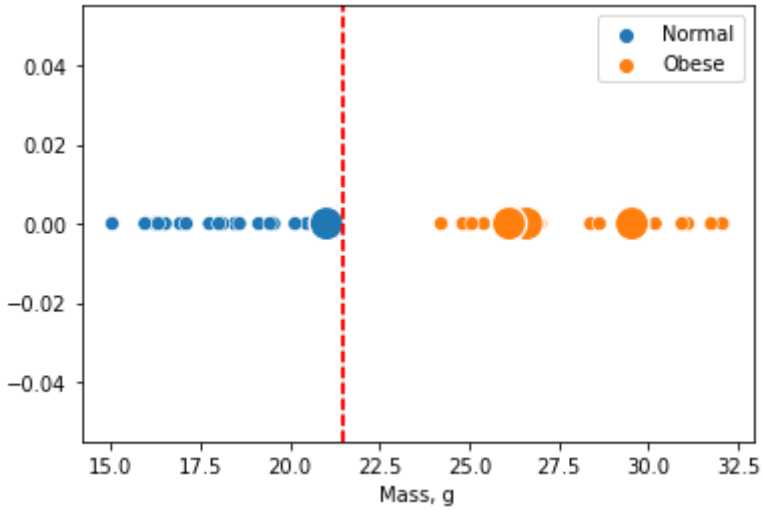
друга, нам достаточно одного критерия. Мы можем посмотреть на график, и визуально

```
import seaborn as sns

def generate_data(total_len=40):
    X = np.hstack([np.random.uniform(14,21, total_len//2),
                   np.random.uniform(24,33, total_len//2)])
    Y = np.hstack([np.zeros(total_len//2),
                   np.ones(total_len//2)])
    return X,Y

def plot_data(X, Y, total_len=40, s=50, threshold=21.5):
    ax = sns.scatterplot(x=X, y=np.zeros(len(X)), hue=Y, s=s)
    ax.axvline(threshold, color='red', ls='dashed')

    handles, labels = ax.get_legend_handles_labels()
    ax.legend(handles, ['Normal', 'Obese'])
    ax.set(xlabel='Mass, g');
    return ax

total_len = 40
X, Y = generate_data(total_len=total_len)
ax = plot_data(X, Y, total_len=total_len)
```



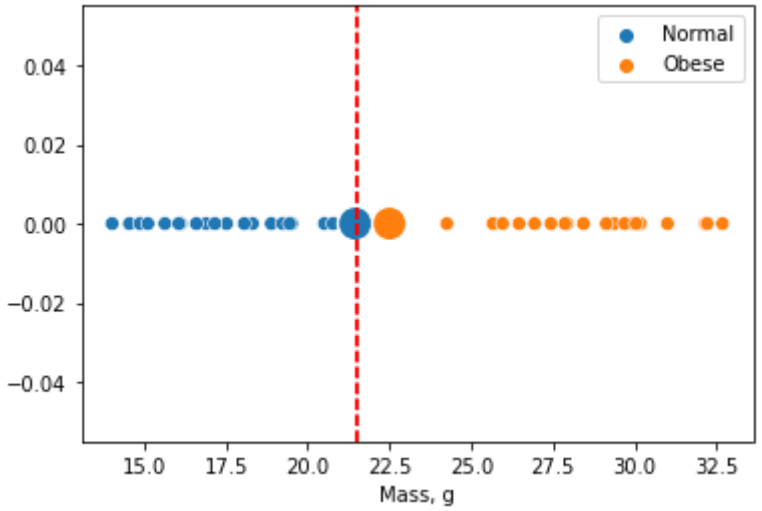
Теперь пользуясь нашим простым критерием, попробуем классифицировать каких-то новых мышей

```
X_test = np.random.uniform(14,30, 5)

def classify(X, threshold=21.5):
    y = np.zeros_like(X)
    y[X > threshold] = 1
    return y

total_len = 40
X, Y = generate_data(total_len=total_len)
ax = plot_data(X, Y, total_len=total_len)
ax = plot_data(X_test, classify(X_test), total_len=total_len, s=300)
```



Но что если наши мыши находятся тут?

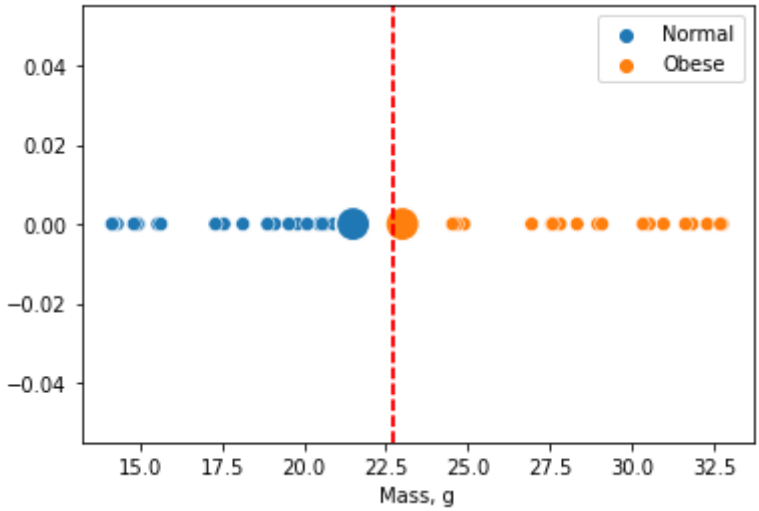
```
X_test = np.array([21.45, 22.5])

total_len = 40
X, Y = generate_data(total_len=total_len)
ax = plot_data(X, Y, total_len=total_len)
ax = plot_data(X_test, classify(X_test), total_len=total_len, s=300)
```



С точки зрения нашего классификатора - все четко. Больше порогового значения - значит перевес, меньше - значит нормальные. Но с точки зрения здравого смысла, логичнее было бы классифицировать обоих мышей как нормальных, так как они значительно ближе к нормальным, чем к ожиревшим.

Вооружившись этим новым знанием, попробуем классифицировать наших отъевшихся мышек по-умному. Возьмем крайние точки в каждом класстере. И в качестве порогового значения будем использовать среднее между ними

```
X, Y = generate_data(total_len=total_len)
normal_limit = X[Y==0].max() # extreme point for 'normal'
obese_limit = X[Y==1].min() # extreme point for 'obese'

threshold = np.mean([normal_limit, obese_limit]) # separated with mean value

X_test = np.array([21.5, 23])
ax = plot_data(X, Y, total_len=total_len, threshold=threshold)
ax = plot_data(X_test, classify(X_test, threshold=threshold), total_len=total_len, s=36
```



Мы можем посчитать, насколько наша мышь близка к тому, чтобы оказаться в другом классе. Такое расстояние называется **margin**. И оно считается как
$\text{margin} = |\text{threshold} - \text{observation}|$

```
margins = np.abs(X_test - threshold)
print(margins)
```

```
[1.22257586 0.27742414]
```

Соответственно, если мы посчитаем margins для наших крайних точек `normal_limit` и `obese_limit`, мы найдем самое большое возможное значение margin для нашего классификатора

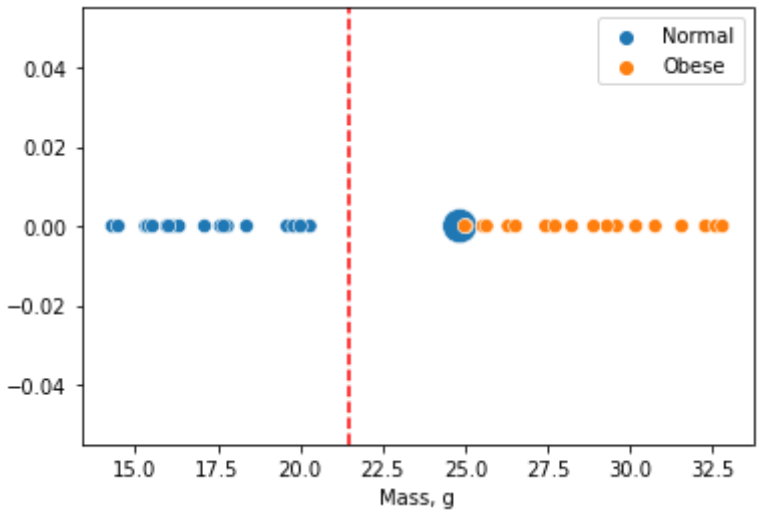
```
margin_0 = np.abs(normal_limit - threshold)
margin_1 = np.abs(obese_limit - threshold)
print(margin_0, margin_1)
```

```
1.8068578531975383 1.8068578531975419
```

Такой классификатор, мы называем **Maximum Margin Classifier**. Он хорошо работает в случае, когда все данные размечены аккуратно. Теперь рассмотрим более реалистичный пример, где что-то пошло не так

```
def generate_realistic_data(total_len=40):
```

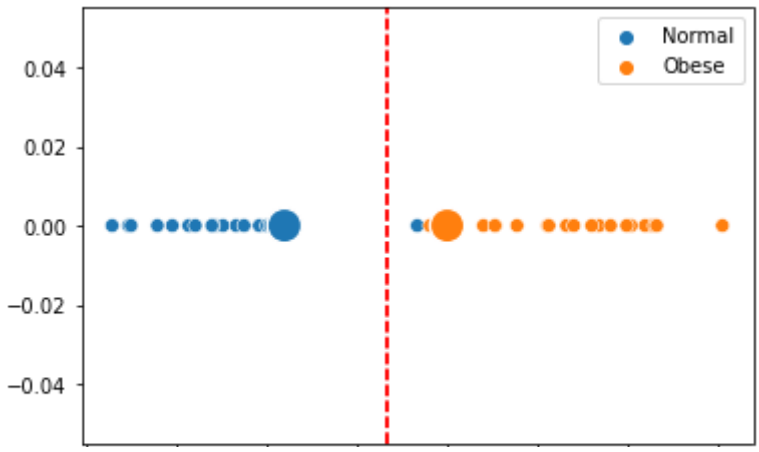
```
    X = np.hstack([np.random.uniform(14,21, total_len//2), np.random.uniform(24,33, tot
    Y = np.hstack([np.zeros(total_len//2), np.ones(total_len//2)])
    indx = np.where(X == X[Y==1].min())[0]
    Y[indx] = 0
    s = np.ones_like(X)*50
    s[indx] = 300
    return X,Y,s

total_len = 40
X,Y,s = generate_realistic_data(total_len=total_len)
ax = plot_data(X, Y, total_len=total_len, s=s)
```



В таком случае, наш **Maximum Margin Classifier** работать не будет. Исходя из этого, мы можем прийти к выводу, что наш классификатор очень чувствителен к выбросам. Давайте подумаем можно ли это как-то исправить?

Например, мы можем разрешить нашему классификатору ошибаться. Если мы будем использовать в качестве порогового значения не самое крайнее, а следующее за ним - мы промажем в классификации конкретно этой странной точки. Но в целом будем правы. Margin определенный таким образом, называется **Soft margin**.

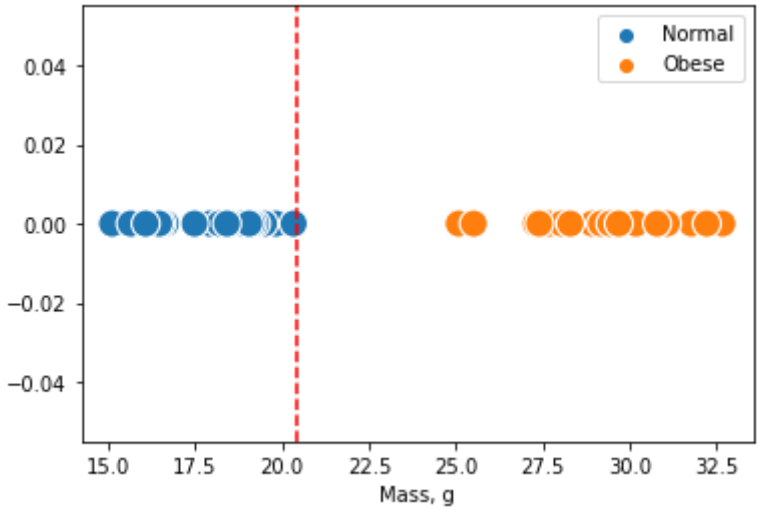
```
total_len = 40
X, Y, s = generate_realistic_data(total_len=total_len)

normal_limit = np.sort(X[Y==0])[-2]
obese_limit = np.sort(X[Y==1])[1]

threshold = np.mean([normal_limit, obese_limit])

X_test = np.array([20.5, 25])
ax = plot_data(X, Y, total_len=total_len, threshold=threshold)
ax = plot_data(X_test, classify(X_test, threshold=threshold), total_len=total_len, s=30
```



Но почему мы решили взять именно следующее значение? Почему не через 2? Откуда мы знаем что это лучший из возможных вариантов? А ни откуда не знаем. Чтобы узнать какой из margins лучше, нам стоит численно это проверить и посчитать, сколько раз мы ошибемся, если возьмем в качестве порогового значения между каждой парой точек. Для этого мы вновь воспользуемся *кросс-валидацией*.

```
total_len = 40
X, Y, s = generate_realistic_data(total_len=total_len)

indxs = []
accuracies = []
thresholds = []

X_test, Y_test = generate_data(total_len=total_len)

for i in range(total_len//2):
    for j in range(total_len//2-1):
        normal_limit = np.sort(X[Y==0])[-i]
        obese_limit = np.sort(X[Y==1])[j]

        threshold = np.mean([normal_limit, obese_limit])

        Y_pred = classify(X_test, threshold=threshold)
        accuracy = np.mean(Y_test == Y_pred)
        indxs.append((-i,j))
        accuracies.append(accuracy)
        thresholds.append(threshold)

print(f'Accuracy = {np.max(accuracies)*100}%')
print(f'Best indexes', indxs[np.argmax(accuracies)])

best_treshold = thresholds[np.argmax(accuracies)]
print('Best treshhold value %.2f'% best_treshold)

ax = plot_data(X_test, classify(X_test, threshold=best_treshold),
               total_len=total_len,
               s=200,
               threshold=best_treshold)
```

```
Accuracy = 100.0%
Best indexes (0, 3)
Best treshhold value 20.41
```



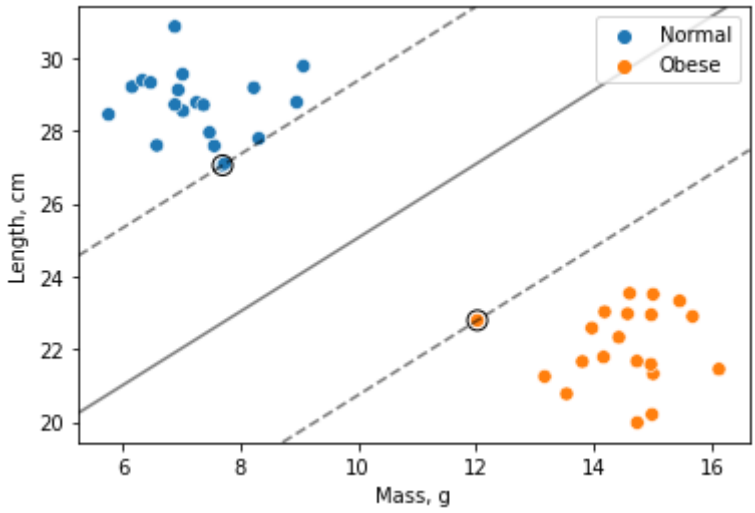
Когда для классификатора используется **Soft Margin** - такой классификатор называют **Soft Margin Classifier** или по другому - **Support Vector Classifier**. По сути это уже SVM.

2D класификация

Теперь рассмотрим пример, где мы измерили не только вес мышей, но и их длину от хвоста до носа. Мы можем вновь применить наш метод Support Vector Classifier, и теперь классы разделяет не одно пороговое значение (по сути, точка), а линия.

```
from sklearn.datasets import make_blobs
from sklearn import svm

def generate_2d_data(total_len=40):
    X, Y = make_blobs(n_samples=total_len, centers=2, random_state=42)
    X[:,0] += 10
    X[:,1] += 20
    return X, Y

def plot_data(X, Y, total_len=40, s=50, threshold=21.5):
    ax = sns.scatterplot(x=X[:,0], y=X[:,1], hue=Y, s=s)
    handles, labels  =  ax.get_legend_handles_labels()
    ax.legend(handles, ['Normal', 'Obese'])
    ax.set(xlabel='Mass, g', ylabel='Length, cm');
    return ax

total_len = 40
X, Y = generate_2d_data(total_len=total_len)
ax = plot_data(X, Y, total_len=total_len)

# Code for illustration, later we will understand how it works
# fit the model, don't regularize for illustration purposes
clf = svm.SVC(kernel='linear', C=1000)
clf.fit(X, Y)
```

```

# plot the decision function
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = clf.decision_function(xy).reshape(XX.shape)

# plot decision boundary and margins
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
           linestyles=['--', '-', '--'])
# plot support vectors
ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100,
           linewidth=1, facecolors='none', edgecolors='k')
plt.show()
```

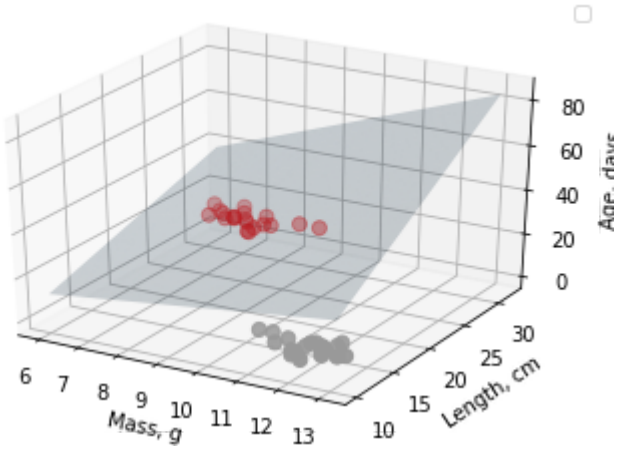


3D классификация

Если мы добавим еще одно измерение - возраст, мы обнаружим, что наши данные стали трехмерными, а разделяет их теперь не линия, а плоскость

```
def generate_3d_data(total_len=40):
    X, Y = make_blobs(n_samples=total_len, centers=2, random_state=42, n_features=3)
    X[:,0] += 10
    X[:,1] += 20
    X[:,2] += 10
    return X, Y

def plot_data(X, Y, total_len=40, s=50, threshold=21.5):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
```

```
    ax.scatter(xs=X[:,0], ys=X[:,1], zs=X[:,2], c=Y, s=s, cmap='Set1')
    # plot the decision function
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # create grid to evaluate model
    xx = np.linspace(xlim[0], xlim[1], 30)
    yy = np.linspace(ylim[0], ylim[1], 30)
    YY, XX = np.meshgrid(yy, xx)
    ax.plot_surface(XX, YY, XX*YY*0.2, alpha=0.2)
    handles, labels  =  ax.get_legend_handles_labels()
    ax.legend(handles, ['Normal', 'Obese'])
    ax.set(xlabel='Mass, g', ylabel='Length, cm', zlabel='Age, days');
    return ax

total_len = 40
X, Y = generate_3d_data(total_len=total_len)
ax = plot_data(X, Y, total_len=total_len)

```



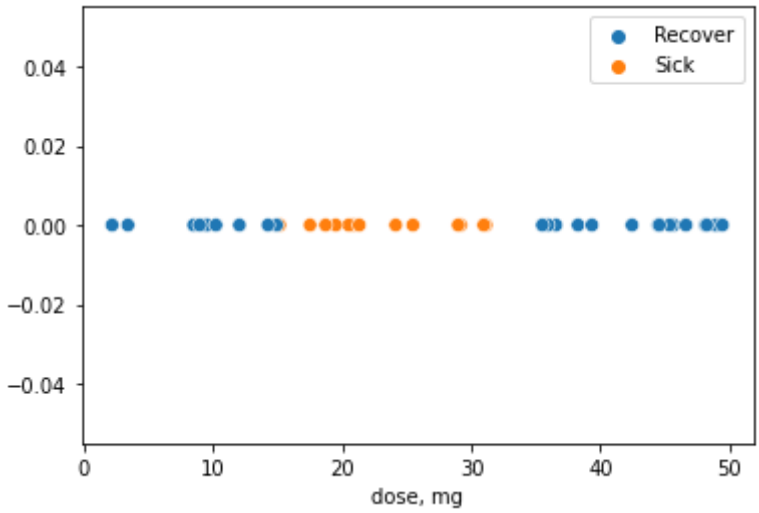
Соответственно, если бы у нас было 4 измерения и больше (например: вес, длинна, возраст, кровяное давление), то многомерная плоскость которая бы разделяла наши классы - называлась бы **гиперплоскость** (рисовать мы ее, конечно же, не будем). Чисто технически, и точка, и линия - тоже гиперплоскости. Но все же гиперплоскостью принято называть то, что нельзя нарисовать на бумаге.

SVM

Данные не всегда разделяются так хорошо как в случае нашего мышиного датасета. Например, рассмотрим следующее: у нас есть данные по дозировке лекарства и 2 класса - пациенты, которые поправились, и те, которым лучше не стало

```
def generate_patients_data(total_len=40):
    X = np.random.uniform(0,50, total_len)
```

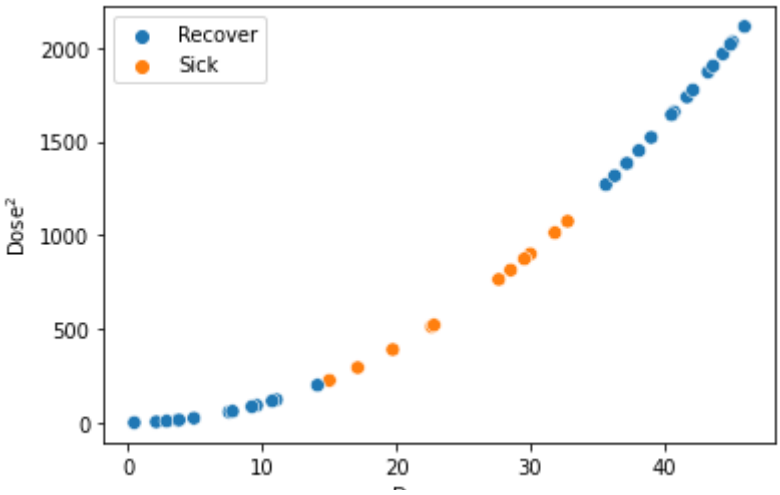
```
    Y = np.zeros_like(X)
    Y[(X > 15) & (X < 35)] = 1
    return X, Y

def plot_data(X, Y, total_len=40, s=50):
    ax = sns.scatterplot(x=X, y=np.zeros(len(X)), hue=Y, s=s)

    handles, labels  =  ax.get_legend_handles_labels()
    ax.legend(handles, ['Recover', 'Sick'])
    ax.set(xlabel='dose, mg');
    return ax

total_len = 40
X, Y = generate_patients_data(total_len=total_len)
ax = plot_data(X, Y, total_len=total_len)
```



Соответственно мы не можем найти такое пороговое значение, которое будет разделять наши классы на больных и здоровых, а следовательно, и Support Vector Classifier работать тоже не будет. Для начала, давайте преобразуем наши данные таким образом, что бы они стали 2-х мерными. В качестве значений по оси Y будем использовать дозу возведенную в квадрат (**доза**^{2}).

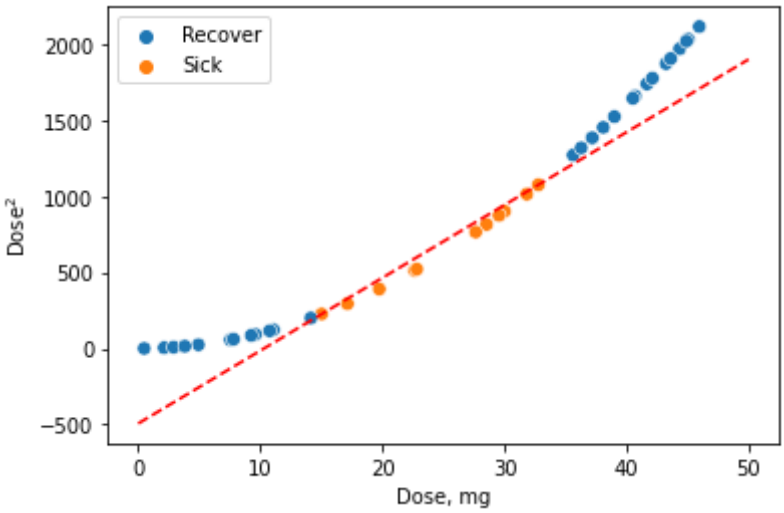
```
def plot_data(X, Y, total_len=40, s=50):
    ax = sns.scatterplot(x=X[0,:], y=X[1,:], hue=Y, s=s)
    handles, labels  =  ax.get_legend_handles_labels()
    ax.legend(handles, ['Recover', 'Sick'])
    ax.set(xlabel='Dose, mg');
    ax.set(ylabel='Dose$^2$');
    return ax

total_len = 40
X_1, Y = generate_patients_data(total_len=total_len)
X_2 = X_1**2
X = np.vstack([X_1, X_2])

```

```
plot_data(X, Y, total_len=40, s=50)
plt.show()
```



Теперь мы можем вновь использовать Support Vector Classifier для классификации

```
plot_data(X, Y, total_len=40, s=50)

x = np.linspace(0,50,50)
xs = [X[0,:][Y==1].min(), X[0,:][Y==1].max()]
ys = [X[1,:][Y==1].min(), X[1,:][Y==1].max()]

# Calculate the coefficients.
coefficients = np.polyfit(xs, ys, 1)

# Let's compute the values of the line...
polynomial = np.poly1d(coefficients)
y_axis = polynomial(x)

# ...and plot the points and the line
plt.plot(x, y_axis, 'r--')
plt.show()
```



Но тут возникает резонный вопрос - почему мы решили возвести в квадрат? Почему не в куб? Или наоборот не извлечь корень? Как нам решить какое преобразование

использовать?

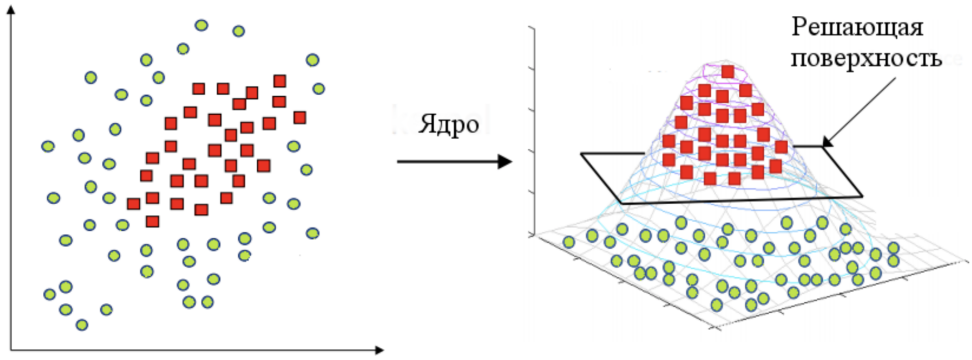
И у нас есть **вторая проблема** - а если перейти надо в пространство очень большой размерности? В этом случае наши данные очень сильно увеличатся в размере.

Комбинация двух проблем дает нам **много радости** - надо перебирать большое число возможных пространств большей размерности

Однако основная фишка Support Vector Machine состоит в том, что внутри он работает на скалярных произведениях. И можно эти скалярные произведения считать, **не переходя в пространство большей размерности**

Для этого SVM использует **Kernel Function**.

Kernel Function может, например, быть полиномом (**Polynomial Kernel Function**), который имеет параметр d - сколько размерностей выбрать.



Примеры ядер :

- $k(x_i, x_j) = (< x_i, x_j > + c)^d$, c, $d \in \mathbb{R}$ - полиномиальное ядро, считает расстояние между объектами в пространстве размерности d
- $k(x_i, x_j) = \frac{1}{z} e^{-\frac{h(x_i, x_j)^2}{h}}$ - радиальная базисная функция RBF

Таким образом, в случае SVM можно легко перебрать много таких пространств на кроссвалидации и выбрать более удобное.

Более того, SVM может проверять пространства признаков бесконечного размера. Если для такого пространства существует kernel function. Иногда такие пространства оказываются очень удобными для решения задач. Часто используют тот же RBF-пространство, приведенное выше. А оно как раз бесконечно-мерное.

Регрессия

Пример простой линейной регрессии

Теперь ненадолго отвлечемся от SVM и рассмотрим другую задачу. В этой задаче мы будем прогнозировать успеваемость студента, в зависимости от количества часов, которые он учил материал. Это простая задача линейной регрессии, поскольку она включает всего две переменные.

Регрессия - это статистический метод, используемый в финансах, инвестировании и других дисциплинах, который пытается определить силу и характер связи между одной зависимой переменной (обычно обозначаемой **Y**) и рядом других переменных (известных как независимые переменные). В задачах регрессии, мы будем пытаться минимизировать ошибку между предсказанием и истинными данными.

"Регрессия" происходит от слова "регресс", которое, в свою очередь, происходит от латинского "regressus" - возвращаться (к чему-либо). В этом смысле регрессия - это техника, которая позволяет "вернуться назад" от беспорядочных, трудно интерпретируемых данных к более четкой и осмысленной модели.

Загрузим датасет

```
!wget https://edunet.kea.su/repo/EduNet-web_dependencies/L02/student_scores.csv
clear_output()
```

Посмотрим, что там в нем. Видим, что у нас есть два признака - часы и результаты

```
import pandas as pd

dataset = pd.read_csv("/content/student_scores.csv")
print(dataset.shape)
dataset.head()
```

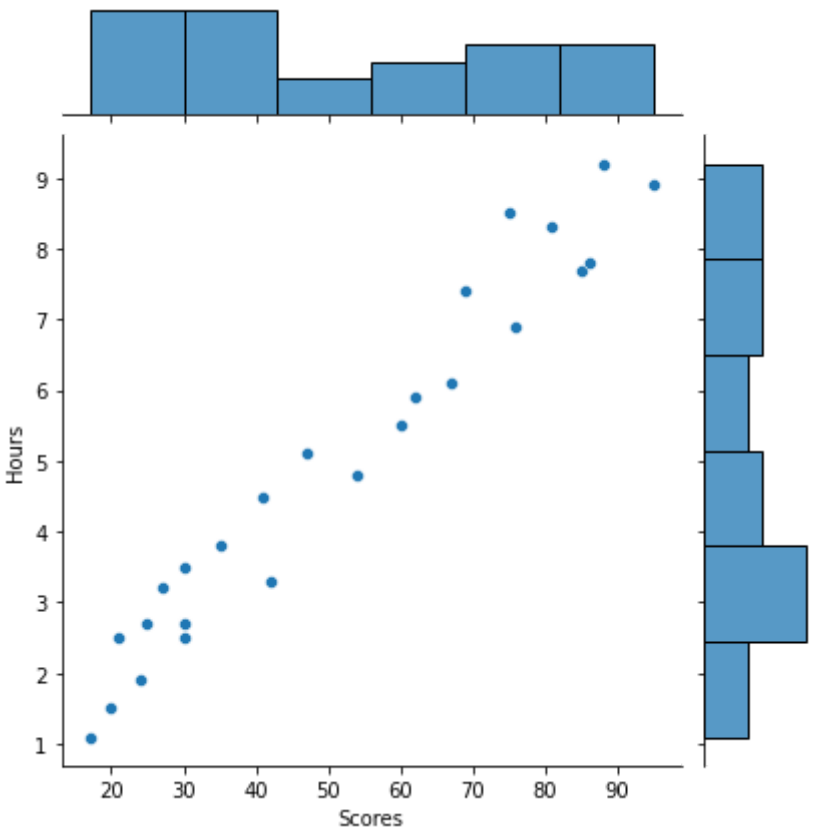
```
(25, 2)
```

	Hours	Scores
0	2.5	21
1	5.1	47
2	3.2	27
3	8.5	75
4	3.5	30

Построим график зависимости одного от другого, а так же отобразим распределения каждой из переменных

```
sns.jointplot(data=dataset, x="Scores", y="Hours")
```

```
plt.show()
```



Разделим наши данные на train и test

```
from sklearn.model_selection import train_test_split

X = dataset.iloc[:, :-1].values # column Hours
Y = dataset.iloc[:, 1].values # column Score

X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size=0.2, random_state=42
)
```

Теперь создадим модель для линейной регрессии. Чтобы не писать с нуля, воспользуемся готовой моделью из библиотеки `sklearn`

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
```

И обучим ее

```
regressor.fit(X_train, Y_train)
```

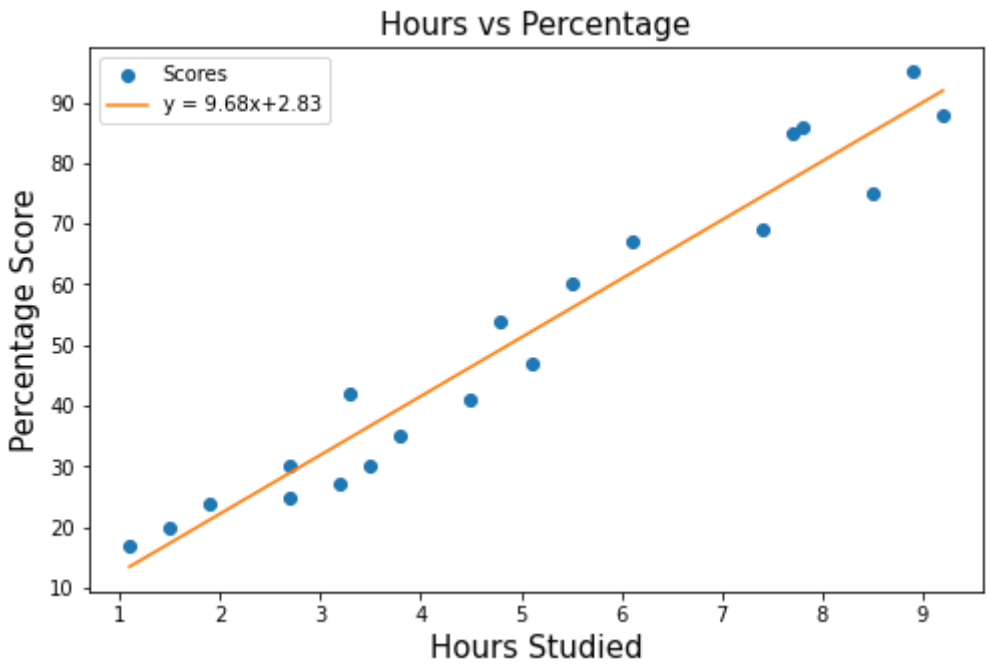
```
LinearRegression()
```

Посмотрим, что получилось

```
X_train.shape
```

```
(20, 1)
```

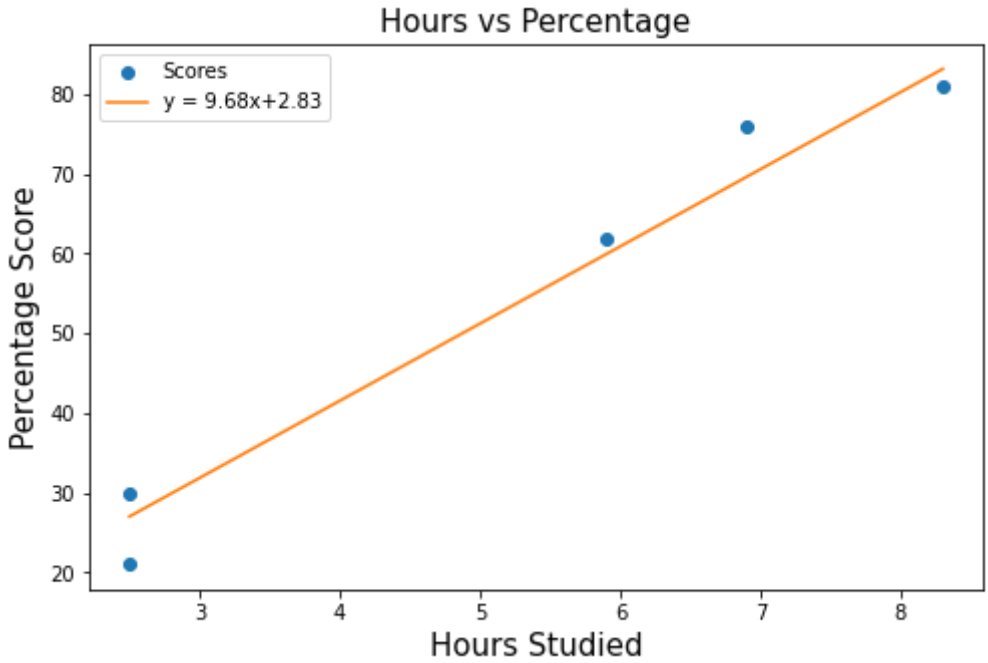
```
X_points = np.linspace(
    min(X_train), max(X_train), 100
)  # 100 dots at min to max
Y_pred = regressor.predict(X_points)

plt.figure(figsize=(8, 5))
plt.plot(X_train, Y_train, "o", label="Scores")
plt.plot(X_points, Y_pred, label="y = %.2fx+%.2f" % (regressor.coef_[0], regressor.inte
plt.title("Hours vs Percentage", size=15)
plt.xlabel("Hours Studied", size=15)
plt.ylabel("Percentage Score", size=15)
plt.legend()
plt.show()
```



Теперь сделаем предсказание для тестовой выборки

```
Y_pred = regressor.predict(X_test)

X_points = np.linspace(
    min(X_test), max(X_test), 100
)
Y_pred = regressor.predict(X_points)

plt.figure(figsize=(8, 5))
plt.plot(X_test, Y_test, "o", label="Scores")
plt.plot(X_points, Y_pred, label="y = %.2fx+%.2f" % (regressor.coef_[0], regressor.inte
plt.title("Hours vs Percentage", size=15)
plt.xlabel("Hours Studied", size=15)
```

```
plt.ylabel("Percentage Score", size=15)
plt.legend()
plt.show()
```



Выглядит не плохо

Посчитаем метрики для наших значений

```
from sklearn import metrics

Y_pred = regressor.predict(X_test)

print("Mean Absolute Error: %9.2f" % metrics.mean_absolute_error(Y_test, Y_pred))
print("Mean Squared Error: %10.2f" % metrics.mean_squared_error(Y_test, Y_pred))
print("Root Mean Squared Error: %5.2f" % np.sqrt(metrics.mean_squared_error(Y_test, Y_p
```

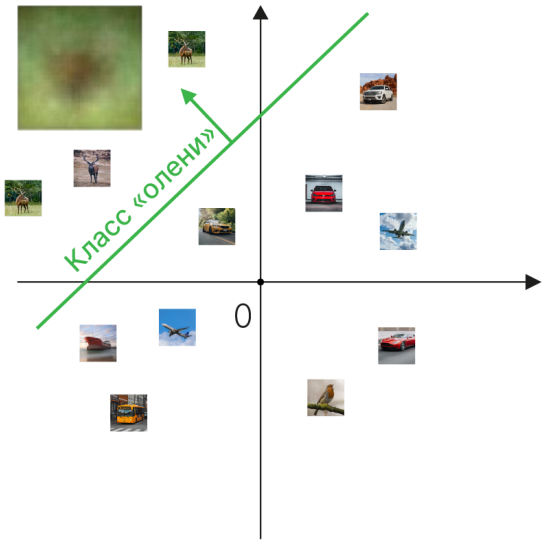
```
Mean Absolute Error:      3.92
Mean Squared Error:      18.94
Root Mean Squared Error:  4.35
```

Геометрическая интерпретация

Теперь, когда мы разобрались с тем, что такое регрессия и с чем ее едят, вернемся к нашим картинкам. Как можно применить регрессию для классификации?

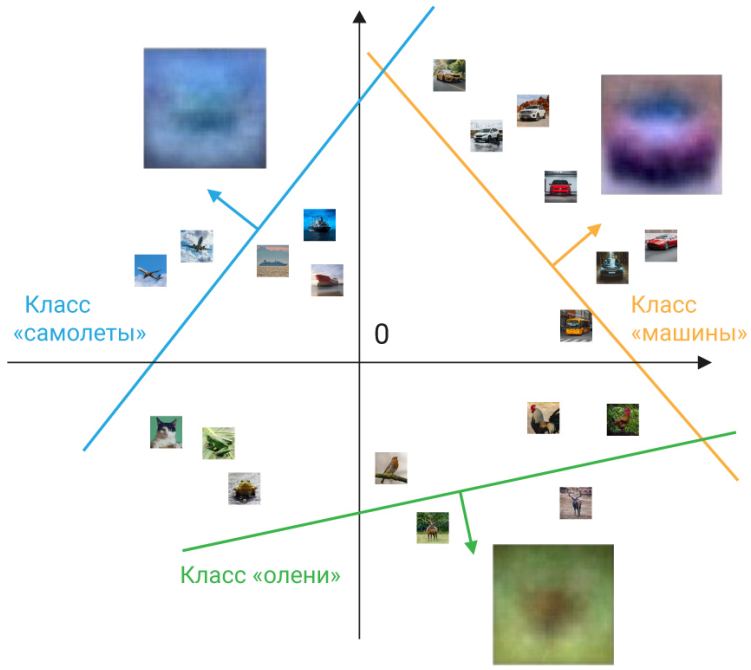
Предположим у нас есть только 2 класса. Как можно использовать регрессию для того, чтобы определить относится ли изображение к классу 0 или к классу 1? В упрощенном варианте, задача будет состоять в том, чтобы провести разделяющую плоскость (прямую) между 2-мя классами. Например, мы можем провести прямую через 0.

Рассмотрим другую ситуацию, в этом случае, мы не можем просто провести прямую через 0. Но можем отступить от 0 на какое-то расстояние и провести ее там. Вспомним, что уравнение прямой это $y = wx + b$, где b - это смещение (*bias*). Соответственно если b != 0, то прямая через 0 проходить не будет, а будет проходить через значение b.



[Linear Classification Loss Visualization]

Если у нас есть несколько классов (несколько шаблонов), мы можем для каждого из них посчитать уравнение $y_i = w_i x_i + b_i$.



На картинке нас интересуют 3 класса. Соответственно, мы можем записать систему линейных уравнений:

$$
\begin{aligned}
y_0 &= w_0 x_0 + b_0 \\
y_1 &= w_1 x_1 + b_1 \\
y_2 &= w_2 x_2 + b_2
\end{aligned}
$$

Добавление смещения

Мы их можем собрать в матрицу, тогда получится следующее:

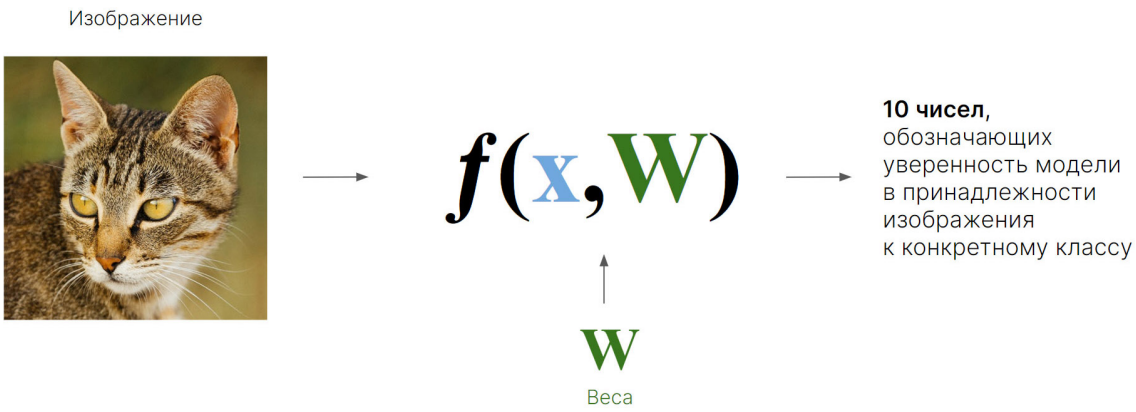
У нас есть матрица коэффициентов, которые мы каким-то образом подобрали, пока ещё не понятно как. Есть вектор x, соответствующий изображению.

Мы умножаем вектор на матрицу, получаем нашу гиперплоскость для четырехмерного пространства в данном случае. Чтобы оно не лежало в 0, мы должны добавить смещение. И мы можем сделать это после, но можно взять и этот вектор смещения (вектор **b**) просто приписать к матрице **W**.

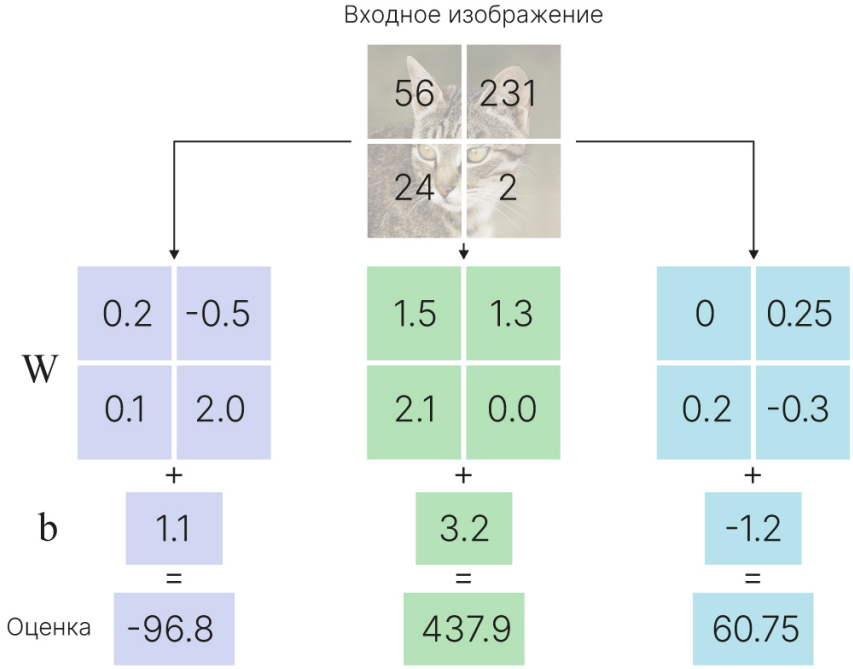
Что будет выходом такой конструкции? Мы умножили матрицу весов на наш вектор, соответствующий изображению, получили некоторый отклик. По этому отклику мы так же, как и при реализации метода ближайшего соседа можем судить: если он больше остальных, то мы предполагаем, что это кошка.

```
img = np.array([56, 231, 24, 2])
w_cat = np.array([0.2, -0.5, 0.1, 2.0])
print("Image ", img)
print("Weights ", w_cat)
print("img * w_cat ", img * w_cat)
print("sum ", (img * w_cat).sum())
print("Add bias ", (img * w_cat).sum() + 1.1)
```

```
Image  [ 56 231  24   2]
Weights  [ 0.2 -0.5  0.1  2. ]
img * w_cat  [  11.2 -115.5    2.4    4. ]
sum  -97.89999999999999
Add bias  -96.8
```



Собирая все вместе, получаем какое-то компактное представление, что у нас есть некоторая функция, на вход которой мы подаем изображение, и у нее есть параметры (веса). Пока происходит просто умножение вектора на матрицу, в дальнейшем это может быть что-то более сложное, функция будет представлять какую-то более сложную модель. А на выходе (для классификатора) мы получаем числа, которые интерпретируют уверенность модели в том, что изображение принадлежит к определенному классу.



Соответственно, эти коэффициенты, которые являются весами модели, надо каким-то образом подбирать. Но прежде чем подбирать коэффициенты, давайте определимся со следующим: как мы будем понимать, что модель работает хорошо или плохо? Ведь она возвращает достаточно абстрактные числа, которые нужно уметь интерпретировать.

Функция потерь SVM

Вернемся к Support Vector Machine. Мы разобрали, как в принципе работает метод опорных векторов (SVM) для задач классификации и регрессионного анализа.

Особым свойством метода опорных векторов является непрерывное уменьшение ошибки классификации и увеличение зазора.

Теперь разберемся, как SVM работает на практике. Вновь вернемся к нашему датасету CIFAR10. Мы же уже утверждали, что SVM работает с векторами (в частном случае с точками), а значит и с векторами размером (H,W,C) SVM работать тоже будет.



самолет	-3.45	-0.51	3.42
машина	-8.87	**6.04**	4.64
птица	0.09	5.31	2.65
кошка	**2.9**	-4.22	5.1
олень	4.48	-4.19	2.64
собака	8.02	3.58	5.55
лягушка	3.78	4.49	**-4.34**
лошадь	1.06	-4.37	-1.5
овца	-0.36	-2.09	-4.79
грузовик	-0.72	-2.93	6.14

По аналогии с тем, что мы уже делали, мы можем сравнивать отклик на ключевой класс (про который нам известно, что он на изображении, так как у нас есть метка этого класса) с остальными. Соответственно, мы подали изображение кошки и получили на выход вектор. Чем больше значение, тем больше вероятность того, что, по мнению модели, на изображении этот класс. Для кошки в данном случае это значение 2.9. Хорошо это или плохо? Нельзя сказать, пока мы не проанализировали остальную часть вектора. Если бы мы могли посмотреть на все значения в векторе, мы бы увидели, что есть значения больше, то есть в данном случае модель считает, что это собака, а не кошка, потому что для собаки значение максимально.

На основании этого можно построить некоторую оценку. Давайте смотреть на разницу правильного класса с неправильными. В зависимости от того, насколько уверенность в кошке будет больше остальных, настолько будет наша модель хорошей.

Но поскольку нам важна не работа модели на конкретном изображении, а важно оценить ее работу в целом, то эту операцию нужно проделать либо для всего датасета, либо для некоторой выборки, которую мы подаем на вход и подсчитываем средний показатель. Этот показатель (насколько хорошо работает модель), называется функцией потерь, или **loss функцией**. Называется она так, потому что она показывает не то, насколько хорошо работает модель, а то, насколько плохо.

Дальше будет понятно, почему так удобнее (разница только в знаке). Как это посчитать для всего датасета?

Мы каким-то образом считаем loss для конкретного изображения, потом усредняем по всем изображениям.

Дано: 3 учебных примера, 3 класса. При некотором W баллы f (a, W) = Wx равны:

кошка	**3.2**	1.3	2.2
машина	5.1	**4.9**	2.5
лягушка	-1.7	2.0	**-3.1**

Функция потерь показывает, насколько хорош наш текущий классификатор.

Дан датасет примеров:

$\{ (x_i, y_i) \}_{i=1}^{N}$

Где x_i изображение и y_i метка (число).

Потери по набору данных — это среднее значение потерь для примеров:

$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$

Снова возьмём наши обучающие примеры x_i и метки классов y_i. Представим прогнозы классификатора (оценки) в виде векторов: $s = f(x_i, W)$. Для каждого отдельного примера просуммируем значения оценок всех y, кроме истинной метки y_i. То есть, мы получим сумму значений всех неправильных категорий.

Теперь посчитаем разницу между ложными прогнозами и истинной оценкой: $s_j - s_y i$; $j \neq y_i$. Если истинная оценка больше, чем сумма всех неправильных прогнозов плюс некоторый дополнительный «зазор» (установим его равным 1) — значит, полученный балл для правильной категории намного превосходит любую ошибочную оценку. Это и будет наша функция потерь.

В символьном виде это выглядит так:

$$f(n) = \sum_{j \neq y_i} \begin{cases} 0, & \text{если } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1, & \text{если наоборот, то} \end{cases}$$

$= \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$

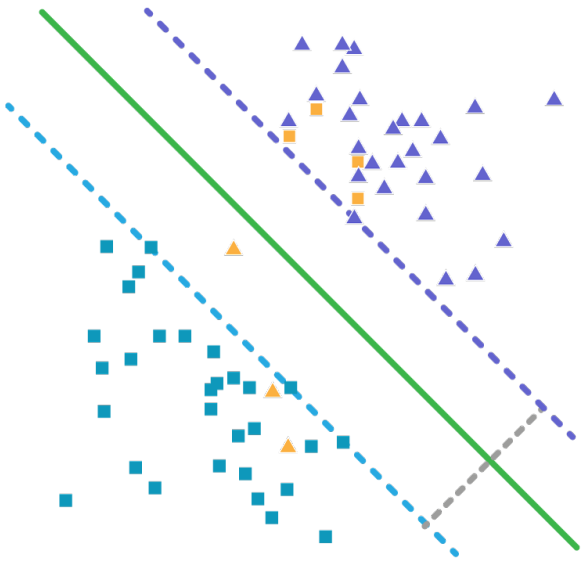
Вычисление функции потерь SVM

Не самая мощная, но достаточно интуитивно понятная loss функция – SVM loss.

Логика такая: если у нас уверенность модели в правильном классе большая, то модель работает хорошо и loss для данного конкретного примера должен быть равен нулю. Если есть класс, в котором модель уверена больше, чем в правильном, то loss должен быть не равен нулю, а отображать какую-то разницу, поскольку модель сильно ошиблась. При этом есть ещё одно соображение: что будет, если на выходе у правильного и ошибочного класса будут примерно равные веса? То есть, например, у кошки было бы 3.2, а у машины не 5.2, а 3.1. В этом случае ошибки нет, но понятно, что при небольшом изменении в данных (просто шум) скорее всего она появится.

То есть модель плохо отличает эти классы. Поэтому мы и вводили некоторый зазор, который должен быть между правильным и неправильным ответом.

Посмотрим на изображение снизу. У нас есть два класса: фиолетовые треугольники и синие квадраты, разделенные зазором. Также можем увидеть желтые треугольники и квадраты - это ошибочно распознанные классы.



И тоже учитывать его в loss функции: то есть сравнивать его результат для правильного класса не с чистым выходом для другого, а добавить к нему некоторую дельту (в данном случае – 1(единица)). Смотрим: если разница больше 0, то модель работает хорошо и loss равно нулю. Если нет, то мы возвращаем эту разницу, и loss будет складываться из этих индивидуальных разниц.

$$f(n) = \sum_{j \neq y_i} \begin{cases} 0, & \text{если } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1, & \text{если наоборот, то} \end{cases}$$

$$= \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

Ниже пример того, как считается loss.

Считаем функцию потерь для 1-ого изображения:

кошка	**3.2**
машина	5.1
лягушка	-1.7
Потери:	2.9

$$\boxed{L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)}$$

= max(0, 5.1 - 3.2 + 1)
+max(0, -1.7 - 3.2 + 1)
= max(0, 2.9) + max(0, -3.9)
= 2.9 + 0
= 2.9

Также считаем потери для 2-ого и 3-его изображения:

кошка	1.3
машина	**4.9**
лягушка	2.0
Потери:	0

= max(0, 1.3 - 4.9 + 1)
+max(0, 2.0 - 4.9 + 1)
= max(0, -2.6) + max(0, -1.9)
= 0 + 0
= 0

кошка	2.2
машина	2.5
лягушка	**-3.1**
Потери:	12.9

= max(0, 2.2 - (-3.1) + 1)
+max(0, 2.5 - (-3.1) + 1)
= max(0, 6.3) + max(0, 6.6)
= 6.3 + 6.6
= 12.9

Значения losses получились следующие:

Потери:	2.9	0	12.9

Считаем среднее значение loss для всего датасета:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i$$

$$L = \frac{2.9+0+12.9}{3} = 5.27$$

SVM loss

$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$

Как записать это в коде:

```
import os
from IPython.display import clear_output

file_exists = os.path.exists("/content/cifar-10-batches-py")
if file_exists == False:
    !wget https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
    !tar -xzf cifar-10-python.tar.gz

clear_output()
```

```
import pickle
import numpy as np

def unpickle(file):
    with open(file, "rb") as fo:
        dict = pickle.load(fo, encoding="bytes")
    return dict

X_train = np.zeros((0, 3072))
Y_train = np.array([])
for i in range(1, 6):
    raw = unpickle(f"/content/cifar-10-batches-py/data_batch_{i}")
    X_train = np.append(X_train, np.array(raw[b"data"]), axis=0)
    Y_train = np.append(Y_train, np.array(raw[b"labels"]), axis=0)

test = unpickle("/content/cifar-10-batches-py/test_batch")
X_test = np.array(test[b"data"])
Y_test = np.array(test[b"labels"])

labels_eng = [
    "Airplane",
    "Car",
    "Bird",
    "Cat",
    "Deer",
    "Dog",
    "Frog",
    "Horse",
    "Ship",
    "Truck",
]

print(X_train.shape)
print(X_test.shape)
```

```
(50000, 3072)
(10000, 3072)
```

```
W = np.random.randn(10, 3072) * 0.0001 # random weights
```

```
assa = np.maximum(0, W.dot(X_train[0]) - W.dot(X_train[0])[6] + 1)
assa[6] = 0
np.sum(assa)
```

```
6.237230308183291
```

```
def Li_svm(X, Y, W):
    Y = int(Y)  # to use like a index
    scores = W.dot(X)
    margins = np.maximum(0, scores - scores[Y] + 1)  # loss
    margins[Y] = 0
    return np.sum(margins)


L0 = Li_svm(X_train[0], Y_train[0], W)
print("Loss on first image", L0)
```

```
Loss on first image 6.237230308183291
```

Дано: 3 учебных примера, 3 класса.
При некотором W баллы f (a, W) = Wx равны:

кошка	**3.2**	1.3	2.2
машина	5.1	**4.9**	2.5
лягушка	-1.7	2.0	**-3.1**

Функция потерь показывает,
насколько хорош наш текущий классификатор

Дан набор примеров

$$\{(x_i, y_i)\}_{i=1}^{N}$$

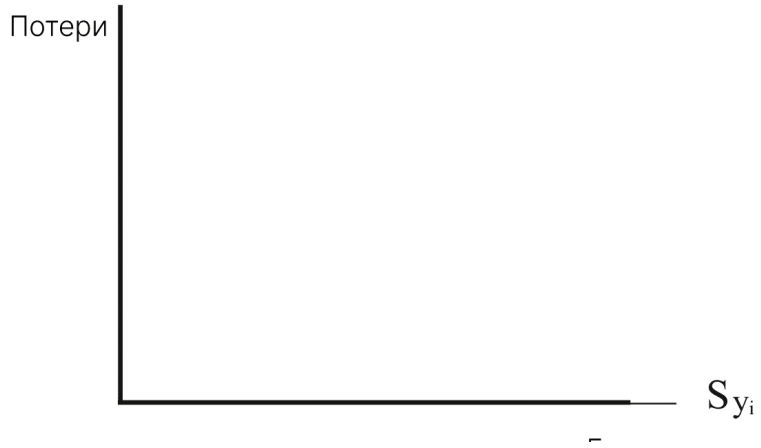
Где $\mathbf{x_i}$s изображение и $\mathbf{y_i}$s метка (число)

Потери по набору данных —
это среднее значение потерь для примеров:

$$L = \frac{1}{N}\sum_i L_i(f(x_i, W), y_i)$$

Собираем все вместе: считаем loss для всего датасета, усредняя ее.

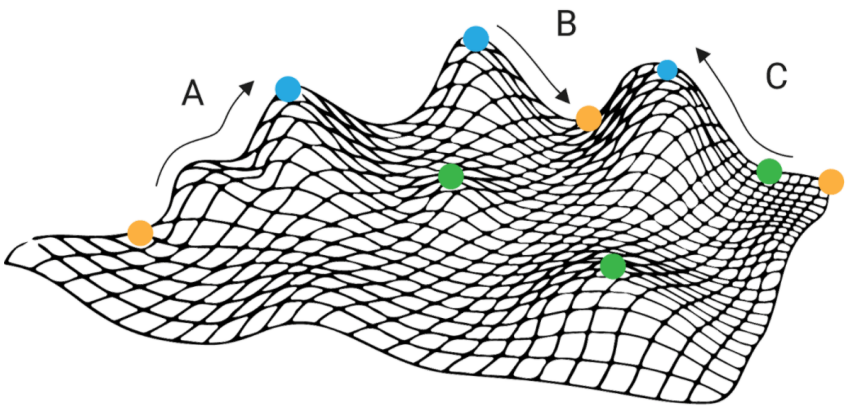
Как будет выглядеть график этой SVM-loss?



Обновления весов методом градиентного спуска

Функция потерь для ситуации, когда зазор большой, будет равна 0. Она будет меняться

Обучение / Optimisation



$$f(x, y) = z$$

$$\text{coord} = [x, y]$$

$$f(\overrightarrow{\text{coord}}) = z$$

$$f(w) = \text{Loss}$$

Идея в следующем: у нас есть некоторая поверхность, на которой будет точка минимума функции. Мы должны обновлять веса таким образом, чтобы смещаться в сторону этой точки. Loss показывает, как возрастает функция потерь. Так как потери - это плохо, мы должны их минимизировать, то есть мы должны уменьшать веса в сторону, обратную росту этой функции.

Если переходить на случай n-мерного (в данном случае трехмерного) пространства, здесь достаточно очевидна аналогия с поверхностью: у нас есть поверхность земли, которая описывается координатами x, y, z. Если мы движемся по этой поверхности, высота (z) будет зависеть от x, y.

x, y - это координаты. Мы можем записать их как вектор. Наша функция будет работать с вектором координат и выдавать скаляр, третью координату. Если в качестве координат мы будем использовать веса нашей модели, а в качестве z - loss, то аналогия станет полной. Наша задача сведется к тому, чтобы найти такой набор весов, при

котором значение функции будет минимально. То есть мы окажемся в каком-то минимуме, где ошибка будет минимально возможна для этих данных.

Для того чтобы предположить, где он может находиться, надо понимать, в какую сторону функция растет, а в какую - убывает. Для этого существует производная.

Поскольку у нас функция от нескольких переменных, если мы будем брать от нее производную, у нас получится вектор частных производных, то есть градиент этой функции, который будет показывать, как она меняется в каждом направлении.

Градиент функции потерь

Градиент — вектор, своим направлением указывающий направление наибольшего возрастания некоторой величины **φ**, значение которой меняется от одной точки пространства к другой (скалярного поля), а по величине (модулю) равный скорости роста этой величины в этом направлении.

Например, если взять в качестве **φ** высоту поверхности земли над уровнем моря, то её градиент в каждой точке поверхности будет показывать «направление самого крутого подъёма», и своей величиной характеризовать крутизну склона.

Другими словами, градиент — это производная по пространству, но в отличие от производной по одномерному времени, градиент является не скаляром, а векторной величиной.

Для случая трёхмерного пространства градиентом скалярной функции **φ=φ(x,y,z)** координат **(x,y,z)** называется векторная функция с компонентами

$$\frac{\partial \varphi}{\partial x}, \frac{\partial \varphi}{\partial y}, \frac{\partial \varphi}{\partial z}$$

.

Если **φ** — функция **n** переменных **X1...xn**, то её градиентом называется **n**-мерный вектор,

$$\frac{\partial \varphi}{\partial x_1}, \ldots, \frac{\partial \varphi}{\partial x_n}$$

компоненты которого, равны частным производным **φ** по всем её аргументам.

Размерность вектора градиента определяется размерностью пространства (или многообразия), на котором задано скалярное поле, о градиенте которого идёт речь. Оператором градиента называется оператор, действие которого на скалярную функцию (поле) даёт её градиент. Этого оператора иногда коротко называют просто «градиентом».

W - матрица(вектор) весов

L - функция потерь

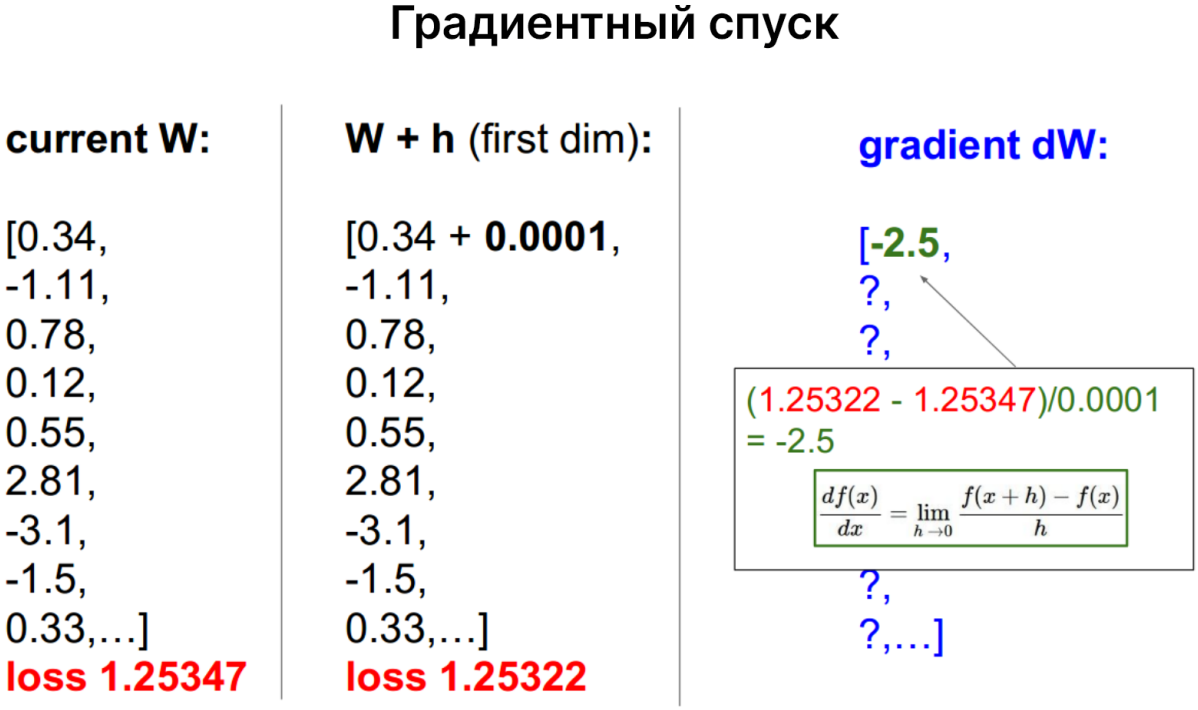
$\partial W = W_2 - W_1$

$\partial L = L_2 - L_1$

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial W_1} \\ \frac{\partial L}{\partial W_2} \\ \dots \\ \frac{\partial L}{\partial W_n} \end{bmatrix}$$

Наша задача будет сводиться к тому, что мы будем искать градиент loss функции по весам, которые будут состоять из производной по каждому направлению. Поскольку у нас здесь числа, можно считать этот градиент численно.

Численный расчет производной



Посчитаем градиент приближенно, воспользовавшись определением (в формуле аргумент обозначен как x, у нас же аргументом будет W): На нулевом шаге у нас есть W_0 найдем $L_0 = Loss(f(W_0, x))$ Прибавим к первому элементу W_0 небольшую величину h = 0.0001 и получим новую матрицу весов W_1 отличающуюся от W_0 на один единственный элемент.

Найдем Loss от $\frac{W_1}{L_1} = Loss(f(W_1, x))$ По определению производной $\frac{dL}{W_0} = \frac{(L_1 - L_0)}{h}$

Повторяя этот процесс для каждого элемента из W, найдем вектор частных производных, то есть градиент $\frac{dL}{dW}$.

Плюсы: Это просто.

Проблемы:

1. Это очень долго, нам придется заново искать значение loss функции для каждого W_i.
2. Это неточно, так как по определению приращение h бесконечно мало, а мы используем конкретное пусть и небольшое число. И если мы сделаем его слишком маленьким, то столкнемся с ошибками связанными с округлением в памяти компьютера.

Поэтому данный метод может использоваться как проверочный. А на практике вместо него используется **аналитический расчет градиента**.

Аналитический расчет производной от функции потерь SVM

Простые производные

$$x' = \frac{\delta x}{\delta x} = 1$$
$$(x^2)' = \frac{\delta x^2}{\delta x} = 2x$$
$$(\log x)' = \frac{\delta \log x}{\delta x} = \frac{1}{x}$$
$$(e^x)' = \frac{\delta e^x}{\delta x} = e^x$$
$$\frac{\delta c f(x)}{\delta x} = c \cdot \frac{\delta f(x)}{\delta x}$$
$$\frac{\delta f(x) + c}{\delta x} = \frac{\delta f(x)}{\delta x}$$

c - константа, не зависящая от x

$$\frac{\delta[f(x) + g(x)]}{\delta x} = \frac{\delta f(x)}{\delta x} + \frac{\delta g(x)}{\delta x}$$
$$\frac{\delta(x^2 + y^3)}{\delta x} = 2x$$

так как y по отношению к x - константа и мы меняем только x

$$\frac{\delta(x^2+y^3)}{\delta y} = 3y^2$$

так как x по отношению к y - константа и мы меняем только y

$$(e^y)' = \frac{\delta e^y}{\delta} = 0$$

Chain-rule

Производная функции $f(g)$:

$$\frac{\delta f}{\delta g}$$

Пусть g на самом деле не просто переменная, а зависит от h. Тогда производная от f по g **не меняется**, а производная f по h запишется следующим образом:

$$\frac{\delta f(g(h))}{\delta h} = \frac{\delta f}{\delta g}\frac{\delta g}{\delta h}$$

Пусть теперь h зависит от x. Все аналогично

$$\frac{\delta f(g(h(x)))}{\delta x} = \frac{\delta f}{\delta g}\frac{\delta g}{\delta h}\frac{\delta h}{\delta x}$$

Так можно делать до бесконечности, находя производную сколь угодно сложной функции. И, что важно - мы можем считать градиенты частями - посчитать сначала f по g, потом g по h....

$$\frac{\delta log(x^2+5)}{\delta x}$$

$$h = x^2+5$$

$$\frac{\delta log(x^2+5)}{\delta x} = \frac{\delta log(h)}{\delta h}\frac{\delta h}{\delta x}$$

$$\frac{\delta log(h)}{\delta h} = \frac{1}{h}$$

$$\frac{\delta h}{\delta x} = 2x$$

$$\frac{\delta log(x^2+5)}{\delta x} = \frac{1}{x^2+5}\cdot 2x = \frac{2x}{x^2+5}$$

Часть MSE-loss

$$loss = (y-\hat{y})^2$$

$$\hat{y} = wx+b$$

$$\frac{\delta loss}{\delta w} = \frac{\delta loss}{\delta \hat{y}}\cdot\frac{\delta \hat{y}}{\delta w}$$

$$\frac{\delta loss}{\delta \hat{y}} = \frac{\delta (y-\hat{y})^2}{\delta (y-\hat{y})} \frac{\delta y - \hat{y}}{\delta \hat{y}} = 2(y-\hat{y}) \cdot -1 = 2(\hat{y}-y)$$

$$\frac{\delta \hat{y}}{\delta w} = \frac{\delta wx + b}{\delta w} = x$$

$$\frac{\delta loss}{\delta w} = 2x \cdot (\hat{y} - y)$$

MSE-loss

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

y_i - константы $\hat{y_i}$ - не являются функциями друг от друга

$$\hat{y} = wx_i + b$$

$$\frac{\delta MSE}{\delta w} = \frac{1}{N} \sum \frac{\delta (y_i - \hat{y_i})^2}{\delta \hat{y_i}} \frac{\delta \hat{y_i}}{\delta w}$$

Часть MAE-Loss

$$loss = |y - \hat{y}|$$

$$\hat{y} = wx + b$$

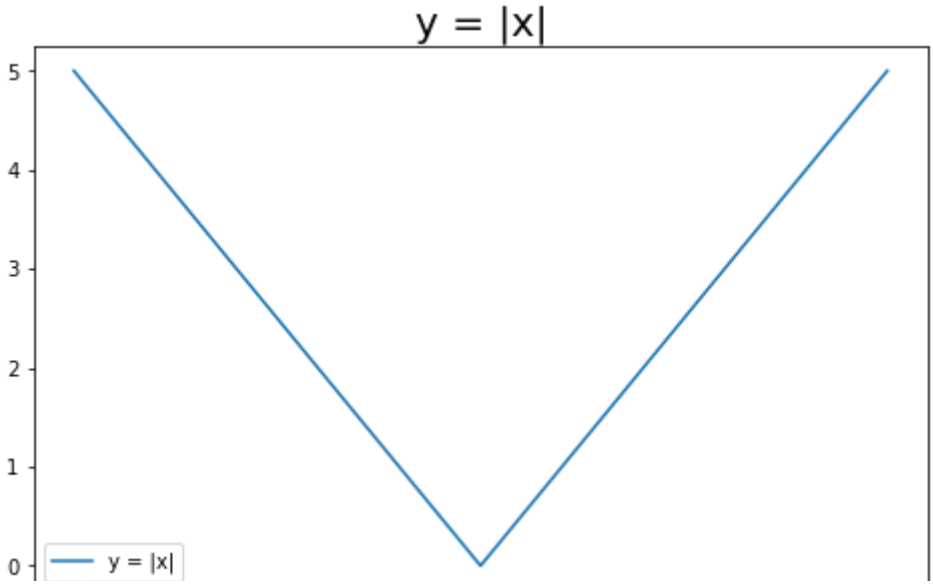
$$\frac{\delta loss}{\delta w} = \frac{\delta loss}{\delta \hat{y}} \cdot \frac{\delta \hat{y}}{\delta w}$$

$$\frac{\delta \hat{y}}{\delta w} = \frac{\delta wx + b}{\delta w} = x$$

$$\frac{\delta loss}{\delta \hat{y}} = \frac{\delta |y - \hat{y}|)}{\delta (y - \hat{y})} \frac{\delta y - \hat{y}}{\delta \hat{y}} = \frac{\delta |y - \hat{y}|)}{\delta (y - \hat{y})} \cdot -1 = -\frac{\delta |y - \hat{y}|)}{\delta (y - \hat{y})}$$

Строго говоря, у модуля не существует производной в 0.

```
import matplotlib.pyplot as plt
import seaborn as sns

X = [i for i in range(-5, 6)]
Y = [abs(i) for i in range(-5, 6)]

plt.figure(figsize=(8, 5))
plt.plot(X, Y, label="y = |x|")
plt.title("y = |x|", size=20)
plt.legend()
plt.show()
```



Но мы можем сказать, что в этой точке производная равна 0. Если аргумент модуля меньше 0, то производная будет -1.

Если больше +1

$$\frac{\delta loss}{\delta \hat{y}} = \frac{\delta|y-\hat{y}|}{\delta(y-\hat{y})}\frac{\delta y-\hat{y}}{\delta \hat{y}} = \frac{\delta|y-\hat{y}|}{\delta(y-\hat{y})}\cdot -1 = -\frac{\delta|y-\hat{y}|}{\delta(y-\hat{y})} = -sign(y-\hat{y}) = sign(\hat{y}$$

```
X = [i for i in range(-5, 1, 1)]
Y = [i * 0 - 1 for i in range(6)]
X_1 = [i for i in range(0, 6)]
Y_1 = [i * 0 + 1 for i in range(0, 6)]

plt.figure(figsize=(8, 5))
plt.plot(X, Y, "b")
plt.plot(X_1, Y_1, "b")
plt.plot(0, 0, "ro")
plt.plot(0, 1, "bo")
plt.plot(0, -1, "bo")
plt.show()
```

Max-Loss

$$b = max(x, y)$$

$$b = x \;\; if \;\; x > y \;\; else \;\; y$$

$$\frac{\delta b}{\delta x} = \frac{\delta x}{\delta x} \;\; if \;\; x > y \;\; else \;\; \frac{\delta y}{\delta x} = 1 \;\; if \;\; x > y \;\; else \;\; 0$$

Если $x > y$, то он оказал влияние на b. Иначе, его вклада в b НЕ БЫЛО - градиент равен 0

SVM-Loss

Из:

$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$

Получаем:

$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$

*Не обязательное задание:

Поставлю дополнительно 20 баллов тому, кто графически оформит результаты этого эксперимента и напишет вывод о зависимости качества обучения от learning_rate и batch_size (возможно, нужно будет добавить варианты их сочетаний и/или увеличить количество эпох до 20).

```
import os
from IPython.display import clear_output

file_exists = os.path.exists("/content/cifar-10-batches-py")
if file_exists == False:
    !wget https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
    !tar -xzf cifar-10-python.tar.gz
clear_output()
```

```
import numpy as np
import pickle

def unpickle(file):
    with open(file, "rb") as fo:
        dict = pickle.load(fo, encoding="bytes")
    return dict

```

```

X_train = np.zeros((0, 3072))
Y_train = np.array([])
for i in range(1, 6):
    raw = unpickle(f"/content/cifar-10-batches-py/data_batch_{i}")
    X_train = np.append(X_train, np.array(raw[b"data"]), axis=0)
    Y_train = np.append(Y_train, np.array(raw[b"labels"]), axis=0)

test = unpickle("/content/cifar-10-batches-py/test_batch")
X_test = np.array(test[b"data"])
Y_test = np.array(test[b"labels"])

labels_eng = [
    "Airplane",
    "Car",
    "Bird",
    "Cat",
    "Deer",
    "Dog",
    "Frog",
    "Horse",
    "Ship",
    "Truck",
]

print(f"X_train shape: {X_train.shape}, Y_train shape: {Y_train.shape}")
print(f"X_train shape: {X_test.shape}, Y_train shape: {Y_test.shape}")
```

```
X_train shape: (50000, 3072), Y_train shape: (50000,)
X_train shape: (10000, 3072), Y_train shape: (10000,)
```

```
import random

class LinearClassifier():
    def __init__(self, labels, batch_size, random_state=42):
        self.labels = labels  # classes names
        self.classes_num = len(labels)  # num of classes

        np.random.seed(
            random_state
        )
        self.W = (
            np.random.randn(3073, self.classes_num) * 0.0001
        )  # generate random weights, reshape to add bias
        self.batch_size = batch_size  # batch_size

    def fit(self, X_train, Y_train, learning_rate=1e-8):
        loss = 0.0  # обнуляем loss
        train_len = X_train.shape[0]  # num of examples
        indexes = list(range(train_len))  # indexes train_len
        random.shuffle(indexes)

        for i in range(
            0, train_len, self.batch_size
```

```
        ):
            idx = indexes[
                i : i + self.batch_size
            ]  #
            X_batch = X_train[idx]
            Y_batch = Y_train[idx]

            X_batch = np.hstack(
                [X_batch, np.ones((X_batch.shape[0], 1))]
            )  # add bias

            loss_val, grad = self.loss(X_batch, Y_batch)  # loss and gradient
            self.W -= learning_rate * grad  # update weigths

            loss += loss_val  # loss sum
        return loss / (train_len)  # mean loss

    def loss(self, X, Y):
        current_batch_size = X.shape[0]  # batch_size
        loss = 0.0
        dW = np.zeros(self.W.shape)
        for i in range(current_batch_size):
            scores = X[i].dot(
                self.W
            )  # vector of shape 10
            correct_class_score = scores[
                int(Y[i])
            ]
            above_zero_loss_count = 0
            for j in range(self.classes_num):
                if j == Y[i]:  # predict class
                    continue
                margin = scores[j] - correct_class_score + 1  # loss
                if margin > 0:
                    above_zero_loss_count += (
                        1
                    )
                    loss += margin  #
                    dW[:, j] += X[i]  #
            dW[:, int(Y[i])] -= above_zero_loss_count * X[i]
        loss /= current_batch_size
        dW /= current_batch_size
        return loss, dW

    def forward(self, X):
        X = np.append(X, 1)  # add 1 (bias)
        scores = X.dot(self.W)
        return np.argmax(scores)
```

```
import time

def validate(model, X_test, Y_test, noprint=False):
    correct = 0
```

```
    for i, img in enumerate(X_test):
        index = model.forward(img)
        correct += (
            1 if index == Y_test[i] else 0
        )
        if noprint is False:
            if i > 0 and i % 1000 == 0:
                print(
                    "Accuracy {:.3f}".format(correct / i)
                )
    return correct / len(Y_test)
```

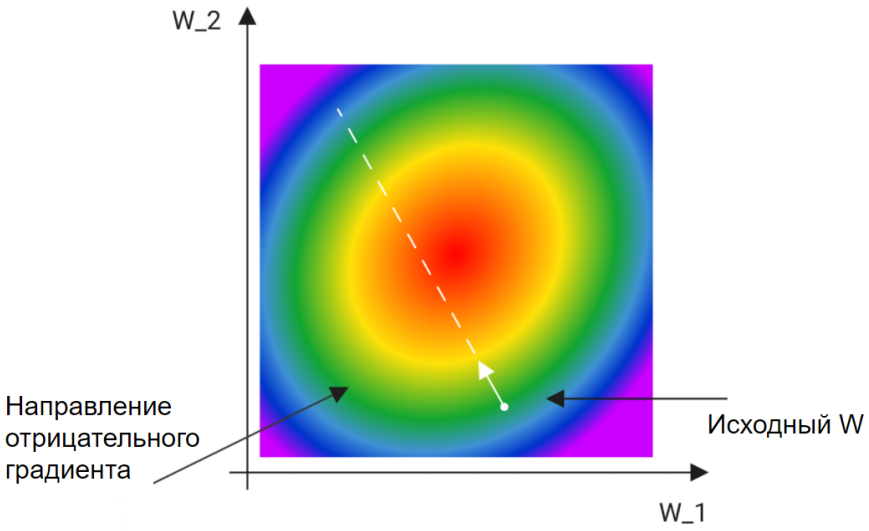
```
print("How learning quality depends of speed:")

for lr in [1e-2, 1e-8]:
    for bs in [256, 2048]:

        print("-" * 50, "\n", "learning_rate =", lr, "\tbatch_size =", bs)
        print()
        lc_model = LinearClassifier(labels_eng, batch_size=bs)

        best_accuracy = 0
        for epoch in range(10):
            loss = lc_model.fit(X_train, Y_train, learning_rate=lr)
            accuracy = validate(lc_model, X_test, Y_test, noprint=True)
            if best_accuracy < accuracy:
                best_accuracy = accuracy
                best_epoch = epoch
            print(f"Epoch {epoch} \tLoss: {loss}, \tAccuracy:{accuracy}")

        print()
        print(f"Best accuracy is {best_accuracy} in {best_epoch} epoch")
```

```
Epoch 0         Loss: 1769.278891925488,        Accuracy:0.161
Epoch 1         Loss: 1371.802667697692,        Accuracy:0.1516
Epoch 2         Loss: 1283.9206069841541,       Accuracy:0.1662
Epoch 3         Loss: 1272.9115407583477,       Accuracy:0.1903
Epoch 4         Loss: 1225.7094989091397,       Accuracy:0.2045
Epoch 5         Loss: 1185.3451529682518,       Accuracy:0.2161
Epoch 6         Loss: 1226.1353027996404,       Accuracy:0.174
Epoch 7         Loss: 1141.603897510695,        Accuracy:0.2759
Epoch 8         Loss: 1153.4723791935476,       Accuracy:0.2206
Epoch 9         Loss: 1119.0350664290538,       Accuracy:0.1081

Best accuracy is 0.2759 in 7 epoch
--------------------------------------------------
 learning_rate = 0.01   batch_size = 2048

Epoch 0         Loss: 285.92672586543046,       Accuracy:0.1629
Epoch 1         Loss: 248.71600755496857,       Accuracy:0.1476
Epoch 2         Loss: 225.50737794004425,       Accuracy:0.2141
Epoch 3         Loss: 213.57248355883405,       Accuracy:0.1906
Epoch 4         Loss: 203.1264588821868,        Accuracy:0.2001
Epoch 5         Loss: 199.156774600413,         Accuracy:0.2559
Epoch 6         Loss: 193.62459922757046,       Accuracy:0.2261
Epoch 7         Loss: 189.49298053362978,       Accuracy:0.2405
Epoch 8         Loss: 189.08562308862428,       Accuracy:0.2147
```

```
Epoch 8         Loss: 189.08562308863438,       Accuracy:0.2147
Epoch 9         Loss: 178.14928425876877,       Accuracy:0.26

Best accuracy is 0.26 in 9 epoch
--------------------------------------------------
 learning_rate = 1e-08  batch_size = 256

Epoch 0         Loss: 0.027139928065061072,     Accuracy:0.2627
Epoch 1         Loss: 0.02243592460493113,      Accuracy:0.2991
Epoch 2         Loss: 0.021140834544993273,     Accuracy:0.3096
Epoch 3         Loss: 0.020438755291006572,     Accuracy:0.3231
Epoch 4         Loss: 0.01997069531355622,      Accuracy:0.3346
Epoch 5         Loss: 0.01964713109184568,      Accuracy:0.3317
Epoch 6         Loss: 0.019364433976962,        Accuracy:0.3419
Epoch 7         Loss: 0.019150845119829868,     Accuracy:0.3408
Epoch 8         Loss: 0.018986691077263956,     Accuracy:0.3451
Epoch 9         Loss: 0.01884490407546561,      Accuracy:0.3455

Best accuracy is 0.3455 in 9 epoch
--------------------------------------------------
 learning_rate = 1e-08  batch_size = 2048

Epoch 0         Loss: 0.0043177205726905605,    Accuracy:0.1417
Epoch 1         Loss: 0.0038343878524549025,    Accuracy:0.1865
Epoch 2         Loss: 0.003539055613940809,     Accuracy:0.2068
Epoch 3         Loss: 0.0033475859667490794,    Accuracy:0.2277
Epoch 4         Loss: 0.003220308610409241,     Accuracy:0.2451
Epoch 5         Loss: 0.0031275561442641276,    Accuracy:0.2589
Epoch 6         Loss: 0.003059681999142495,     Accuracy:0.2647
Epoch 7         Loss: 0.0030022968195279644,    Accuracy:0.2719
Epoch 8         Loss: 0.0029524362485628085,    Accuracy:0.2813
Epoch 9         Loss: 0.002911637183729191,     Accuracy:0.2872

Best accuracy is 0.2872 in 9 epoch
```

Выбор шага обучения

Изменение весов

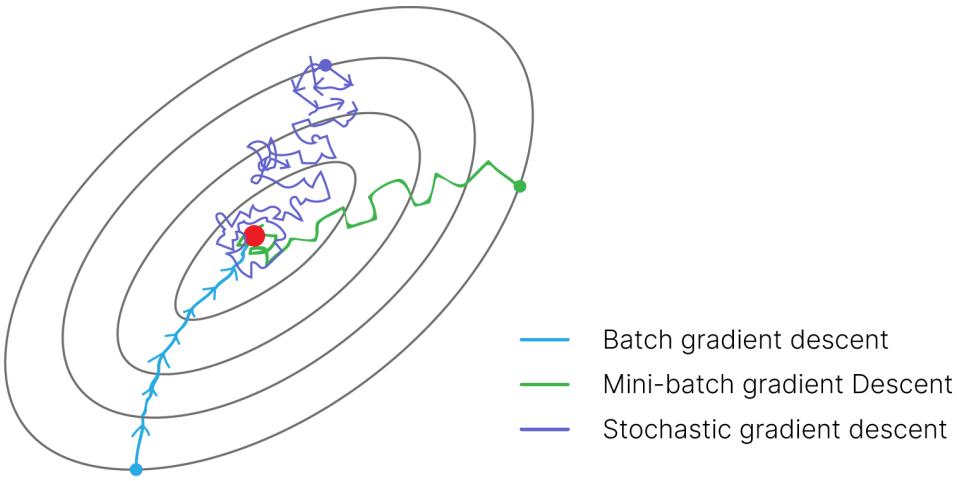


Шаг обучения - некоторый коэффициент, как правило, небольшой, который не позволяет нам двигаться слишком быстро. У нас есть точка, в которую мы хотим попасть. Если мы сделаем слишком большой шаг, то мы ее перескочим (график справа), поэтому надо подобрать шаг, который не позволит ее перескочить, но в то же время, чтобы тот же процесс не шел слишком медленно (как на графике слева)

Пока изменения loss функции достаточно большие, сам по себе градиент тоже большой. За счет этого можно двигаться быстро. Когда он будет уменьшаться, мы должны оказаться рядом с этим минимумом, и шаг, который мы выбрали, не должен мешать этому процессу.

Бывают ситуации, когда шаг можно менять в самом процессе обучения, когда в начале обучения модели шаг большой, потом, по мере того, как она сходится, чтобы более четко найти минимум, шаг можно изменить вручную. Но изначально его нужно каким-то образом подобрать, и это зависит от данных и от самой модели. Это тоже гиперпараметр, связанный с обучением.

Выбор размера батча



Мы с самого начала говорили о выборках, некоторого количества примеров. В дальнейшем, мы будем называть их батчами. Батч (англ. *batch*) - это некоторое подмножество обучающей выборки фиксированного размера.

При этом было не очень понятно, чем они мотивированны. Точнее, мы мотивировали это тем, что у нас много данных, и мы не сможем их обработать все, и это правда. Даже если мы сможем загрузить все данные в память, нам нужно будет загрузить их и использовать при расчете, в том числе градиента. Это ещё более затратно.

Зачем батчи нужны при рассчете loss? Если мы посчитаем loss по одному изображению, скорее всего он будет очень специфичен, и это движение, которое произойдет, будет направлено в сторону минимума, потому что по этому конкретному изображению мы улучшим показатели,что не отражает обобщения всех данных, поэтому мы используем батчи.

Нас в первую очередь интересует более общее направление, которое будет работать на большинстве наших данных. Поэтому если мы будем оптимизировать модели, исходя из одного элемента данных, путь будет витиеватый, и процесс будет происходить достаточно долго. Если же данные не помещаются в память, то **можно использовать батч того размера, который у нас есть**. Можно загрузить в память и считать по батчам градиент. В этом случае спуск будет более плавным, чем по одному изображению.

Также при использовании всего датасета тоже есть свои минусы. **Не всегда загрузка всего датасета приводит к увеличению точности**.

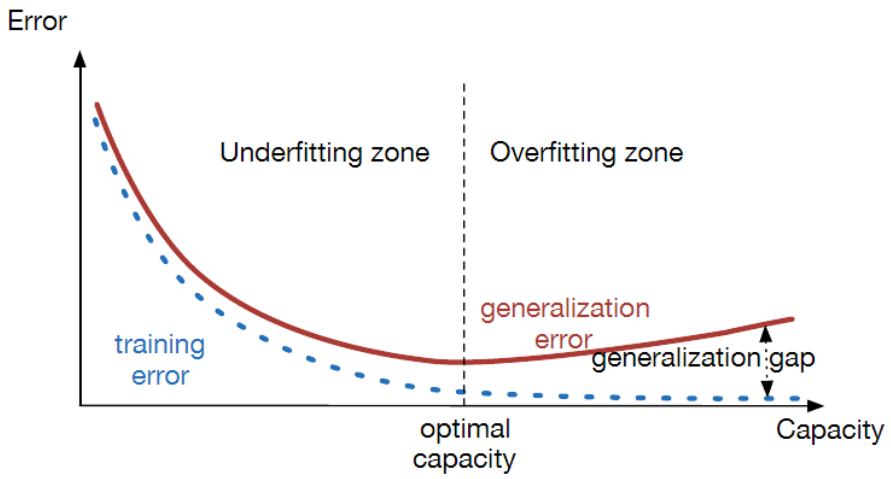
Подробнее:

- [Методы оптимизации нейронных сетей]
- [Обучение нейронной сети]

Регуляризация

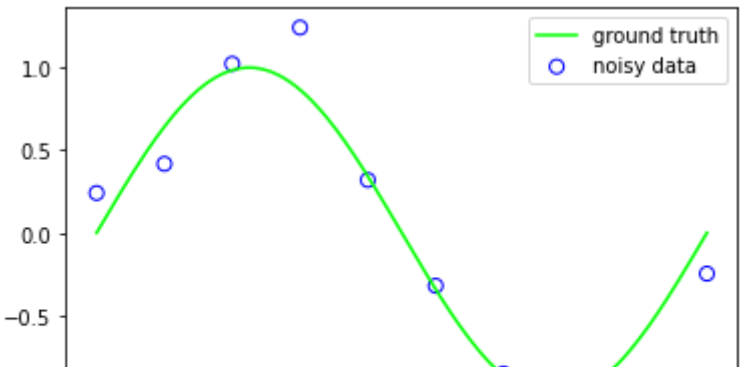
Сложность модели (*model complexity*) — важный гиперпараметр. В частности, для линейной модели, сложность может быть представлена количеством параметров, для полиномиальных моделей — степенью полинома, для деревьев решений — глубиной дерева и т.д.

Сложность модели тесно связана с **ошибкой обобщения** (*generalization error*). Ошибка обобщения отличается от ошибки обучения, измеряемой на тренировочных данных, тем, что позволяет оценить обобщающую способность модели, приобретенную в процессе обучения, давать точные ответы на неизвестных ей объектах. Слишком простой модели не будет хватать мощности для обобщения сложной закономерности в данных, что приводит к большой ошибке обобщения, с другой стороны слишком сложная модель также приводит к большой ошибке обобщения за счет того, что в силу своей сложности модель начинает пытаться искать закономерности в шуме, добиваясь большей точности на тренировочных данных, теряя при этом часть обобщающей способности.



Проиллюстрируем описанное явление на примере полиномиальной модели.

```
x = np.linspace(0, 2*np.pi, 10)
y = np.sin(x) + np.random.normal(scale=0.25, size=len(x))
plt.scatter(x, y, s=50, facecolors='none', edgecolors='b', label='noisy data')

x_true = np.linspace(0, 2*np.pi, 200)
y_true = np.sin(x_true)
plt.plot(x_true, y_true, c='lime', label='ground truth')
plt.legend()
plt.show()
```

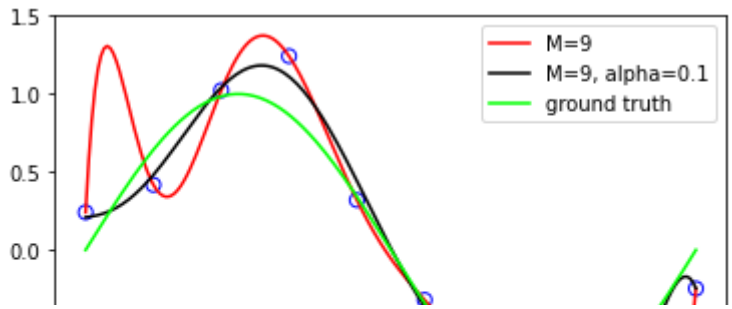


Попробуем аппроксимировать имеющуюся зависимость с помощью полиномиальной модели, используя шумные данные в качестве тренировочных данных:

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline

x_train = x.reshape(-1,1)

fig = plt.figure(figsize=(12,6))

for i, degree in enumerate([0,1,3,9]):

    model = make_pipeline(PolynomialFeatures(degree), LinearRegression())

    model.fit(x_train, y)
    y_plot = model.predict(x_true.reshape(-1,1))

    fig.add_subplot(2,2,i+1)
    plt.plot(x_true, y_plot, c='red', label=f'M={degree}')
    plt.scatter(x, y, s=50, facecolors='none', edgecolors='b')
    plt.plot(x_true, y_true, c='lime')
    plt.legend()
plt.show()
```

Видно, что модель может переобучаться, подстраиваясь под тренировочную выборку. В полиноме степень, и как следствие количество весов — это гиперпараметр, который можно подбирать на кросс-валидации, однако, когда мы таким образом подбираем сложность модели мы налагаем довольно грубое ограничение на обобщающую способность модели в целом. Вместо этого более разумным было бы оставить модель сложной, но использвать некий ограничитель (**регуляризатор**), который будет заставлять модель отдавать предпочтение выбору более простого обобщения.

```
from sklearn.linear_model import Ridge

model = make_pipeline(PolynomialFeatures(9), LinearRegression())
model_ridge = make_pipeline(PolynomialFeatures(9), Ridge(alpha=0.1))

model.fit(x_train, y)
y_plot = model.predict(x_true.reshape(-1,1))

model_ridge.fit(x_train, y)
y_plot_ridge = model_ridge.predict(x_true.reshape(-1,1))

plt.plot(x_true, y_plot, c='red', label=f'M={degree}')
plt.plot(x_true, y_plot_ridge, c='black', label=f'M={degree}, alpha=0.1')
plt.scatter(x, y, s=50, facecolors='none', edgecolors='b')
plt.plot(x_true, y_true, c='lime', label='ground truth')
plt.legend()
plt.show()

poly_coef = model[1].coef_

eq = f'y = {round(poly_coef[0], 2)}+{round(poly_coef[1], 2)}*x'
for i in range(2, 10):
    eq += f'+{round(poly_coef[i], 2)}*x^{i}'

print('Without regularization: ', eq)

poly_coef = model_ridge[1].coef_

eq = f'y = {round(poly_coef[0], 2)}+{round(poly_coef[1], 2)}*x'
for i in range(2, 10):
    eq += f'+{round(poly_coef[i], 2)}*x^{i}'

print('With regularization: ', eq)
```



Видно, что одним из "симптомов" переобучения являются аномально большие веса. Модель Ridge Regression, показанная в примере выше, использует L2 регуляризуцию для борьбы с этим явлением.

```
Without regularization:  y = 0.0+11.84*x+-43.61*x^2+63.07*x^3+-46.32*x^4+19.4*x^5+-4
```

$$x = [1, 1, 1, 1]$$

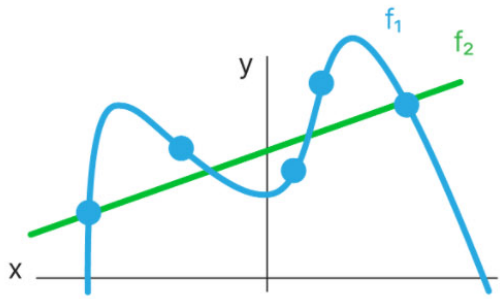
$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$



L2 Regularization = weights decay

Идея состоит в том, что мы можем наложить некоторое требование на сами веса. Дело в том, что можно получить один и тот же выход модели при разных весах (выход модели соответствует умножению весов на x), при разных w выход может быть идентичен.

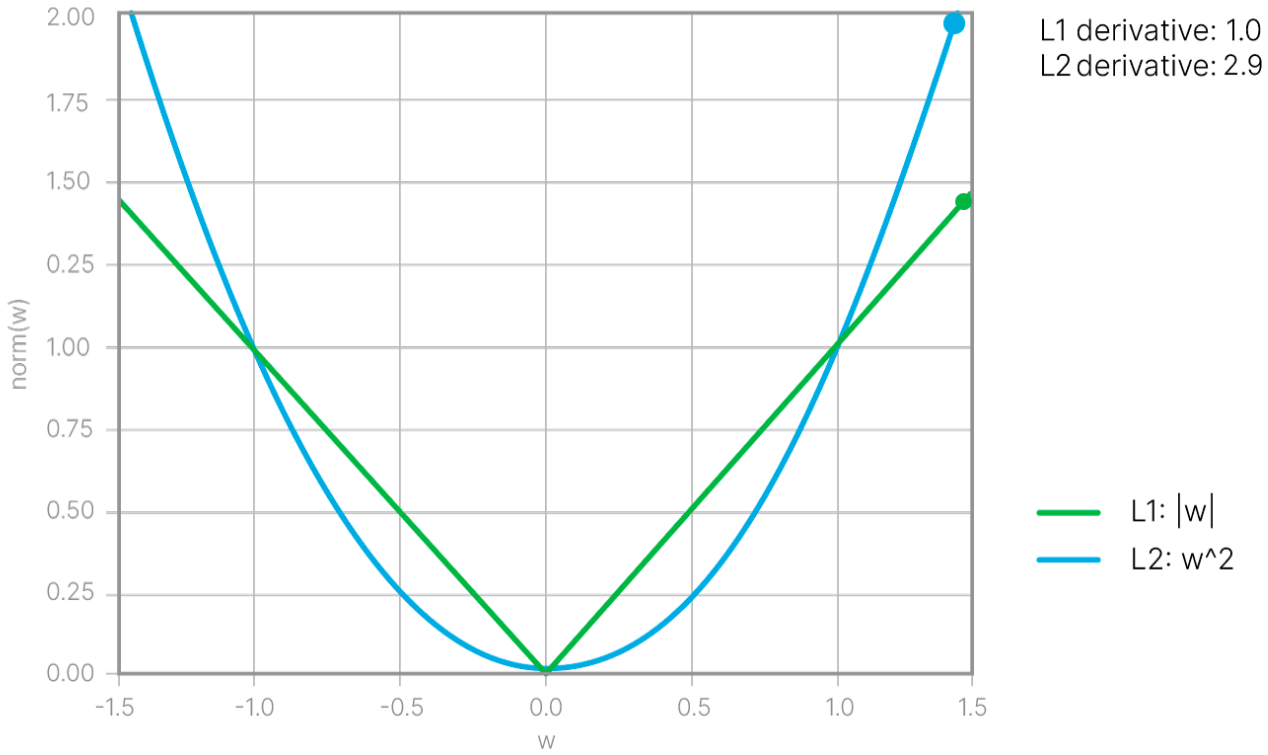
Эти параметры задают некоторую аппроксимацию нашей целевой функции, аппроксимировать функцию можно двумя способами:

1. Использовать все имеющиеся данные и провести ее строго через все точки, которые нам известны;
2. Использовать более простую функцию: (в данном случае линейную), которая не попадет точно во все данные, но зато будет соответствовать некоторым общим закономерностям, которые у них есть.

Характерной чертой переобучения является второй сценарий, и сопровождается он, как правило, большими весами. Введение L2-регуляризации приводит к тому, что большие веса штрафуются и предпочтение отдается решениям, использующим малые значения весов.

Модель может попробовать схитрить и по-другому - использовать все веса, все признаки, даже незначимые, но с маленькими коэффициентами. С этим L2-loss поможет хуже, так как он не сильнее штрафует мелкие веса. Результатов его применения - малые значения весов, которые использует модель

В этом случае на помощь приходит L1-loss, который штрафует вес за сам факт отличия его от нуля. Но и штрафует он все веса одинаково. Результат его применения - малое число весов, которые использует модель в принципе.



Это лоссы можно комбинировать - получится Elastic Net

$\lambda =$ regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\textbf{Data loss}} + \underbrace{\lambda R(W)}_{\textbf{Regularization}}$$

Берем сумму всех весов по всей матрице w, и добавляем ее к loss. Соответственно, чем больше будет эта сумма, тем больше будет суммарный loss.

В дальнейшем проблема с переобучением будет вставать довольно часто. Методов регуляризации модели существует достаточно много. Этот — один из базовых, который будет использоваться практически во всех оптимизаторах, с которыми познакомимся позже.

Функция потерь Кросс-энтропия

[Отличное видео от Statquest про энтропию]

На Cross-entropy Loss построены практически все модели, про которые будет идти речь.

Это оценка результата и обновление весов с использованием градиента. У нас один слой, поэтому мы достаточно легко посчитали от него градиент. Если слоев будет много, сделать это будет сложней.

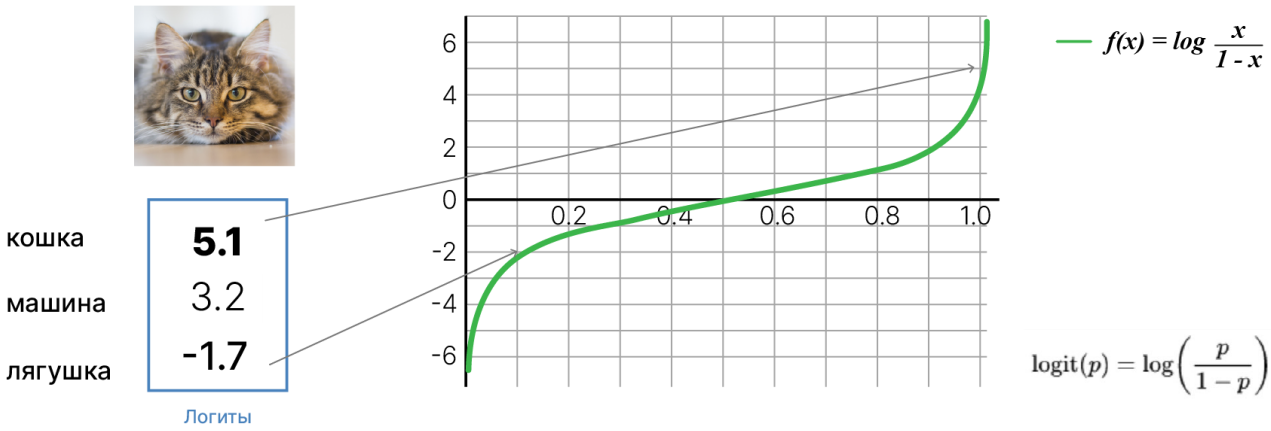
Рассмотрим ещё одну loss-функцию.

SVM loss хороша тем, что интуитивно понятно, как ее посчитать. Но проблема состоит в том, что модель выдает некоторые числа, которые сами по себе невозможно интерпретировать. Сами по себе выходы модели мало что означают. Было бы неплохо, если бы мы сразу, глядя на выход модели для кошки, могли бы их как-то интерпретировать, не просматривая остальные веса. Хорошо бы, чтобы модель выдавала не какую-то абстрактную уверенность, а **вероятность** того, что по ее мнению, на картинке изображена кошка.

Переход к вероятностям

Softmax

[Видео от StatQuest, которое объясняет Softmax]



Перейти к вероятностям мы сможем, проведя с весами некоторые не очень сложные математические преобразования.

На слайде выше показано, почему выходы модели часто называют logit'ами. Если предположить, что у нас есть некая вероятность, от которой мы берем такую функцию (logit), то она может принимать значения от нуля до бесконечности. Мы можем считать, что выходы модели - это logit'ы.

Например, мы могли бы просто взять индекс массива, в котором значение (logit)

```
import numpy as np

logits = [
          5.1, # cat
          3.2, # car
          -1.7, # frog
]
```

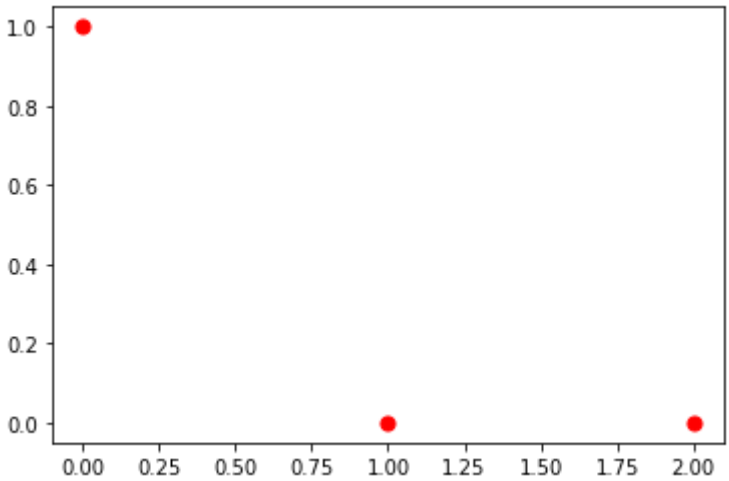
Тогда, чтобы узнать какой класс наша сеть предсказала, мы могли бы просто взять `argmax` от наших `logits`

```
print('Predicted class = %i (Cat)' % (np.argmax(logits)))
```

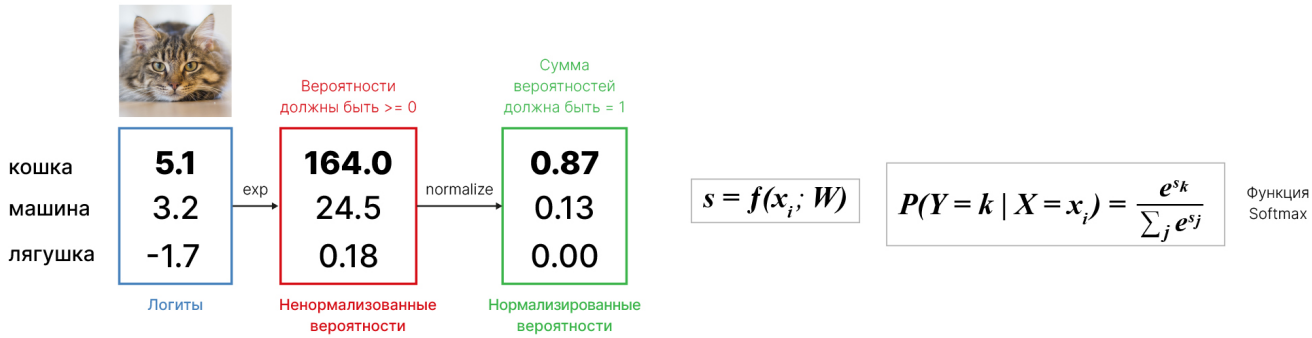
```
Predicted class = 0 (Cat)
```

Но от argmax нельзя посчитать градиент, так как производная от константы равна 0. Соответственно, если бы мы вставили производную от argmax в градиентный спуск, мы бы получили везде нули, и соответственно, наша сеть бы вообще ничему не научилась

```
import matplotlib.pyplot as plt

plt.scatter(np.arange(3), [1,0,0], color='red', s=50)
plt.show()
```



А мы бы хотели получить не logit'ы, а настоящую вероятность на выходе модели. Да еще и таким образом, что бы от наших вероятностей можно было бы посчитать градиент. Для этого мы можем применить к нашим логитам функцию **Softmax**



Мы можем провести над логитами операцию экспоненцирования, то есть число Эйлера (2.71828) **возвести в степень**, соответствующую этому выходу. В результате, мы получим вектор, гарантировано неотрицательных чисел (потому что мы ввели неположительное число в степень, пускай даже отрицательную, то есть выходы могут быть маленькие, но всегда положительные).

Дальше, чтобы можно было интерпретировать эти числа как вероятности, их сумма должна быть равна единице. Мы должны их нормализовать, то есть **поделить на сумму**. Это преобразование называется **Softmax функцией**.

Получаются вероятности, то есть числа, которые можно интерпретировать таким образом.

$$\text{Softmax}_{\text{кошка}} = \frac{e^{5.1}}{e^{5.1}+e^{3.2}+e^{-1.7}}$$

```
def softmax(logits):
    return np.exp(logits)/np.sum(np.exp(logits))

print(softmax(logits))
print('Sum = %.2f' % np.sum(softmax(logits)))
```

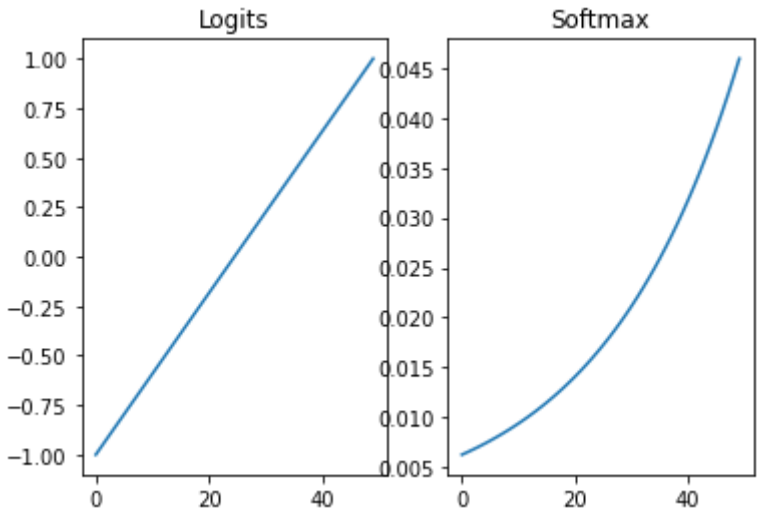
```
[0.86904954 0.12998254 0.00096793]
Sum = 1.00
```

Можно обратить внимание, что Softmax, никоим образом не поменял порядок значений. Самому большому logit'у соответствует самая большая вероятность, а самому маленькому, соответственно самая маленькая

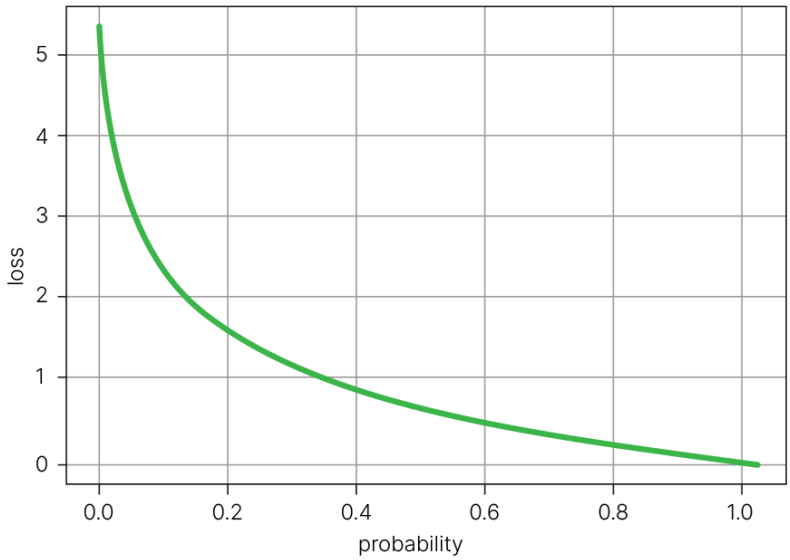
Посмотрим на графиках. Возьмем массив случайных логитов и применим к ним softmax

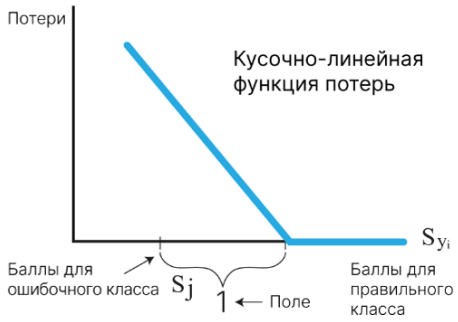
```
rand_logits = np.linspace(-1,1,50)
fig,ax = plt.subplots(ncols=2)

ax[0].plot(np.arange(50), rand_logits)
ax[0].set_title('Logits')
```

```
ax[1].plot(np.arange(50), softmax(rand_logits))
ax[1].set_title('Softmax')
plt.show()
```



Cross-entropy / log loss





$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Теперь, с использованием такого вектора можно определить другую loss функцию. Если не вникать в детали, можно **взять от нее логарифм**. Тогда loss от такого выхода будет выглядеть вот таким образом:

$$L_i = -log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}})$$

как нормализованный выход от верного класса на сумму остальных, к которому добавили минус.

Получится график loss (слева). Его плюс заключается в том, что у него нет участка с плато, практически по всей длине функция получается гладкой, с хорошими мощными производными. Когда loss большой, производная тоже большая, и за счет этого можно быстро обучать модель. Для кусочно-линейной функции потерь она же равна константе.

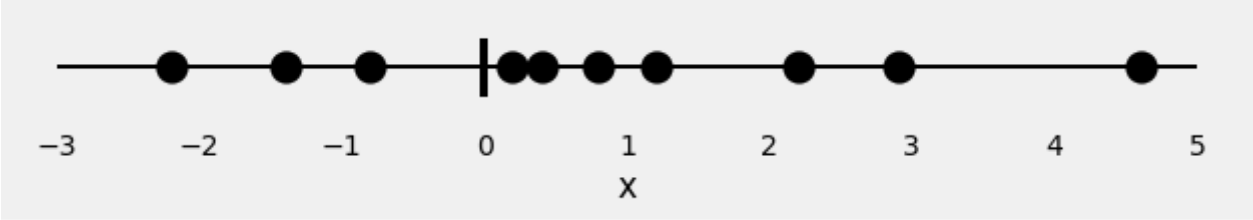
До тех пор, пока модель не будет работать с точностью 100%, то есть пока все остальные выходы не станут нулевыми, мы можем продолжить обучение, даже если у нас нет явной регуляризации.

Осталось выяснить, почему такая простая на вид функция **называется так сложно** - Кросс-энтропия.

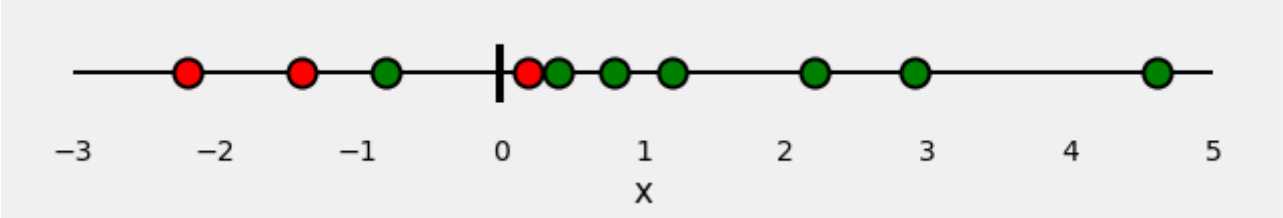
Определение Кросс-энтропии

Начнем с простого примера, пусть у нас есть 10 точек со следующими значениями признака x:

```
x = [-2.2, -1.4, -0.8, 0.2, 0.4, 0.8, 1.2, 2.2, 2.9, 4.6]
```



Пусть наши точки принадлежат двум классам: зеленый и красный:



Перед нами простая задача классификации: по признаку x предсказать класс наших точек. Мы можем переформулировать задачу как нахождение вероятности того, что точка зеленая или красная. В идеальной ситуации для зеленой точки вероятность того, что она зеленая равна 1, в то же время вероятность того, что красная точка — зеленая должна быть равна 0.

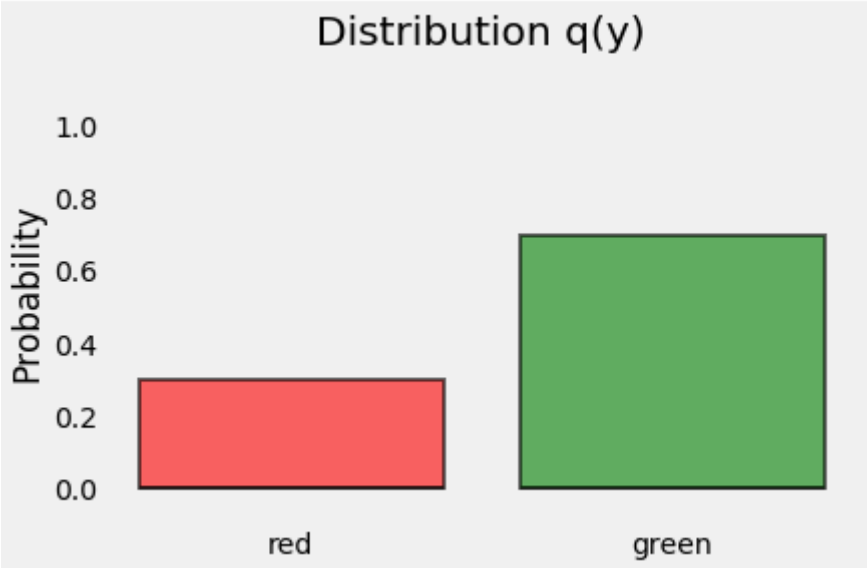
Вероятности, которые выдает, обучаемая нами модель, зачастую далеки от идеальной ситуации. В нашем примере сравнить насколько сильно предсказанная вероятность класса отличается от вероятности, выдаваемой "идеальной моделью" можно за счет **бинарной кросс-энтропии** частного случая кросс-энропии.

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

где y — метка класса (1 для зеленого, 0 для красного), которую можно также интерпретировать как вероятность, предсказанную "идеальной моделью", $p(y)$ — вероятность того, что точка зеленая, предсказываемая оцениваемой моделью.

Какое же отношение энтропия имеет к этой формуле? Давайте углубимся в детали.

Поскольку y представляет метку классов точек, то его распределение $q(y)$ выглядит следующим образом:



Энтропия — мера неуверенности, связанная с распределением $q(y)$. Какова была бы мера неуверенности распределения $q(y)$ если бы все точки были зелеными? Так как у нас бы не было сомнений насчет цвета точки (он всегда зеленый), значение энтропии было бы 0. Теперь представим другую ситуацию, пусть у нас поровну точек зеленого и красного цвета. Для нас это наихудшая ситуация, поскольку попытка определить цвет точки по сути представляет случайное угадывание. В этом случае энтропия вычисляется по формуле Хартли:

$$H(q) = log(2)$$

Мы рассмотрели два крайних случая, но как быть с промежуточными ситуациями? Для этих случаев мы можем использовать формулу Шеннона:

$$H(q) = -\sum_{c=1}^{C} q(y_c) \cdot log(q(y_c))$$

где C — количество классов. Нетрудно заметить, что формула Хартли является частным случаем формулы Шеннона.

Таким образом, зная истинное распределение случайной величины, мы можем рассчитать его энтропию. А что будет если мы попытаемся аппроксимировать истинное распределение $q(y)$ некоторым другим распределением $p(y)$? Допустим, что наши цветные точки подчиняются этому распределению $p(y)$, также мы знаем, что исходят они из неизвестного нам истинного распределения $q(y)$, если мы посчитаем следующую энтропию, это и будет **кросс-энтропия**:

$$H_p(q) = -\sum_{c=1}^{C} q(y_c) \cdot log(p(y_c))$$

Если окажется, что распределения $p(y)$ и $q(y)$ совпадают, в этом случае энтропия $H(q)$ и кросс-энтропия $H_p(q)$ также будут совпадать. Однако в реальности такое случается редко и кросс-энтропия бывает больше энтропии истинного распределения

$$H_p(q) - H(q) \geq 0$$

Разница между кросс-энтропией и энтропией называется **дивергенцией Кульбака-Лейблера**, которая является мерой различия между двумя распределениями:

$$D_{KL}(q||p) = H_p(q) - H(q) = \sum_{c=1}^{C} q(y_c) \cdot [log(q(y_c)) - log(p(y_c))]$$

Это значит, что чем ближе $p(y)$ к $q(y)$, тем меньше будет значение дивергенции Кульбака-Лейблера и, следовательно, меньше значение кросс-энтропии.

Таким образом, мы хотим добиться, чтобы модель, которую мы оцениваем порождала $p(y)$ близкое к $q(y)$. Для этих целей мы стремимся **минимизировать кросс-энтропию**

Энтропия

p = истинное распределение

q = прогнозируемое распределение

Расстояние Кульбака — Лейблера

— неотрицательнозначный функционал, являющийся несимметричной мерой удалённости друг от друга двух вероятностных распределений, определённых на общем пространстве элементарных событий.

$$H(P,Q) = -\sum_{x} P(x) \log Q(x)$$

$$D_{KL}(P \parallel Q) = \sum_{i=1}^{n} p_i \log \frac{p_i}{q_i} \quad = H(P,Q) - H(P)$$

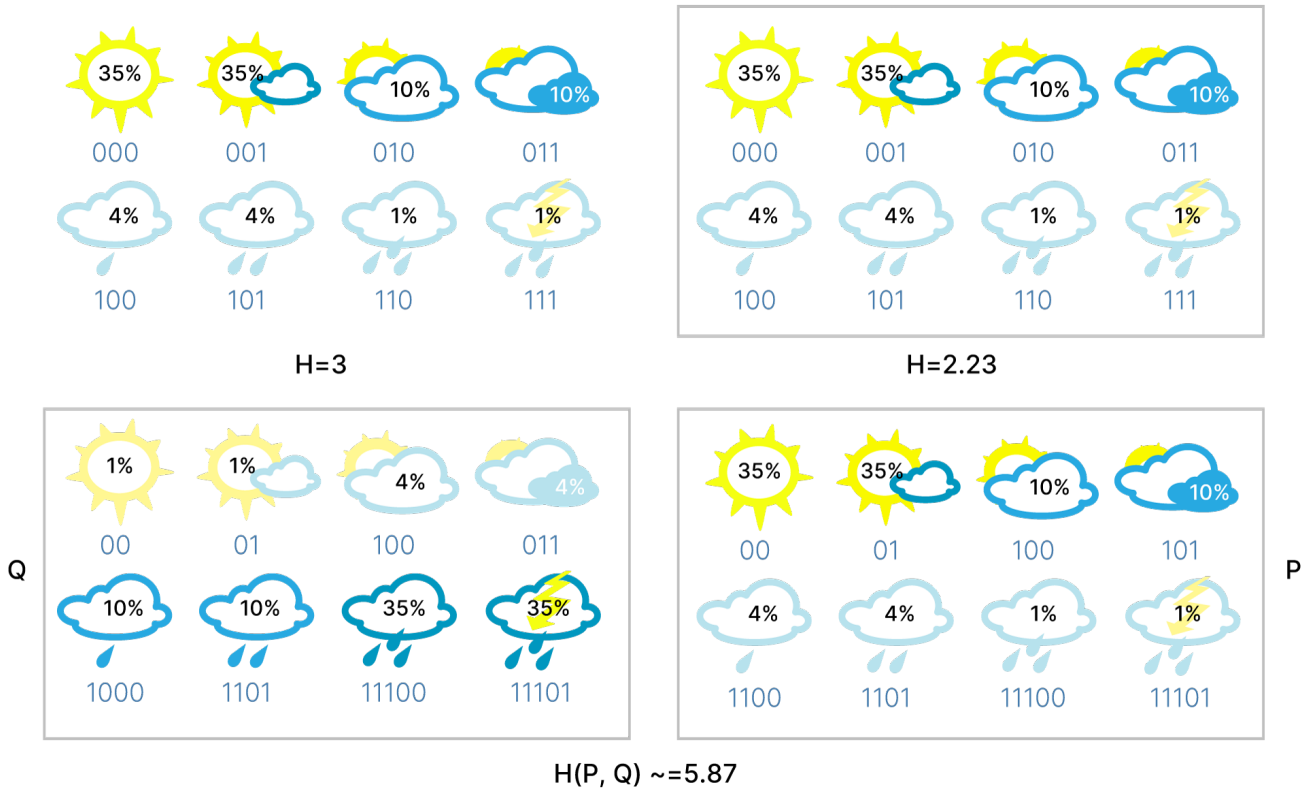
Формула Шеннона

где Vмакс – максимальная скорость передачи (бит/сек), – полоса пропускания линии передачи и, одновременно, полоса частот, занимаемая сигналами (если не используется частотное разделение каналов), S/N – отношение сигнал/шум по мощности.

$$\mathrm{H}(x) = -\sum_{i=1}^{N} \mathrm{p_i} \log(\mathrm{p_i}) = \sum_{i=1}^{N} \mathrm{p_i} \log\left(\frac{1}{\mathrm{p_i}}\right)$$

$$-\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

Если мы рассматриваем выходы нашей модели как вероятности, то мы можем их сравнивать. У нас есть номер правильного класса. Соответственно, можно сказать, что вероятность этого класса - единица, а всех остальных 0, и получить, таким образом, вероятностное распределение. Выход модели тоже можно интерпретировать как вероятности (тоже можно получить вероятностное распределение). И для работы с этими распределением есть некоторый математический аппарат, который основан на понятии энтропии, который ввел Клод Шеннон.



Идея в следующем: у нас есть некоторые данные, которые мы передаем по каналу связи. Например, у нас есть метеостанция, которая сообщает прогноз погоды. Допустим, она может передавать 8 вариантов прогноза. Мы в каждый момент времени получаем от нее сообщение. Предположим, что потребуется 3 бит, чтобы передавать это сообщение.

Допустим, мы знаем некую вероятность, с которой в нашем регионе может наступить хорошая либо плохая погода. То есть мы знаем вероятностное распределение, по которому у нас, допустим, солнечное место, где почти не бывает бури. Если мы знаем об этом, мы можем сэкономить на передаче информации. Мы можем закодировать наиболее вероятные сообщения двумя битами, причем придумать такие коды, чтобы они почти не пересекались. Они будут начинаться с нуля и передавать два бита вместо трех. Таким образом, можно сэкономить на передачи информации. То есть идея такая: **если мы знаем, что события маловероятны, мы можем кодировать их более длинной цепочкой, а более вероятные события - более короткими цепочками**. В этом случае в среднем количество информации, которое можно передать, сократится.

Формула Шеннона позволяет посчитать, насколько сильно мы можем сэкономить для конкретного вероятностного распределения. То есть если подставить эти данные в формулу, то получим 2,23.

На практике реализовать это, используя биты, возможно, не выйдет, но это свойство данного вероятностного распределения можно посчитать. Оно несёт в себе некоторое количество полезной информации, соответствующее этой энтропии.

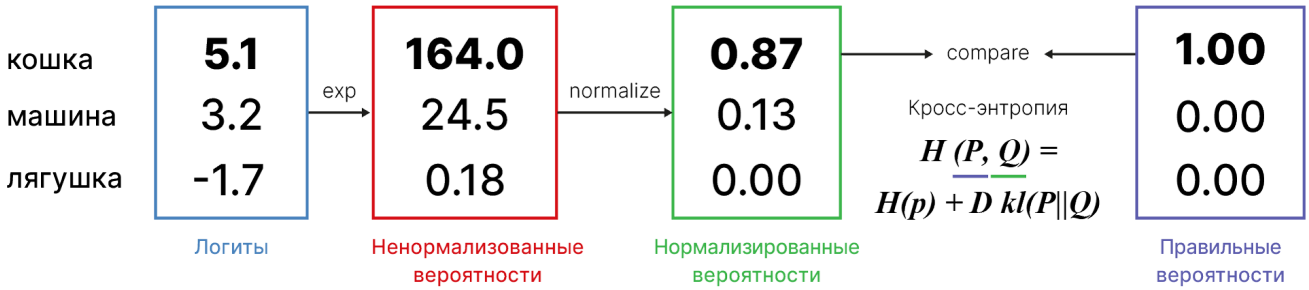
Что же такое кросс-энтропия?

Давайте представим, что мы каким-то образом начали кодировать эту информацию, прошили погодную станцию так, что она может эту кодированную информацию передавать, а потом перенесли в другой регион, где ситуация противоположная. И мы кодируем события, которые стали частыми, длинными цепочками, а маловероятные - короткой, и у нас в этом случае явно передается больше информации, чем мы могли бы, если бы сделали все банально.

Кросс-энтропия позволяет посчитать, насколько большая будет потеря в данном случае. То есть это некий **способ сравнить между собой вероятностные распределения**.

Что, собственно говоря, отображено в формуле, потому что мы считаем кросс-энтропию по нашему вектору P и Q. Вектор P состоит из нулей и единиц, соответственно, во всех случаях, кроме одного, это выражение будет нулевым, кроме одного случая с единицей. Случай с единицей соответствует нашему правильному классу. Поэтому сумма исчезает, остаются логарифм и вектор, а вероятность для правильного класса мы считаем как Soft max. Отсюда название кросс-энтропия.

(В теории информации кросс-энтропия между двумя распределениями вероятностей p и q по одному и тому же базовому набору событий измеряет среднее число битов, необходимых для идентификации события, взятого из набора, если схема кодирования, используемая для набора, оптимизирована для оценочного распределения вероятностей q, а не для истинного распределения p.)



Аналогичное обоснование: есть понятие дивергенция, которая позволяет оценить расхождение между вероятностными распределениями, которая нам здесь и нужна. Есть дивергенция **Кульбака-Лейблера**, которая выражается через кросс-энтропию, а энтропия от нашего вектора нулевая. Поэтому фактически здесь кросс-энтропия равна дивергенции, которая показывает, насколько не похожи два распределения.

Softmax

кошка **5.1**

машина 3.2

лягушка -1.7

Хотим интерпретировать необработанные баллы классификатора как **вероятности**

$$s = f(x_i; W)$$

$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Функция Softmax

Максимизируйте вероятность правильного класса.

$$L_i = -\log P(Y = y_i | X = x_i)$$

Складываем все это вместе:

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Эта функция очень популярна, она используется при обучении реальных моделей на практике. Поскольку внутри находится Softmax, часто ее называют Softmax-классификатором, что, строго говоря, не совсем верно. Здесь функция потерь - кросс-энтропия, Softmax же просто используется внутри нее.

Градиент функции потерь Кросс-энтропии

[Cross Entropy Loss]

Подавляющее большинство методов классификации и регрессии сформулированы в терминах евклидовых или метрических пространств, то есть подразумевают представление данных в виде векторов одинаковой размерности. В реальности мы имеем дело с категориальными данными, которые нужно представить в виде вектора. One Hot Encoding подразумевает создание вектора, размером равным количеству классов, все из которых равны нулю за исключением одного. На позицию, соответствующую значению класса мы помещаем 1.

Например, если у нас 10 классов, тогда для каждого класса соответственно

Класс 0 = [**1**,0,0,0,0,0,0,0,0,0]
Класс 1 = [0,**1**,0,0,0,0,0,0,0,0]
...
Класс 9 = [0,0,0,0,0,0,0,0,0,**1**]

```
num_classes = 10
```

```
Y_class = 5

one_hot = np.zeros(num_classes)
one_hot[Y_class] = 1
Y_i = one_hot
print(Y_i)
```

```
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Практическое вычисление SoftMax

Когда мы возводим число в степень, может получиться очень большое число и произойти переполнение памяти. Неизвестно, какие будут у модели выходы.

```
from warnings import simplefilter
simplefilter("ignore", category=RuntimeWarning)

f = np.array([123, 456, 789])
p = np.exp(f) / np.sum(np.exp(f))
print(p)
```

```
[ 0.  0. nan]
```

Поэтому мы можем вычесть из каждого s_i положительную константу, чтобы уменьшить значения экспонент. В качестве константы можно выбрать максимальный элемент этого вектора, тогда у нас гарантированно не будет очень больших чисел, и такой способ будет работать более стабильно.

$$M = \max_j s_{y_j}$$

$$s_{y_i}^{new} = s_{y_i} - M$$

$$\frac{e^{s_{y_i}^{new}}}{\sum_j e^{s_{y_j}^{new}}} = \frac{e^{s_{y_i} - M}}{\sum_j e^{s_{y_j} - M}} = \frac{e^{s_{y_i}} e^{-M}}{\sum_j e^{s_{y_j}} e^{-M}} = \frac{e^{-M} e^{s_{y_i}}}{e^{-M} \sum_j e^{s_{y_j}}} = \frac{e^{s_{y_i}}}{\sum_j e^{s_{y_j}}}$$

```
f = np.array([123, 456, 789])
f -= f.max()
p = np.exp(f) / np.sum(np.exp(f))
print(f, p)
```

```
[-666 -333    0] [5.75274406e-290 2.39848787e-145 1.00000000e+000]
```

✓ 0 сек. выполнено в 14:59