Машинное обучение

Пензин М.С.

In [1]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
import warnings
warnings.filterwarnings('ignore')
from pylab import rcParams
rcParams['figure.figsize'] = 9, 6
```

In [2]:
```
def AMS(w, y, y_pred):
    """
        Расчет метрики, работает только с Numpy-массивами
    """
    s = (w * (y == 1) * (y_pred == 1)).sum()
    b = (w * (y == 0) * (y_pred == 1)).sum()
    bReg = 10.
    return np.sqrt(2 * ((s + b + bReg) *
                        np.log(1 + s / (b + bReg)) - s))
```

In [3]:
```
## Долгая скучная инициализация данных

def loadTrain():
    import zipfile
    z = zipfile.ZipFile("data/training.zip")
    df = pd.read_csv(z.open("training.csv"))
    return df

df = loadTrain()

df[["log_PRI_met_sumet", "log_PRI_met", "log_DER_pt_ratio_lep_tau"]] = np.log(
    df[["PRI_met_sumet", "PRI_met", "DER_pt_ratio_lep_tau"]]
)

columns = ["log_PRI_met_sumet", "log_PRI_met", "log_DER_pt_ratio_lep_tau"]

df["Y"] = df["Label"].map({
    "s": 1,
    "b": 0,
})

from sklearn.model_selection import train_test_split

# Разобьем наши данные
X_train, X_test, y_train, y_test = train_test_split(
    df[columns + ["Weight"]].to_numpy(), df["Y"].to_numpy(),
    test_size=0.3, random_state=13)

# Отщепляем последний столбец

w_train = X_train[:, -1]
w_test = X_test[:, -1]

X_train = X_train[:, :-1]
X_test = X_test[:, :-1]
```

8. Линейные модели

Регрессия

В общем случае, при решении задачи регрессии, на необходимо найти функцию $f(\vec{x})$, которая по набору признаков $\vec{x}$ дает оценку исхода y.

Давайте рассмотрим самый простой вариант такой функции:

$$\begin{aligned} f(\vec{x}) &= \omega_0 x_0 + \omega_1 x_1 + \ldots + \omega_k x_K \\ &= \sum_{k=0}^{K} \omega_k x_k \\ &= \vec{x}^T \vec{\omega} \end{aligned}$$

Здесь x_0 всегда равен 1, $\vec{\omega}$ - вектор параметров модели(веса)

Добавим случайную ошибку ε

$$y_i = \vec{x}^T \vec{\omega} + \varepsilon_i$$

такую, что

1. $\forall i,\ E[\varepsilon_i] = 0$
2. $\forall i,\ D[\varepsilon_i] = \sigma < \infty$
3. $\forall i \neq j, \mathrm{Cov}(\varepsilon_i, \varepsilon_j) = E[\varepsilon_i \varepsilon_j] - E[\varepsilon_i]E[\varepsilon_j] = 0$

$$\vec{y} = X\vec{\omega} + \vec{\varepsilon}$$

Здесь

$$\begin{aligned} \vec{y} &= [y_1, y_2, \ldots, y_N]^T \\ X &= [\overrightarrow{x_1}, \overrightarrow{x_2}, \ldots, \overrightarrow{x_N}]^T \\ \vec{x} &= [1, x_1, \ldots, x_K]^T \\ \vec{\omega} &= [\omega_0, \omega_1, \ldots, \omega_K]^T \\ \vec{\varepsilon} &= [\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_K]^T \end{aligned}$$

Как оценить значения весов $\vec{\omega}$?

Функция потерь

Здесь нам на помощь приходит функция потерь. Нам нужно найти такие значения, чтобы она принимала наименьшее значение

$$\hat{\vec{\omega}} = \arg\min_{\omega} L(\mathbf{X}, \vec{\omega})$$

В простейшем случае, такой функцией выступает сумма квадратов отклонений

$$\begin{aligned} L(\vec{\omega}) &= \sum_{i=1}^{N} \left(y_i - \vec{x}_i^T \vec{\omega}\right)^2 \\ &= (\vec{y} - X\vec{\omega})^T (\vec{y} - X\vec{\omega}) \\ &= ||\vec{y} - X\vec{\omega}||_2^2 \end{aligned}$$

Поиск решения

Для того, чтобы найти решение, нам нужно найти производные по параметрам $\vec{\omega}$

$$\frac{\partial L}{\partial \vec{\omega}} = -2X^T\vec{y} + 2X^TX\vec{\omega} = 0$$

$$X^TX\vec{\omega} = X^T\vec{y}$$

Полиномиальная регрессия

Мы можем ввести новые признаки в виде $\prod_i \vec{x}_i^{p_i}$.

Например, для одного признака

$$y = \omega_0 + \omega_1 x + \ldots + \omega_p x^p + \varepsilon$$

для двух признаков (x_1, x_2) и для второго порядка - это даст $(1, x_1, x_2, x_1^2, x_2^2, x_1x_2)$

Проблемы

Что будет, если ранг матрицы X^TX будет меньше N?

Это соответствует тому, что какие-то признаки являются линейной комбинацией других признаков (мультиколлинеарность).

В нашем решением перестает существовать обратная матрица

$$\vec{\omega} = (X^TX)^{-1}X^T\vec{y}$$

Регуляризация

В случае, если обратная матрица $(X^TX)^{-1}$ не существует, это переводит нашу задачу в категорию некорректно поставленных.

Тихонов предложил ввести дополнительное внешнее условие - регуляризацию.

$$L = ||\vec{y} - X\vec{\omega}||_2^2 + \lambda J(\vec{\omega})$$

Самый простой случай - это l_2-регуляризация

$$J(\vec{\omega}) = ||\vec{\omega}||_2^2 = \vec{\omega}^T\vec{\omega}$$

Найдем производную

$$\frac{\partial J}{\partial \vec{\omega}} = 2\vec{\omega}$$

$$\frac{\partial L}{\partial \vec{\omega}} = -2X^T\vec{y} + 2(X^TX + \lambda I)\vec{\omega} = 0$$

$$(X^TX + \lambda I)\vec{\omega} = X^T\vec{y}$$

$$\vec{\omega} = (X^TX + \lambda I)^{-1}X^T\vec{y}$$

Такая регрессия также называется гребневой (ridge regression). Её основной минус в том, что оценки параметров начинают получатся смещенным к нулю.

Регуляризация (попытка 2)

В предыдущем случае (l_2-регуляризация), если мы попытаемся восстановить кубическую

функцию с помощью полинома 10 степени, то его все 10 коэффициентов будут ненулевыми.

Хотелось бы придумать такой регуляризатор, чтобы ненужные коэффициенты равнялись нулю, чтобы снова можно было получить кубическую функцию.

Такой регуляризатор есть и он соответствует добавлению регуляризующего члена в виде:

$$J(\vec{\omega}) = ||\vec{\omega}||_1 = \sum_{i=0}^{K} |\omega_i|$$

$$\frac{\partial L}{\partial \vec{\omega}} = -2X^T\vec{y} + 2(X^TX)\vec{\omega} + \lambda \operatorname{sign}(\vec{\omega})$$

Такая регуляризация называется l_1-регуляризацией или Least Absolute Shrinkage and Selection Operator(LASSO).

И, увы, задача не имеет решения в явном виде.

Метод градиентного спуска

Градиент соответствуют вектору, указывающему направление наибольшего роста функции. Следовательно. если идти в обратном направлении, то можно рано или поздно придти в минимум (в какой-нибудь).

$$\vec{\omega}^{(k+1)} = \vec{\omega}^{(k)} - \alpha \frac{\partial L}{\partial \vec{\omega}}\bigg|_{\vec{\omega} = \vec{\omega}^{(k)}}$$

Стохастический градиентный спуск

В случае, когда у нас очень большая выборка данных, посчитать градиент функционала вида

$$L = \frac{1}{2N}||\vec{y} - X\vec{\omega}||_2^2$$

не является уже такой простой задачей.

Вместо обычного шага против градиента, мы используем следующее выражение

$$\vec{\omega}^{(k+1)} = \vec{\omega}^{(k)} - \alpha\, g(\vec{\omega}^{(k)}, i)$$

где

$$E_i[g(\vec{\omega}, i)] = \frac{\partial L}{\partial \vec{\omega}}$$

$$L = \frac{1}{2N}\sum_{i=1}^{N}(y_i - \vec{x}_i^T\vec{\omega})^2$$

$$\frac{\partial L}{\partial \vec{\omega}} = -\frac{1}{N}\sum_{i=1}^{N}(y_i - \vec{x}_i^T\vec{\omega})\vec{x}_i$$

отсюда видно, что

$$g(\vec{\omega}, i) = (y_i - \vec{x}_i^T\vec{\omega})\vec{x}_i$$

Линейный классификатор

При построении линейного классификатора, мы пытаемся разделить пространство признаков с помощью гиперплоскости на два подпространства, соответствующих целевым классам.

Пусть у нас есть задача бинарной классификации, при метки целевого класса обозначим "-1" и "+1".

В этом случае, простейших линейный классификатор можно записать как

$$f(\vec{x}) = \text{sign}\,(\vec{x}^T\vec{\omega})$$

Логистическая регрессия

Частным случаем линейного классификатора является логистическая регрессия. С помощью неё можно оценить вероятность принадлежности прецедента $\vec{x}$ к определенную классу

$$p_+ = P(y = 1|\vec{x}, \vec{\omega})$$
$$p_- = P(y = -1|\vec{x}, \vec{\omega})$$

Будем рассматривать вместо p_+ и p_- их отношение:

$$R(+) = \frac{p_+}{1 - p_+} \in [0, \infty)$$

Возьмем логарифм

$$\log R(+) = \log(p_+) - \log(1 - p_+) \in (-\infty, \infty)$$

Для некоторого набора весов $\vec{\omega}$, выражение $\vec{x}^T\vec{\omega} = 0$ определяет гиперплоскость в пространстве параметров.

Пусть $\log R(+) = \vec{x}^T\vec{\omega}$.

Тогда с помощью небольших преобразований, можем получить

$$p_+ = \frac{R(+)}{1 + R(+)} = \frac{e^{\vec{x}^T\vec{\omega}}}{1 + e^{\vec{x}^T\vec{\omega}}} = \frac{1}{1 + e^{-\vec{x}^T\vec{\omega}}}$$

Мы получили сигмоиду

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

Хорошо, а как же теперь оценить параметры модели $\vec{\omega}$?

По принципу максимального правдоподобия.

$$P(y = 1|\vec{x}, \vec{\omega}) = \sigma(\vec{x}^T\omega)$$
$$P(y = -1|\vec{x}, \vec{\omega}) = 1 - \sigma(\vec{x}^T\omega) = \sigma(-\vec{x}^T\omega)$$

Или одним выражением

$$P(y = y_i|\vec{x}_i^T, \vec{\omega}) = \sigma(y_i\vec{x}_i^T\vec{\omega})$$

Если наш классификатор действительно выдает вероятности, то они должны согласовываться с нашей выборкой (считаем, что каждое наблюдение не зависит от других наблюдений)

$$P(\vec{y}|\vec{\omega}) = \prod_{i=1}^{N} P(y = y_i|\vec{x}_i^T, \vec{\omega}) = \prod_{i=1}^{N} \sigma(y_i \vec{x}_i^T \vec{\omega})$$

и нам фактически нужно максимизировать данный функционал

Это удообно сделать в виде логарифма

$$\log P(\vec{y}|\vec{\omega}) = \sum_{i=1}^{N} \log \sigma(y_i \vec{x}_i^T \vec{\omega}) = -\sum_{i=1}^{N} \log\left(1 + e^{-y_i \vec{x}_i^T \vec{\omega}}\right)$$

$$L_{logloss} = \sum_{i=1}^{N} \log\left(1 + e^{-y_i \vec{x}_i^T \vec{\omega}}\right)$$

$$L_{logloss} = \sum_{i=1}^{N} (-[y = -1] \log p_- - [y = 1] \log p_+)$$

Регуляризация логистической регрессии

Всё практически без изменений

$$L = L_{logloss} + \frac{1}{C}||\vec{\omega}||_2^2$$

здесь C - это гиперпараметр нашей модели.

В **sklearn** поддерживаются линейные модели. В частности, они представлены

- LinearRegression, Ridge, Lasso, LassoCV, RidgeCV
- LogisticRegression, LogisticRegressionCV, RidgeClassifier, LassoClassifier
- и другие

In [6]:
```
from sklearn.linear_model import RidgeClassifier
from sklearn.metrics import accuracy_score

rc = RidgeClassifier()
rc.fit(X_train, y_train)
predict = rc.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.67648
AMS = 0.6383239186781775
```

In [10]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, PolynomialFeatures

pipe = Pipeline([
    ("poly", PolynomialFeatures(2)),
    ('scaler', StandardScaler()),
    ('ridge', RidgeClassifier())])
pipe.fit(X_train, y_train)
predict = pipe.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.6979066666666667
AMS = 0.9396654848979002
```

Плюсы

- очень хорошо изученные методы
- довольно быстро работают
- позволяет оценивать вероятность
- можно строить нелинейные плоскости, если добавлять полиномиальные признаки

Минусы

- плохо работают в задачах, где зависимость ответов от признаков нелинейная
- теорема Гаусса-Маркова на практике практически никогда не выполняется, из-за чего линейные модели работают обычно хуже, чем ожидается

9. Метод опорных векторов

Метод опорных векторов

[Support Vector Machine (https://svmtutorial.online/download.php?file=SVM_tutorial.pdf)](https://svmtutorial.online/download.php?file=SVM_tutorial.pdf) - алгоритм обучения с учителем в семействе линейных моделей.

Пусть есть набор признаков X с метками класса $y_n \in \{-1, +1\}$.

Напомним, как выглядит линейный классификатор можно представить в виде:

$$f(\vec{x}) = \text{sign}\left(\vec{\omega}^T\vec{x} + \omega_0\right) = \text{sign}\left(\langle\vec{\omega}, \vec{x}\rangle + \omega_0\right)$$

Здесь $\vec{\omega} = (\omega_1, \dots, \omega_K)$ и $\vec{x} = (x_1, \dots, x_K)$.

$$\langle\vec{\omega}, \vec{x}\rangle = -\omega_0$$

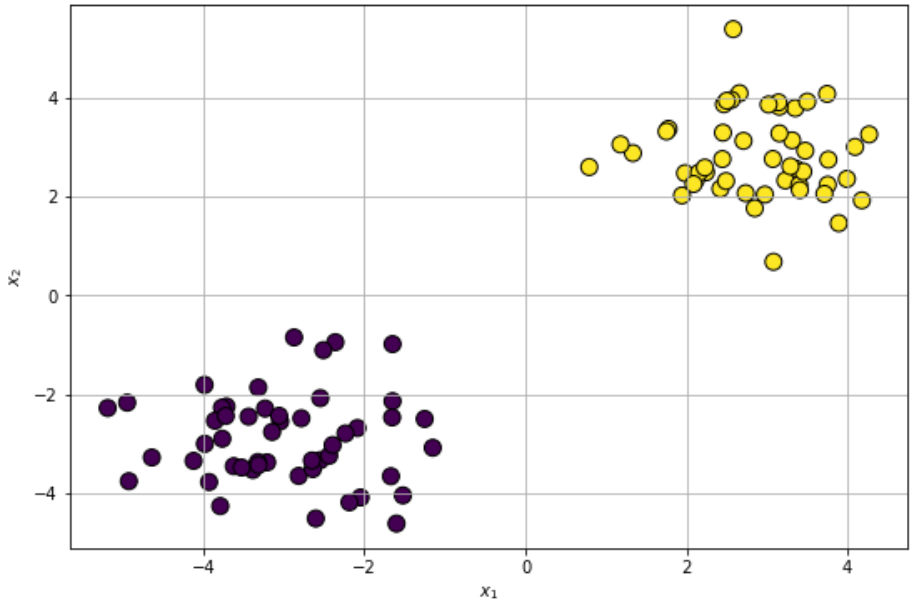
Фактически это выражение определяет гиперплоскость, разделяющую прецеденты на два класса. Если класс лежит с одной стороны плоскости - это (+1), со другой стороны - (-1).

Выражение $\langle\vec{\omega}, \vec{x}\rangle$ фактически проецирует вектор признаков на нормаль нашей разделяющей плоскости, знак этой проекции с учетом сдвига ω_0 дает классификатору возможность определить класс, а ее величина будет отображать степень уверенности в данном классе, ведь чем дальше от разделяющей поверхности, тем меньше возможности ошибиться.

Допустим у нас есть набор данных, которые мы должны разнести по классам.

In [11]:
```
from sklearn.datasets import make_blobs
X, Y = make_blobs(centers=[[-3, -3], [3, 3]], random_state=13)
```

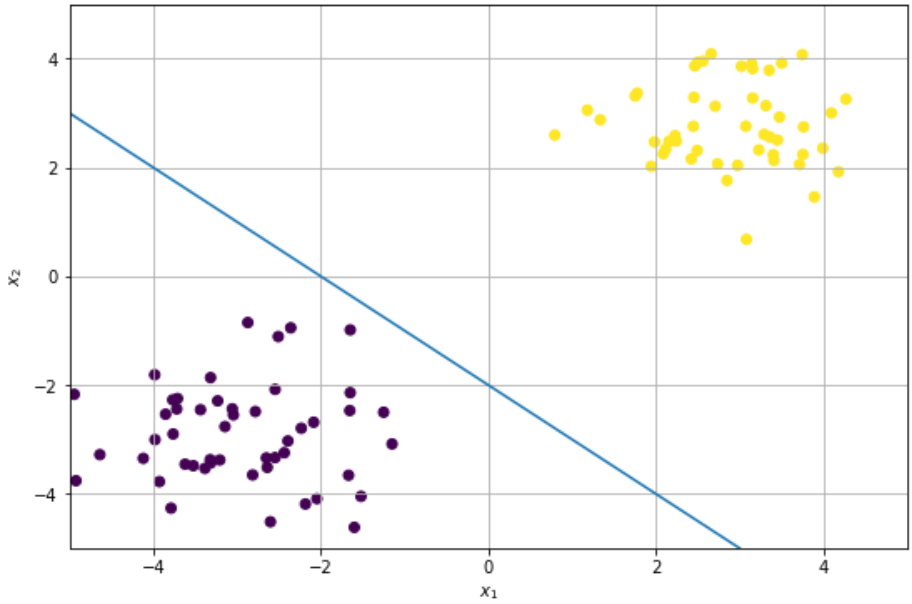
In [12]:
```
plt.scatter(X[:,0], X[:,1], c=Y, edgecolor="k", s=100);
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.grid()
plt.show()
```



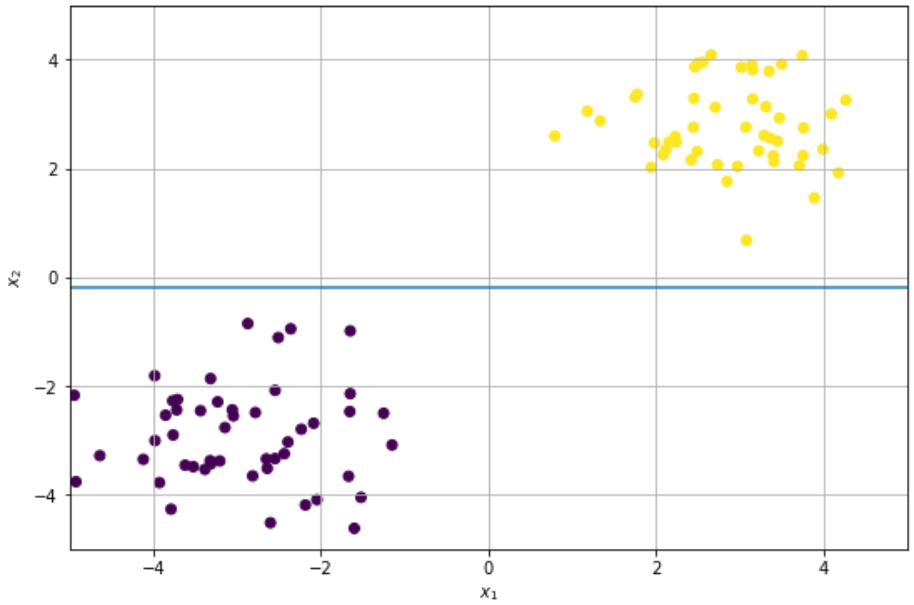
Очевидно, что данные идеально разделяются в признаковом пространстве.

In [13]:
```
def plot(X, Y, angle=135, b=0):
    plt.scatter(X[:,0], X[:,1], c=Y);
    x = np.linspace(-6, 6, 100)
    plt.ylim((-5, 5))
    plt.xlim((-5, 5))
    plt.plot(x, np.tan(angle/180 * np.pi) * x + b)
    plt.xlabel("$x_1$")
    plt.ylabel("$x_2$")
    plt.grid()
    plt.show()
```

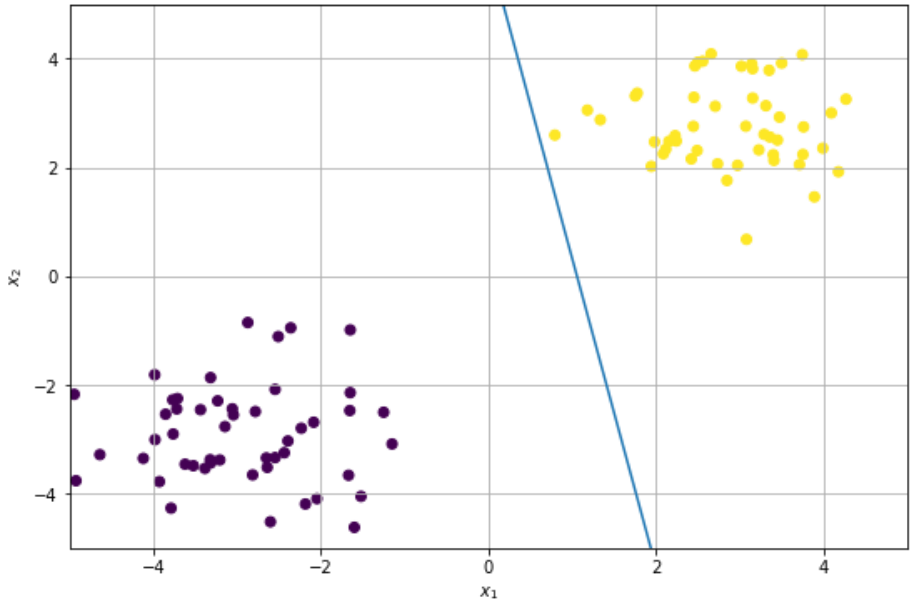
In [14]:
```
plot(X, Y, 135, -2)
```



In [15]:
```
plot(X, Y, 0, -0.2)
```



In [16]:
```
plot(X, Y, 100, 6)
```



А можно ли найти наилучшее положение разделяющей гиперплоскости?

Зазор

Логичным предположением является, максимизация **зазора**, т.е. расстояния между классами, что будет способствовать более уверенной классификации. Разделяющая гиперплоскость будет просто находится в середине зазора.

Заметим, что мы можем определить $\vec{\omega}$ и ω_0 с точностью до нормировки, т.е. если мы их умножим на одну и ту же положительную константу, то результат классификации останется тем же.

Выберем такую нормировку, чтобы для всех прецедентов на краях зазора выполнялось условие

$$\langle \vec{\omega}, \vec{x}_m \rangle + \omega_0 = y_m$$

таким образом для всех объектов будет иметь место

$$\langle \vec{\omega}, \vec{x}_n \rangle + \omega_0 = \begin{cases} \leq -1, & \text{если } y_n = -1 \\ \geq +1, & \text{если } y_n = +1 \end{cases}$$

В общем виде

$$y_n \left(\langle \vec{\omega}, \vec{x}_n \rangle + \omega_0 \right) \geq 1$$

А следующее условие, задаёт полосу разделяющую классы

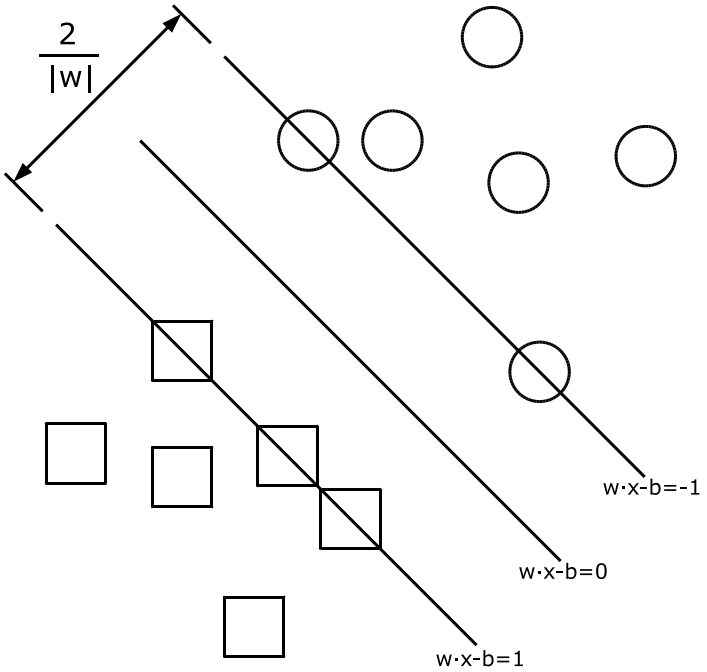
$$-1 < \langle \vec{\omega}, \vec{x} \rangle < +1$$

Для того, чтобы зазор был максимальным, расстояние между граничными гиперплоскостями должно быть максимально. Пусть у нас есть два произвольных прецедента $\vec{x}_-$ и $\vec{x}_+$ для классов -1 и $+1$, соответственно, лежащие на границах разделяющей полосы. Тогда ширина ищется довольно просто

$$\left\langle \vec{x}_+ - \vec{x}_-, \frac{\vec{\omega}}{\|\vec{\omega}\|} \right\rangle = \frac{\langle \vec{\omega}, \vec{x}_+ \rangle - \langle \vec{\omega}, \vec{x}_- \rangle}{\|\vec{\omega}\|} =$$
$$= \frac{(-\omega_0 + 1) - (-\omega_0 - 1)}{\|\vec{\omega}\|} = \frac{2}{\|\vec{\omega}\|}$$

$$\frac{2}{\|\vec{\omega}\|}$$

Таким образом, ширина полосы максимальна, когда норма вектора $\vec{\omega}$ минимальна.



Линейно разделимый случай

В случае полной линейной разделимости, мы можем свести нашу задачу к минимизации

квадратичной формы при N ограничениях

$$\begin{cases} \|\vec{\omega}\|^2 \to \min \\ y_n \left(\langle \vec{\omega}, \vec{x}_n \rangle + \omega_0 \right) \geq 1, \text{ где } 1 \leq n \leq N \end{cases}$$

По теореме Куна-Таккера (http://www.machinelearning.ru/wiki/images/b/ba/MO17_seminar8.pdf), эта задача эквивалентна двойственной задачи поиска седловой точки функции Лагранжа

$$\begin{cases} L(\vec{\omega}, \omega_0, \vec{\lambda}) = \frac{1}{2}\|\vec{\omega}\|^2 - \sum\limits_{n=1}^{N} \lambda_n \left(y_n \left(\langle \vec{\omega}, \vec{x}_n \rangle + \omega_0 \right) - 1 \right) \to \min\limits_{\vec{\omega}, \omega_0} \max\limits_{\vec{\lambda}} \\ \lambda_n \geq 0, \text{ где } 1 \leq n \leq N \end{cases}$$

При этом в точке минимума $\vec{\omega}^*$ должно выполнятся условие дополняющей нежесткости

$$\lambda_n^* \left(y_n \left(\langle \vec{\omega}^*, \vec{x}_n \rangle + \omega_0^* \right) - 1 \right) = 0$$

оно оставляет лишь только те λ, где выполняется условие $\langle \vec{\omega}^*, \vec{x}_n \rangle + \omega_0^* = y_n$, для остальных случаев $\lambda = 0$

Далее, нам нужно приравнять к нулю первые производные Лагранжиана

$$\frac{\partial L}{\partial \vec{\omega}} = \vec{\omega} - \sum_{n=1}^{N} \lambda_n y_n \vec{x}_n \qquad \Rightarrow \qquad \vec{\omega} = \sum_{n=1}^{N} \lambda_n y_n \vec{x}_n$$

$$\frac{\partial L}{\partial \omega_0} = -\sum_{n=1}^{N} \lambda_n y_n \qquad \Rightarrow \qquad \sum_{n=1}^{N} \lambda_n y_n = 0$$

Подставим их в Лагранжиан

$$\begin{cases} L(\vec{\lambda}) = -\frac{1}{2} \sum\limits_{n=1}^{N} \sum\limits_{k=1}^{N} \lambda_n \lambda_k y_n y_k \langle \vec{x}_n, \vec{x}_k \rangle + \sum\limits_{n=1}^{N} \lambda_n \to \max\limits_{\vec{\lambda}} \\ \lambda_n \geq 0, \text{ где } 1 \leq n \leq N \\ \sum\limits_{n=1}^{N} \lambda_n y_n = 0 \end{cases}$$

Или если домножить на минус

$$\begin{cases} -L(\vec{\lambda}) = \frac{1}{2} \sum\limits_{n=1}^{N} \sum\limits_{k=1}^{N} \lambda_n \lambda_k y_n y_k \langle \vec{x}_n, \vec{x}_k \rangle - \sum\limits_{n=1}^{N} \lambda_n \to \min\limits_{\vec{\lambda}} \\ \lambda_n \geq 0, \text{ где } 1 \leq n \leq N \\ \sum\limits_{n=1}^{N} \lambda_n y_n = 0 \end{cases}$$

Допустим мы решили задачу и получили набор $\vec{\lambda}$, тогда мы легко можем найти $\vec{\omega}$

$$\vec{\omega} = \sum_{n=1}^{N} \lambda_n y_n \vec{x}_n$$

а ω_0 можно найти используя любой граничный прецедент ($\lambda > 0$)

$$\omega_0 = y_m - \langle \vec{\omega}, \vec{x}_m \rangle$$

Итоговый классификатор

$$f(\vec{x}) = \text{sign}\left(\sum_{m=1}^{M} \lambda_m y_m \langle \vec{x}_m, \vec{x} \rangle + \omega_0\right)$$

стоить отметить, что суммирование производится только по граничным прецедентам, т.н. опорным векторам.

Данный вариант называется SVM с жестким зазором (hard-margin SVM).

Линейно неразделимая выборка

А что делать, если прецеденты нельзя линейно разделить?

Разрешим некоторым объектам неправильного класса находит за разделяющей гиперплоскостью, т.е. разрешим нашему классификатору допускать ошибку на обучающих прецедентах.

$$\begin{cases} \frac{1}{2}\langle \vec{\omega}, \vec{\omega} \rangle + C \sum_{n=1}^{N} \xi_n \to \min\limits_{\vec{\omega}, \omega_0, \vec{\xi}} \\ y_n \left(\langle \vec{\omega}, \vec{x}_n \rangle + \omega_0 \right) \geq 1 - \xi_n, \text{ где } 1 \leq n \leq N \\ \xi_n \geq 0, \text{ где } 1 \leq n \leq N \end{cases}$$

И если всё подставить, получим

$$\begin{cases} \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{N} \lambda_n \lambda_k y_n y_k \langle \vec{x}_n, \vec{x}_k \rangle - \sum_{n=1}^{N} \lambda_n \to \min\limits_{\vec{\lambda}} \\ 0 \leq \lambda_n \leq C, \quad n = 1, \ldots, N \\ \sum_{n=1}^{N} \lambda_n y_n = 0 \end{cases}$$

Что отличается от линейно разделимого случая только тем, что у нас появилось ограничение на λ сверху и в итоговом классификаторе присутствуют объекты-нарушители, помимо опорных объектов.

C - гиперпараметр.

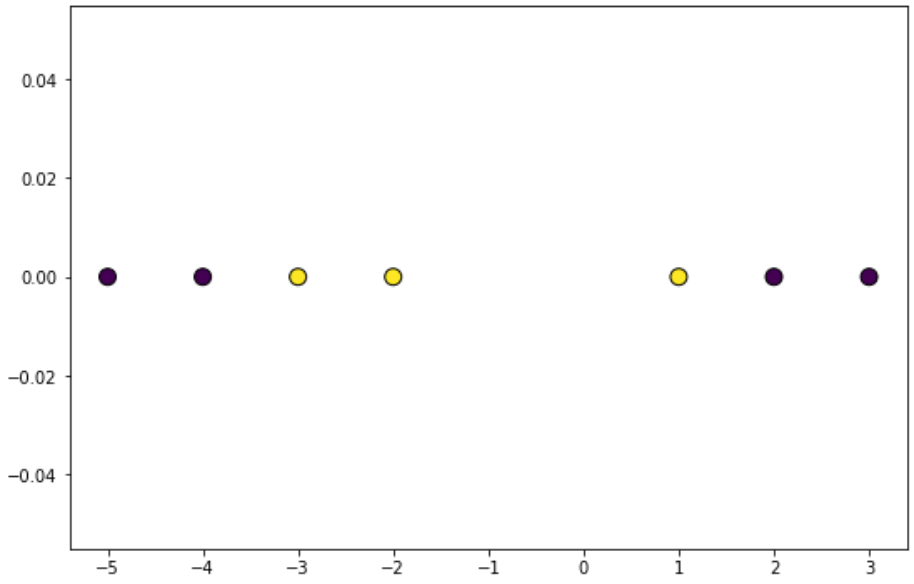
Данный вариант называется SVM с мягким зазором (soft-margin SVM).

Альтернативный подход

В качестве альтернативного подхода к решению проблемы линейной неразделимости классов, мы можем попытаться перейти в новое пространство H (*спрямляющее*) более высокой размерности с помощью некоторого преобразования $\psi(\vec{x})$, в котором обучающая выборка будет линейно разделима.

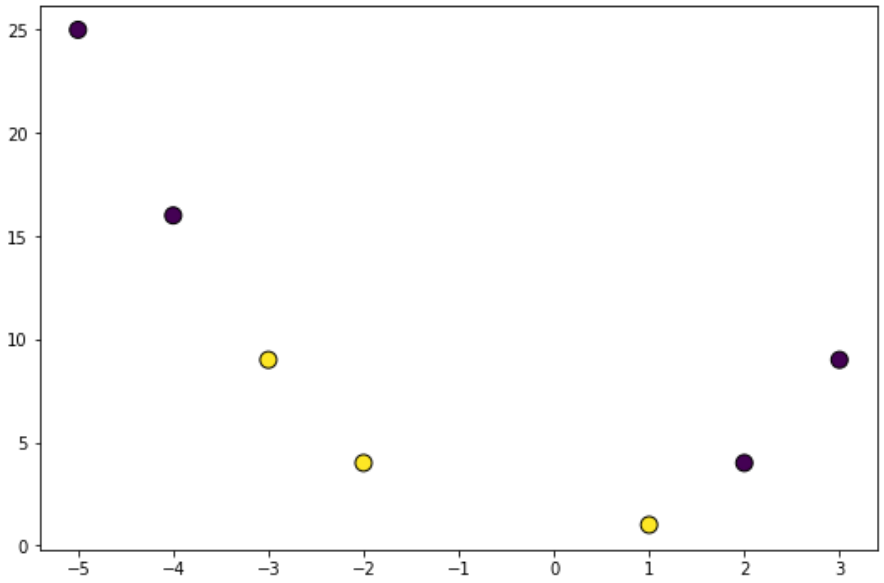
При этом, алгоритм построения SVM не изменится, просто везде скалярное произведение $\langle \vec{x}, \vec{x}' \rangle$ будет заменено на $\langle \psi(\vec{x}), \psi(\vec{x}') \rangle$.

In [17]:
```
x1 = np.array([-5, -4, -3, -2, 1, 2, 3])
y = np.array([-1, -1, 1, 1, 1, -1, -1])
plt.scatter(x1, np.zeros(x1.shape), c=y, edgecolor="k", s=100);
```



Перейдем в новое пространство, такое, что $\psi(\vec{x}) = (\vec{x}, \vec{x}^2)$

In [18]:
```
x2 = x1 * x1
plt.scatter(x1, x2, c=y, edgecolor="k", s=100);
```



Ядро - это функция $K : \mathrm{X} \times \mathrm{X} \to R$, которая может быть представима в виде $K(x, x') = \langle \psi(\vec{x}), \psi(\vec{x}') \rangle$ при некотором отображении $\psi : \mathrm{X} \to H$, где H - пространство, в котором определено скалярное произведение.

Функция $K(\vec{x}, \vec{x}')$ называется ядром, если

1. Она симметрична: $K(\vec{x}, \vec{x}') = K(\vec{x}', \vec{x})$
2. Положительно определена: $\int \int K(\vec{x}, \vec{x}') g(\vec{x}) g(\vec{x}') d\vec{x} d\vec{x}'$

Kernel Trick

Постановка задачи SVM и сам алгоритм классификации зависят только от скалярного произведения прецедентов, что позволяет нам формально заменить скалярное произведение некой ядерной функцией $K(\vec{x}, \vec{x}')$.

Данная идея позволяет вместо поиска спрямляющего пространства, искать необходимое ядро.

Способы построения ядер

1. Любое скалярное произведение является ядром: $K(\vec{x}, \vec{x}') = \langle \vec{x}, \vec{x}' \rangle$
2. Константа является ядром: $K(\vec{x}, \vec{x}') = 1$
3. Произведение ядер - ядро: $K(\vec{x}, \vec{x}') = K_1(\vec{x}, \vec{x}')K_2(\vec{x}, \vec{x}')$
4. Произведение отображений - ядро: $K(\vec{x}, \vec{x}') = \phi_1(\vec{x})\phi_2(\vec{x}'), \forall \phi : \mathrm{X} \to \mathbb{R}$
5. Линейная комбинация ядер - ядро: $K(\vec{x}, \vec{x}') = \sum_k \alpha_k K_k(\vec{x}, \vec{x}'), \forall \alpha_k \geq 0$

6. Композиция ядра и отображения - ядро: $K(\vec{x}, \vec{x}') = K_0(\phi(\vec{x}), \phi(\vec{x}'))$
7. Интегральное скалярное произведение: $\int S(\vec{x}, \vec{z})S(\vec{x}', \vec{z})dz$, для любой симмтрично интегрируемой функции
8. Функция вида $K(\vec{x}, \vec{x}') = k(\vec{x} - \vec{x}')$ является ядром, если Фурье-образ $F[k](\omega) = (2\pi)^{n/2} \int e^{-i\langle \vec{\omega}, \vec{x} \rangle} k(\vec{x}) d\vec{x}$ не отрицателен.
9. Степенной ряд по K с неотрицательными коэффициентами также ядро (например e^{-z}).
10. и т.д.

Примеры ядер

Рассмотрим простейшее ядро

$$K(\vec{u}, \vec{v}) = \langle \vec{u}, \vec{v} \rangle^2$$

где

$$\vec{u} = (u_1, u_2)^T,$$
$$\vec{v} = (v_1, v_2)^T,$$

$$K(\vec{u}, \vec{v}) = \langle \vec{u}, \vec{v} \rangle^2 =$$
$$= (u_1 v_1 + u_2 v_2)^2 = u_1^2 v_1^2 + 2u_1 u_2 v_1 v_2 + u_2^2 v_2^2 =$$
$$= \left\langle (u_1^2, u_2^2, \sqrt{2}u_1 u_2)^T, (v_1^2, v_2^2, \sqrt{2}v_1 v_2)^T \right\rangle$$

То есть наше ядро представимо в виде скалярного произведения в пространстве $H = \mathbb{R}^3$ с преобразованием $\psi : (u_1, u_2)^T \mapsto (u_1^2, u_2^2, \sqrt{2}u_1 u_2)^T$. При этом гиперповерхности в H соответсвует квадратичная поверхность в X.

Более сложный пример (полиномиальное ядро)

$$K(\vec{u}, \vec{v}) = \left(\langle \vec{u}, \vec{v} \rangle + 1\right)^d$$

Радиальная базисная функция (radial basis function, RBF)

$$K(\vec{u}, \vec{v}) = \exp\left(-\gamma \|\vec{u} - \vec{v}\|^2\right)$$
$$\gamma > 0$$

Почитать

- Support Vector Machines: A Simple Tutorial (https://svmtutorial.online/download.php?file=SVM_tutorial.pdf)
- К. В. Воронцов, Метод опорных векторов (http://www.machinelearning.ru/wiki/images/a/a0/Voron-ML-Lin-SVM.pdf)
- Kevin P. Murphy Machine Learning_ A Probabilistic Perspective (https://mitpress.mit.edu/books/machine-learning-1)

sklearn

В sklearn данный подход реализован для задач

- классификации: **SVC**, **LinearSVC**
- регрессии: **SVR**, **LinearSVR**

In [15]:
```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, PolynomialFeatures

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('svd', SVC(kernel="linear", max_iter=2000))])

pipe.fit(X_train, y_train)
predict = pipe.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.34453333333333336
AMS = 0.559029720888705
```

In [16]:
```
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('svd', SVC(kernel="poly", max_iter=2000))])

pipe.fit(X_train, y_train)
predict = pipe.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.3432
AMS = 0.5906548470665631
```

In [17]:
```
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('svd', SVC(kernel="rbf", max_iter=2000))])

pipe.fit(X_train, y_train)
predict = pipe.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.6584666666666666
AMS = 0.4463450930046813
```

Плюсы SVM

- как задача оптимизации - имеет одно единственное решение (если K - ядро)

- эффективная, уверенная классификация при правильном выборе ядра
- позволяет строить довольно сложные разделяющие поверхности
- может работать с довольно большими выборками

Минусы SVM

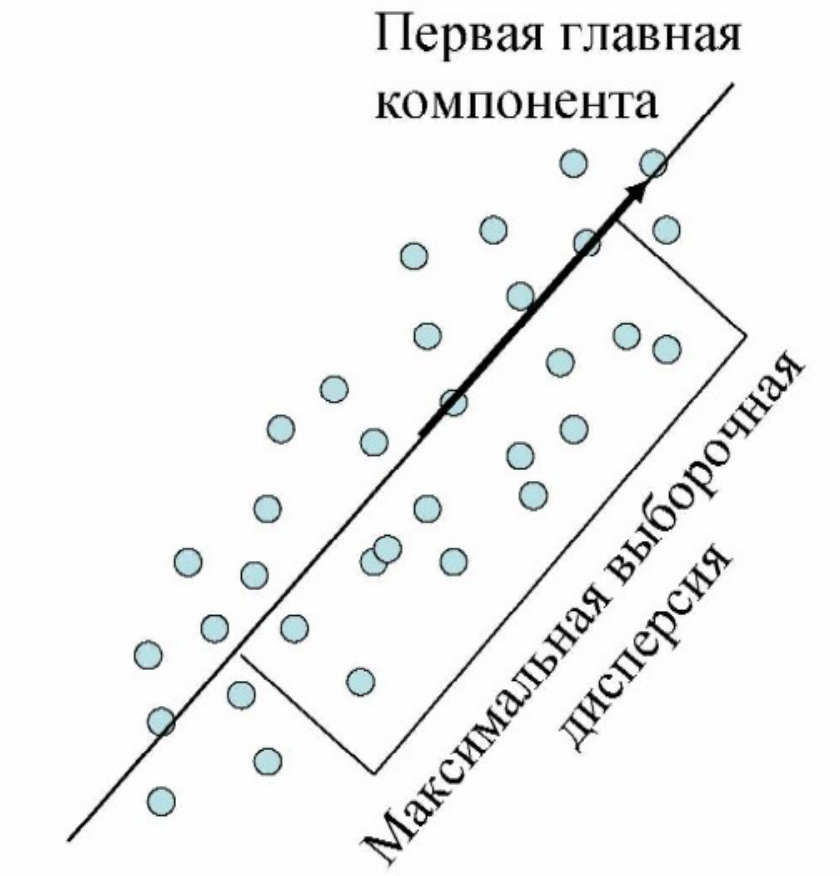
- не устойчив по отношению к шуму (выбросы по возможности необходимо убирать)
- нет общих методов построения спрямляющих пространств и ядер
- если нет полной линейной разделимости, необходимо подбирать параметр C

10. Метод главных компонент

Метод главных компонент (Principal component analysis)

Порой у нас может быть довольно много признаков, при этом какие-то признаки зависят от других признаков, что, следовательно, делает их лишними и не информативными. Хотелось бы найти такое преобразование признаков, чтобы в новом пространстве они были независимы, а каждый новый признак отражал наибольшее количество информации.

Попробуем этого добиться следующим образом: построим гиперплоскость таким образом, чтобы дисперсия расстояний до этой гиперплоскости была максимальна.



В качестве нулевого шага, выравняем наши признаки, вычтя из них среднее значение. Это нам даст новые признаки, у которых среднее значение даст 0.

После этого первую главную компоненту определить достаточно просто

$$S^2 = \frac{1}{N}\sum_n \langle \vec{\omega}_1, \vec{x}_n \rangle^2 = \frac{1}{N}\vec{\omega}_1^T \mathbf{X}^T \mathbf{X} \vec{\omega}_1 \to \max_{\vec{\omega}_1}$$

Добавим при этом требование ортонормированости вектора нормали

$$\langle \vec{\omega}_1, \vec{\omega}_1 \rangle = 1$$

Само решение ищется с помощью множителей Лагранжа

$$L = \vec{\omega}_1^T \mathbf{X}^T \mathbf{X} \vec{\omega}_1 - \lambda \vec{\omega}^T \vec{\omega}$$
$$\frac{\partial L}{\partial \vec{\omega}_1} = 2\mathbf{X}^T \mathbf{X} \vec{\omega}_1 - 2\lambda \vec{\omega}_1$$
$$\mathbf{X}^T \mathbf{X} \vec{\omega}_1 = \lambda \vec{\omega}_1$$

Для тех, кто знаком с линейной алгеброй, может увидеть здесь, что λ должно быть собственным значением матрицы $\mathbf{X}^T\mathbf{X}$, а $\vec{\omega}_1$ одним из собственных векторой этой матрицы.

Посмотрим на дисперсию

$$S^2 = \frac{1}{N}\vec{\omega}_1^T \mathbf{X}^T \mathbf{X} \vec{\omega}_1 = \frac{1}{N}\vec{\omega}_1^T \lambda \vec{\omega}_1 = \frac{\lambda}{N}$$

То есть максимум нашей дисперсии равен максимальному из возможных собственных значений.

Отлично, мы нашли плоскость, которая определяет первую компоненту. Таким образом, первая компонента будет:

$$c_n^1 = \langle \vec{\omega}_1, \vec{x}_n \rangle = \vec{x}_n^T \vec{\omega}_1$$

Или в старых координатах

$$\vec{x}_n^1 = \langle \vec{\omega}_1, \vec{x}_n \rangle \vec{\omega}_1 = c_n^1 \vec{\omega}_1$$

Как искать остальные?

Дальше мы будет искать следующую гиперплоскость, но такую, чтобы она была ортогональная первой. Это будет эквивалентно тому, что мы спроецируем все точки пространства на нашу первую гиперплоскость и в рамках координат этой гиперплоскости будем искать новую гиперплоскость размерностью на 1 меньше.

Формульно, мы вычитаем из наших точек наши "координаты" (проецируем)

$$\hat{x}_n^{(2)} = \vec{x} - \langle \vec{\omega}_1, \vec{x}_n \rangle \vec{\omega}_1$$
$$\hat{\mathbf{X}}_2 = \mathbf{X} - \mathbf{X}\vec{\omega}_1 \vec{\omega}_1^T$$

Затем среди этих точек ищем новую гиперповерхность, что максимизирует дисперсию

$$\frac{1}{N}\sum_n \langle \hat{x}_n^2, \vec{\omega}_2 \rangle = \frac{1}{N}\vec{\omega}_2^T \hat{\mathbf{X}}_2^T \hat{\mathbf{X}}_2 \vec{\omega}_1$$

Здесь также нужно добавить условие ортонормированности, дополнительно мы потребуем ортогональность предыдущему вектору

$$\langle \vec{\omega}_2, \vec{\omega}_2 \rangle = 1$$
$$\langle \vec{\omega}_1, \vec{\omega}_2 \rangle = 0$$

Тут мы должны присмотреться к $\hat{\mathbf{X}}_2\vec{\omega}_2$ и дополнительно воспользоваться условием ортогональности

$$\hat{\mathbf{X}}_2\vec{\omega}_2 = \mathbf{X}\vec{\omega}_2 - \mathbf{X}\vec{\omega}_1\vec{\omega}_1^T\vec{\omega}_2 = \mathbf{X}\vec{\omega}_2$$

Что приводит нас к тому же ответу

$$\mathbf{X}^T\mathbf{X}\vec{\omega}_2 = \lambda\vec{\omega}_2$$

Максимум мы получим тогда, когда $\vec{\omega}_2$ является собственным вектором. При этом, максимальное собственное значение мы взять не можем, оно запрещено условием ортогональности. Значим берем второе по величине.

Можно проделать ту же операцию и для третьей компоненты, мы получим тот же результат. Таким образом, вектора нормалей определяющих гиперплоскости для главных компонент являются собственными векторами матрицы $\mathbf{X}^T\mathbf{X}$.

Сингулярное разложение

Сингулярным разложением любой прямоугольной матрицы $\mathbf{X}$ называют

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

где $\mathbf{U}$ - унитарная матрица собственных векторов для $\mathbf{X}\mathbf{X}^T$; $\mathbf{V}$ - это унитарная матрица собственных векторов для $\mathbf{X}^T\mathbf{X}$;

- $\mathbf{U}$ - унитарная матрица, состоящая из левых сингулярных векторов, которые являются собственными векторами матрицы $\mathbf{X}\mathbf{X}^T$;
- $\mathbf{V}$ - унитарная матрица, состоящая из правых сингулярных векторов, которые являются собственными векторами матрицы $\mathbf{X}^T\mathbf{X}$, в нашем случае - это именно те вектора нормалей, которые мы искали ранее;
- $\mathbf{\Sigma}$ - матрица, у которой только диагональные элементы не равны нулю и отсортированы, эти числа называют сингулярными.

Отсюда

$$\mathbf{X}^T\mathbf{X} = \mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T$$

Таким образом, сингулярное разложение матрицы $\mathbf{X}$ тесным образом связано с методом главных компоненты. При этом значение координат в главных компонентах

$$\mathbf{T} = \mathbf{X}\mathbf{V} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{V} = \mathbf{U}\mathbf{\Sigma}$$

Уменьшение размерности

Метод главных компонент можно использовать для уменьшения размерности. Согласно основной идее метода, наши компоненты отсортированы по мере важности, при чем степень важности определяется размером собственного значения. Следовательно, мы можем попробовать отобрать только L самых важных компонент, обрезав $\mathbf{V}$, $\mathbf{\Sigma}$, $\mathbf{U}$ до нужной

размерности.

$$\mathbf{T}_L = \mathbf{U}_L \mathbf{\Sigma}_L$$

Также, нам ничего не мешает восстановить приближенное значение $\mathbf{X}_L$ из ограниченного числа главных компонент

$$\mathbf{X}_L = \mathbf{U}_L \mathbf{\Sigma}_L \mathbf{V}_L^T$$

В ряде случаев это позволяет убрать шум и выбросы в данных

Sklearn

Уже содержит реализацию метода главных компонент: **PCA**

In [18]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
pca.fit(X_train)
pca.transform(X_train)
```

Out[18]:
```
array([[-0.43321068,  0.41776716],
       [-1.48379959,  0.07298332],
       [-0.36820476, -0.13614051],
       ...,
       [-0.11171176, -0.60002523],
       [-1.37751383,  0.54305056],
       [ 1.37539432,  0.24801778]])
```

11. Градиентный бустинг

Взвешенное голосование

для регрессии

$$R(f_1(\vec{x}), \ldots, f_M(\vec{x})) = \sum_{m=1}^{M} \omega_m f_m(\vec{x})$$

для бинарной классификации с классами $y = \{-1, +1\}$

$$R(f_1(\vec{x}), \ldots, f_M(\vec{x})) = \text{sign}\left[\sum_{m=1}^{M} \omega_m f_m(\vec{x})\right]$$

Мы уже рассматривали вариант, когда у нас модели обучались заранее, а потом на основе уже обученных моделей мы искали коэффициенты взвешенного голосования.

А что если мы не будет обучать модели заранее, а будем определять параметры нашей модели наравне с весами?

$$R(f_1(\vec{x}), \ldots, f_M(\vec{x})) = \text{sign}\left[\sum_{m=1}^{M} \omega_m f_m(\vec{x})\right]$$

Основная идея

Будем рассматривать задачу бинарной классификации, где $y = \{-1, +1\}$.

Для упрощения нахождение оптимальной комбинации моделей, мы будет искать их итеративно, добавляя модели по одной (жадный алгоритм).

$$R_m = \sum_{i=1}^{m} \alpha_i f_i(\vec{x}) = R_{m-1} + \alpha_m f_m(\vec{x})$$

Теперь нам нужно найти самые оптимальные α_m и f_m минимизируя функционал качества Q_m, который просто является количеством неправильных ответов. Для классификации - это сумма "ступенек".

$$Q_m = \sum_n \left[\text{sign}\left(R_m(\vec{x}_n)\right) \neq y_n\right]$$

Для упрощения задачи, мы приближаем одну ступеньку непрерывно дифференцируемой оценкой сверху.

Если взять в качестве оценки

$$L(f, \vec{x}, y) = \exp(-y f(\vec{x}))$$

то просто получим **AdaBoost**

AdaBoost

[Adaptive Boosting (https://mbernste.github.io/files/notes/AdaBoost.pdf)](https://mbernste.github.io/files/notes/AdaBoost.pdf) - один из первых и простейших алгоритмов бустинга.

- Пусть есть набор прецендентов X размером N и набор меток класса Y.
- Инициализируем вектор весов наших прецендентов $\omega_1(n) = \frac{1}{N}, i = 1, \dots, N$

Для каждого шага

1. Находим классификатор, который минимизирует взвешенную ошибку классификации
$$\varepsilon_m = \sum_n \omega_m(n)[y_n \neq f_m(\vec{x}_n)]$$
2. Выбираем α_m
$$\alpha_m = \frac{1}{2}\ln\frac{1-\varepsilon_m}{\varepsilon_m}$$
3. Обновляем веса
$$\omega_{m+1}(n) = \frac{\omega_m(n)\exp(-\alpha_m y_n f_m(\vec{x}_n))}{\Omega_m}$$

где Ω_m - это константа нормализация, такая что $\sum \omega(n) = 1$

Результирующий классификатор

$$R(\vec{x}) = \text{sign}\sum_m \alpha_m f_m(\vec{x})$$

Небольшие проблемы

Не все методы поддерживают обучение со взвешенными данным, в этом случае, вектор весов ω_m можно трактовать как вероятность достать данный прецедент при формировании бутстрап-

sklearn

В библиотеке **sklearn** уже реализован данный алгоритм и он представлен двумя классам: **AdaBoostClassifier** и **AdaBoostRegressor**.

Их основными гиперпараметрами являются: класс слабой модели и количество моделей.

In [21]:
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

ada = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1), 15)
ada.fit(X_train, y_train)
predict = svm.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.34282666666666667
AMS = 0.5853228882886334
```

Плюсы AdaBoost

- данный алгоритм очень прост для реализации
- эффективен с точки зрения вычислительной сложности
- позволяет решать достаточно сложные задачи с помощью слабых моделей
- фактически только два параметра настройки: класс моделей и их количество
- обеспечивает высокую точность прогнозирования
- прост для модификации

Минусы AdaBoost

- использование сложных или сильных моделей приводит к переобучению
- чувствительность к выбросам (им достаются самые высокие значения весов), что приводит к переобучению
- практически не поддается интерпретации
- желательно иметь достаточно большую выборку, иначе может привести к переобучению

Градиентный бустинг

[Gradient Boosting Machine (GBM) (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3885826/)](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3885826/) - обобщенный и стандартизированный подход.

Пусть у нас есть набор прецедентов $\mathbf{X}$ и набор целевых исходов $\vec{y}$, для которых мы будем восстанавливать зависимость в виде $y = f(\vec{x})$.

Для того, чтобы понять какое из наших приближений $\bar{f}$ лучше, мы будет его выбирать так, чтобы минимизировать функционал потерь

$$\bar{f} = \arg\min_{f} \sum_{n} L(y_n, f(\vec{x}_n))$$

Искать решение итогового классификатора будем в виде

$$R(\vec{x}) = \sum_m \alpha_m f_m(\vec{x})$$

при этом искать (α_m, f_m) мы будем итеративно, по одному, каждый раз решая задачу

$$(\alpha_m, f_m) = \arg\min_{\alpha, f} \sum L(y_n, R_{m-1}(\vec{x}_n) + \alpha f(\vec{x}_n))$$

$$(\alpha_m, f_m) = \arg\min_{\alpha, f} \sum_n L(y_n, R_{m-1}(\vec{x}_n) + \alpha f(\vec{x}_n))$$

Самый простой вариант: мы можем выбрать конкретную функцию потерь и с ней работать.

Если взять квадратичную ошибку (L2boosting)

$$L = \frac{1}{2}(y - f(\vec{x}))^2$$

$$(\alpha_m, f_m) = \arg\min_{\alpha, f} \sum_n (y_n - R_{m-1}(\vec{x}_n) - \alpha f(\vec{x}_n))^2$$

$$(\alpha_m, f_m) = \arg\min_{\alpha, f} \sum_n (r_{nm} - \alpha f(\vec{x}_n))^2$$

$$r_{nm} = y_n - R_{m-1}(\vec{x}_n)$$

Стало намного легче искать, но что если нам теперь хочется сменить функцию потерь?

Попробуем теперь обобщить. Будем искать нашу модель, минимизируя

$$\bar{R} = \arg\min_R \sum_n L(y_n, R(\vec{x}_n))$$

Тут если очень присмотреться к записи

$$R(\vec{x}) = \sum_m \alpha_m f_m(\vec{x})$$

то можно заметить, что это что-то вроде перемещения от точки к точке в функциональном пространстве. Нечто похожее мы можем видеть, когда ищем минимум функциоанала с помощью обратного градиента.

В обычном пространстве минимизировать мы уже умеем, но что нам мешает сделать тоже самое в функциональном?

$$g_{nm} = \left[\frac{\partial L(y_n, f(\vec{x}_n))}{\partial f(\vec{x}_n)}\right]_{f=R_{m-1}}$$

Отсюда новый набор наших "параметров"

$$g_{nm} = \left[\frac{\partial L(y_n, f(\vec{x}_n))}{\partial f(\vec{x}_n)}\right]_{f=R_{m-1}}$$

$$R_m(\vec{x}_n) = R_{m-1}(\vec{x}_n) - \alpha_m g_{nm}$$

$$\alpha_m = \arg\min_\alpha \sum_n L(R_{m-1}(\vec{x}_n) - \alpha g_{nm})$$

И это нам не очень помогает, т.к. это оптимизирует какую-то абстрактную функцию R зависящую

от неизвестной нам функции f

Однако, мы можем потребовать, чтобы слабая модель апроксимировала антиградиент

$$\theta_m = \arg\min_{\theta} \sum_n (-g_{nm} - f_m(\vec{x}_n, \theta))^2$$

при этом не обязательно использовать минимизацию квадратичной ошибки.

Теперь мы готовы описать обобщенный алгоритм

$$r_{nm} = -g_{nm} = -\frac{\partial L(y_n, f(\vec{x}_n))}{\partial f(\vec{x}_n)}\Big|_{f=R_{m-1}}$$

$$\theta_m = \arg\min \sum_n (r_{nm} - f_m(\vec{x}_i, \theta))^2$$

$$\alpha_m = \arg\min \sum_n L(y_n, R_{m-1}(\vec{x}_i) + \alpha f_m(\vec{x}_i, \theta_m))$$

- Пусть есть набор прецендентов $\mathbf{X}$ размером N и набор меток класса $\vec{y}$.
- Первой моделью выбираем константный прогноз

$$f_0(\vec{x}) = \arg\min_{\alpha_0} \sum_n L(\alpha_0, y_n)$$

Для каждого шага

1. Вычисляем остатки предыдущей композиции r_{nm}
2. Обучаем базовую модель $f_m(\vec{x})$ на остатках r_{nm}
3. Вычисляем α_m как решение простой оптимизационной задачи
4. Добавить новый классификатор в ансамбль $R_m = R_{m-1} + \alpha_m f_m$

Итоговая модель

$$R(\vec{x}) = \alpha_0 + \sum_{m=1}^{M} \alpha_m f_m(\vec{x})$$

Регрессия

- квадрат ошибки(L_2 loss)

$$L(y, f) = \frac{1}{2}(y - f)^2$$

- абсолютная ошибка(L_1 loss)

$$L(y, f) = |y - f|$$

- Huber loss

$$L(y, f) = \begin{cases} \frac{1}{2}(y - f)^2, & |y - f| \le \delta \\ \delta\left(|y - f| - \frac{\delta}{2}\right), & |y - f| > \delta \end{cases}$$

- Quantile loss (L_q loss)

$$L(y, f) = \begin{cases} (1 - \alpha)|y - f|, & y - f \le 0 \\ \alpha|y - f|, & y - f > 0 \end{cases}$$

Классификация

- логистическая функция потерь

$$L(y, f) = \log(1 + \exp(-2yf)), y \in \{-1, 1\}$$

- экспоненциальная функция потерь (Adaboost)

$$L(y, f) = \exp(-yf), y \in \{-1, 1\}$$

Стохастический градиентный бустинг

Можно расчитывать новые модели опираясь не на всю выборку, а на случайную подвыборку фиксированного размера.

Темп обучения

Для обеспечения устойчивости решения, можно добавить множитель к α_m, называемый темпом обучения γ(learning rate). При низком значении γ мета-модель обучается медленее, но результат обычно становится лучше.

$$R_m(\vec{x}) = R_{m-1} + \gamma\alpha_m f_m(\vec{x})$$

Регуляризация

Помимо темпа обучения, нам ничего не мешает добавлять регуляризующие члены к нашей функции потерь

$$L(y, f(\vec{x})) + \Omega(f)$$

sklearn

В библиотеке **sklearn** данный алгоритм реализован с помощью деревьев решений в двух классах **GradientBoostingClassifier** и **GradientBoostingRegressor** с довольно большим количеством гиперпараметров.

In [22]:
```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import load_digits

gbf = GradientBoostingClassifier()
gbf.fit(X_train, y_train)
predict = gbf.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.7018933333333334
AMS = 0.9543806652360566
```

Плюсы GBM

- один из наиболее мощных алгоритмов распознования
- если использовать в качестве базовой модели решающее дерево, то не очень чувствителен к масштабу данных
- множество способов улучшить алгоритм
- общий подход к решению задачи
- подходит по регрессию, классификацию и ранжирование

- можно использовать произвольную функцию потерь (зависит от поставленной задачи)
- есть очень хорошие готовые решения

Минусы

- довольно трудоемкий алгоритм, если использовать много базовых моделей
- в "чистой" реализации очень склонен к переобучению
- не подходит для использования со сложными или сильными моделями
- иногда довольно сложная настройка
- неинтерпретируем

Готовые решения

Одной из самых простых и быстрых моделей является - дерево решений. На основе деревьев реализованы очень мощные библиотеки:

- XGBoost (https://xgboost.ai/)
- LightGBM (https://github.com/Microsoft/LightGBM)
- CatBoost (https://catboost.ai/)

Отдельно стоит отметить H2O (https://www.h2o.ai/)

In [26]:
```
import xgboost as xgb

clf = xgb.XGBClassifier(max_depth=3, learning_rate=0.2, n_estimators=100)

clf.fit(X_train, y_train)
predict = clf.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.7028133333333333
AMS = 0.9539233231473808
```

In [27]:
```
import lightgbm as lgbm

clf = lgbm.LGBMClassifier(max_depth=3, learning_rate=0.2, n_estimators=100)

clf.fit(X_train, y_train)
predict = clf.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.7035333333333333
AMS = 0.9584567638038991
```

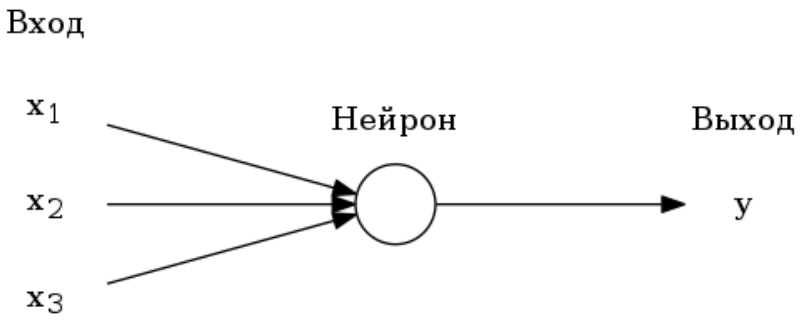
In [28]:
```
import catboost as cbgm

clf = cbgm.CatBoostClassifier(max_depth=3, n_estimators=100,   verbose=False)

clf.fit(X_train, y_train)
predict = clf.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.70372
AMS = 0.9571437002536894
```

12. Нейронные сети

Нейрон

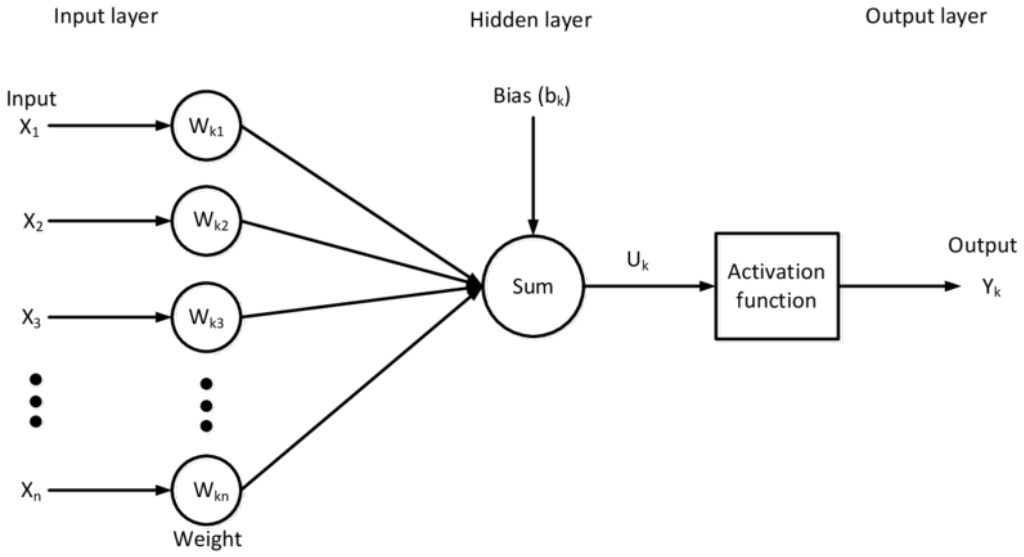
Простейшую модель нейрона можно представить как на картинке ниже



У нас есть некий набор входных значений, которые поступают в нейрон. Он их как-то обрабатывает, затем выдает результат.

Модель нейрона

Нейрон можно описать довольно большим количеством способов. Мы же будет работать с частным случаем, который используется повсеместно, который описывается изображением ниже



Или тоже самое, но формульно

$$z = b + \sum_{n=1}^{N} \omega_n x_n = b + \vec{\omega}^T \vec{x}$$

если ввести $x_0 = 1$ и $\omega_0 = b$, то можно переписать в чуть более удобной форме

$$z = \sum_{n=0}^{N} \omega_n x_n = \vec{\omega}^T \vec{x}$$

$$y = f(z)$$

где $f(z)$ - функция активации

$$z = \sum_{n=0}^{N} \omega_n x_n = \vec{\omega}^T \vec{x}$$

$$y = f(z)$$

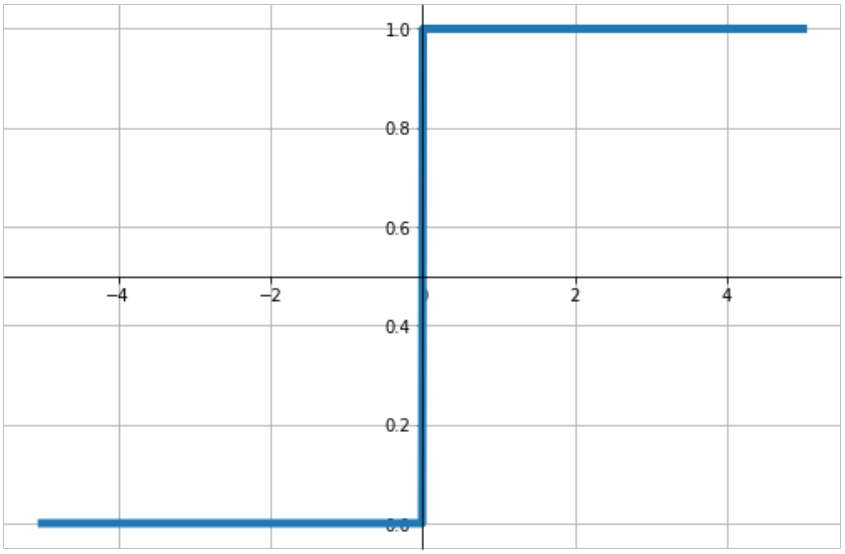
Функции активации

In [29]:
```
x = np.linspace(-5, 5, 1000)

def plot(x, y):
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.spines['left'].set_position('center')
    ax.spines['bottom'].set_position('center')
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.yaxis.set_ticks_position('left')
    plt.grid()
    plt.plot(x, y, lw=5)
    plt.show()
```

Одной из первых функций активации была функция Хевисайда

$$f(x) = \begin{cases} 0, \text{ для } x < 0 \\ 1, \text{ для } x \geq 0 \end{cases}$$

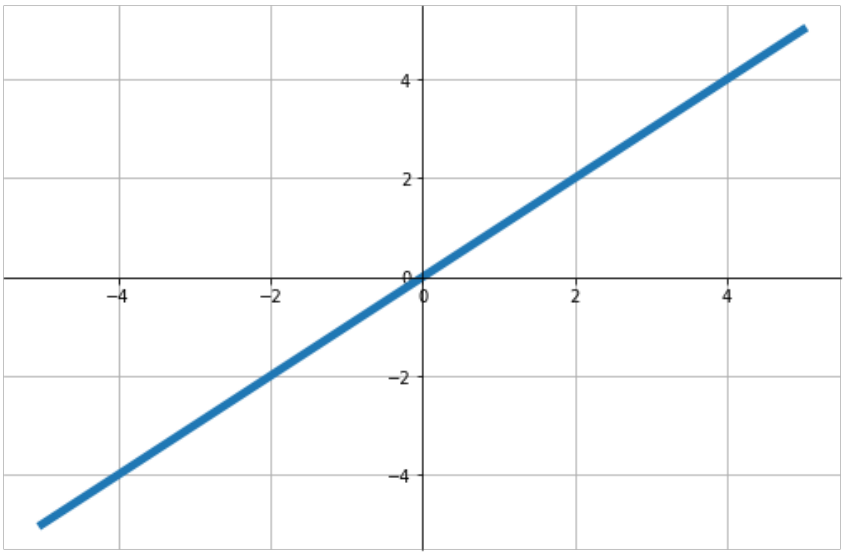
In [30]:
```
plot(x, np.heaviside(x, 1))
```



Следующая простейшая функция активации, которая может придти нам в голову - это обычная линейная функция

$$f(x) = x$$

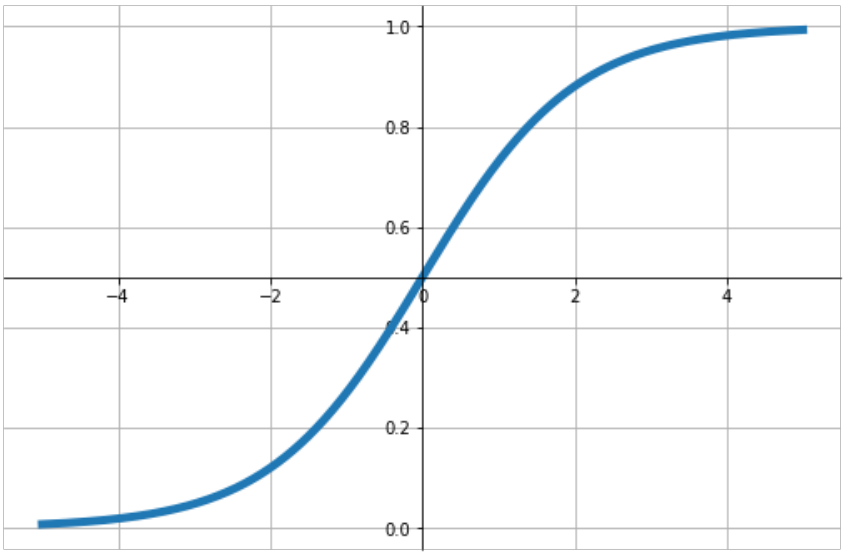
In [31]:
```
plot(x, x)
```



Функция Хевисайда не очень удобная для задач оптимизации - у неё рвется производная в нуле. Вместо неё мы можем попробовать взять ее приближение, одним из вариантов - это сигмоида

$$f(x) = \frac{1}{1 + e^{-\alpha x}}$$

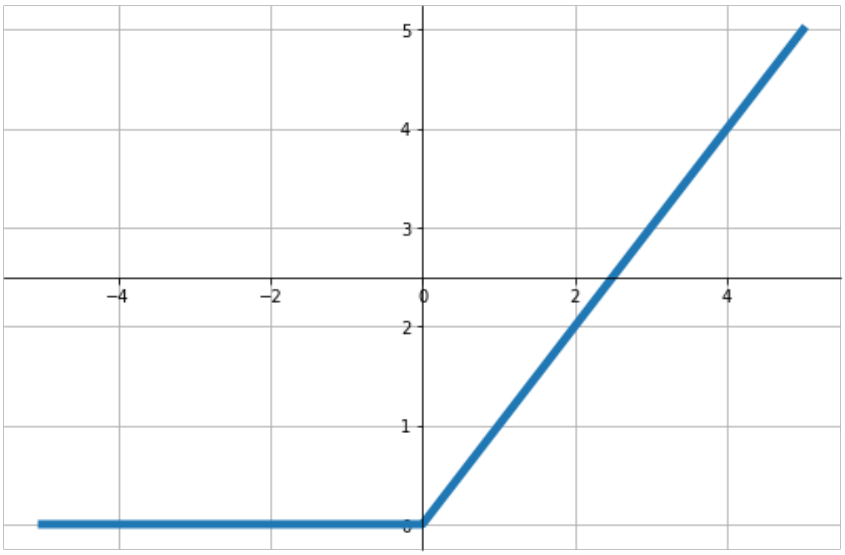
In [32]:
```
plot(x, 1/(1 + np.exp(-x)))
```



В настоящее время, весьма хорошо себя показала функция активации ReLU (rectified linear unit)

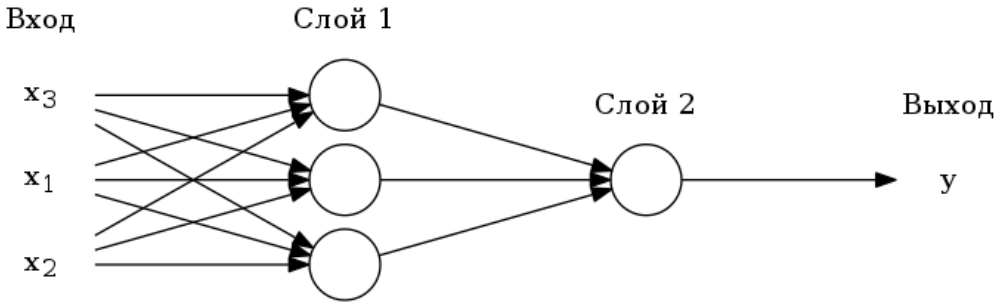
$$f(x) = \max(0, x)$$

In [33]:
```
plot(x, np.maximum(x, 0))
```



Нейронные сети

Нам ничего не запрещает объединить множество нейронов друг с другом. Такие объединения будем называть нейронными сетями.



Многослойная сеть

Введем ряд обозначений:

- $n = 1, \dots, N$ - номер слоя
- $j = 0, \dots, J_n$ - нумерация нейронов внутри слоя, при этом $(j = 0)$ будет соответствовать фиктивному нейрону, который всегда выдает 1.
- $x_j, \vec{x}$ - входные данные (признаки)
- $y_j^{(n)}, \vec{y}^{(n)}$ - вывод j-го нейрона в n-ом слое. $y_j^{(0)} = x_j$.
- $f_j^{(n)}$ - функция активации j-го нейрона в n-ом слое, для простоты будем считать, что для всех нейронов внутри слоя они одинаковы
- $z_j^{(n)}$ - результат суммирования j-го нейрона в n-ом слое

$$z_j^{(n)} = \sum_{i=0}^{I_n} \omega_{ij}^{(n)} y_i^{(n-1)} = (\vec{\omega}_j^{(n)})^T \vec{y}^{(n-1)}$$

$$y_j^{(n)} = f_j^{(n)}(z_j^{(n)})$$

А если ввести матрицу $\mathbf{W}^{(n)}$, в строках которой написаны веса нейронов для слоя (n), то можно

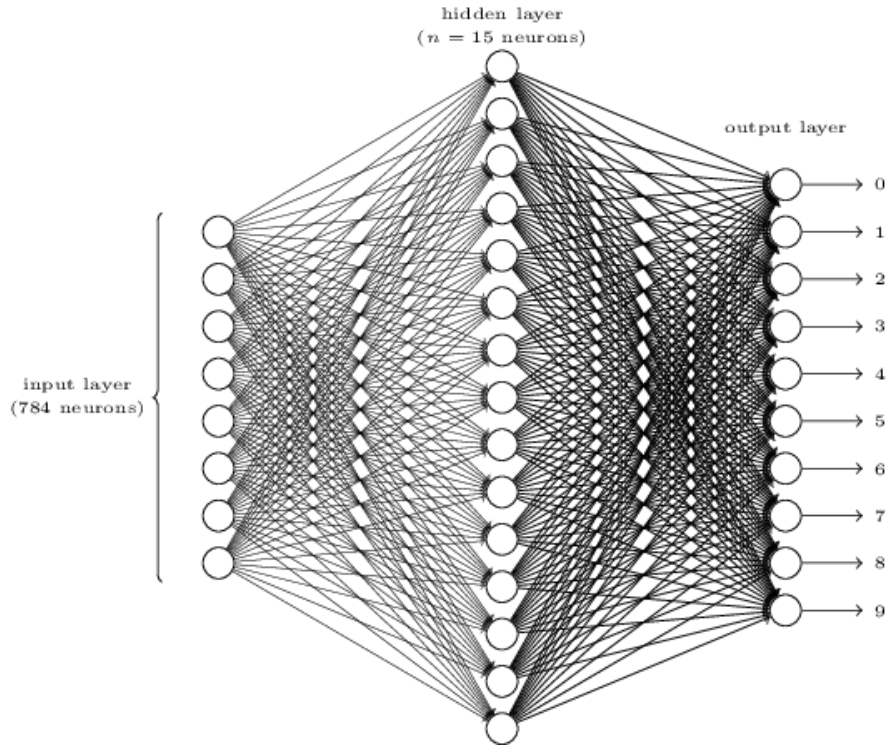
записать эти формулы даже проще

$$\mathbf{W}^{(n)} = (\vec{\omega}_1^{(n)}, \vec{\omega}_2^{(n)}, \dots, \vec{\omega}_J^{(n)})^T$$

$$\vec{z}^{(n)} = \mathbf{W}^{(n)}\vec{y}^{(n-1)}$$

$$\vec{y}^{(n)} = f^{(n)}(\vec{z}^{(n)})$$

$$f^{(n)}(\vec{z}^{(n)}) = \begin{bmatrix} f_1^{(n)}(z_1^{(n)}) \\ f_2^{(n)}(z_2^{(n)}) \\ \dots \\ f_J^{(n)}(z_J^{(n)}) \end{bmatrix}$$



При создании сети, как лучше всего инициализировать веса?

В общем случае, достаточно проинициализировать случайными значениями.

sklearn

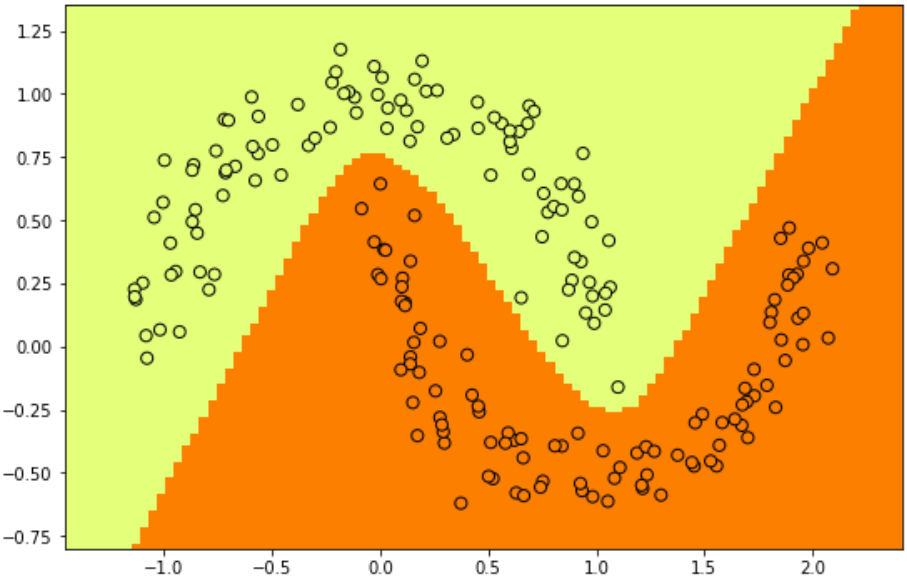
В sklearn уже реализована простейшая многослойная нейронная сеть с поддержкой регуляризации и различных видов функций активации

- MLPClassifier (https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html)
- MLPRegressor (https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html#sklearn.neural_network.MLPRegressor)

In [34]:
```
def plot(clf, X, y):
    x_max, x_min = X[:, 0].max(), X[:, 0].min()
    y_max, y_min = X[:, 1].max(), X[:, 1].min()

    dx = (x_max - x_min) * 0.1
    dy = (y_max - y_min) * 0.1

    xx, yy = np.meshgrid(np.linspace(x_min - dx, x_max + dx, 100),
```

```
                          np.linspace(y_min - dy, y_max + dy, 100))

    Z = np.zeros(xx.size)

    Z = clf.predict(np.vstack([xx.ravel(), yy.ravel()]).T).ravel()

    plt.pcolormesh(xx, yy, Z.reshape(xx.shape), cmap=plt.cm.Wistia)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.get_cmap("Wistia"),
                edgecolors='k', s=50)
```

In [63]:
```
from sklearn.datasets import make_blobs, make_moons
from sklearn.neural_network import MLPClassifier

#X, y = make_blobs(200, centers=2, random_state=15, center_box=(-2, 2))
X, y = make_moons(200, noise=0.1, random_state=13)

net = MLPClassifier([5,3], activation='logistic', learning_rate_init=0.1, tol=1
net.fit(X, y)
plot(net, X, y)
```



In [61]:
```
net = MLPClassifier([5, 2], activation='relu', learning_rate_init=0.1)
clf.fit(X_train, y_train)
predict = clf.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.70372
AMS = 0.9571437002536894
```

Обучение

Как же можно обучать нейронную сеть?

Необходимо завести функцию потерь, отображающую качество классификации и воспользоваться любым способом её минимизации.

Мы будем пользоваться градиентным спуском.

Градиентный спуск

Тут всё довольно просто, мы выбираем функцию потерь в виде

$$L = \frac{1}{M}\sum_{m=1}^{M}\mathcal{L}_m$$

Здесь $\mathcal{L}_m$ — функция потери для конкретного объекта

Находим производную по весам и просто движемся в обратную сторону от направления градиента

$$\tilde{\omega}_{ij}^{(n)} = \omega_{ij}^{(n)} - \frac{\partial L}{\partial \omega_{ij}^{(n)}} = \omega_{ij}^{(n)} - \frac{1}{M}\sum_{m}\frac{\partial \mathcal{L}_m}{\partial \omega_{ij}^{(n)}}$$

Стохастический градиентный спуск

Для довольно больших объемов данных порой удобнее использовать стохастический градиентный спуск, для которого один шаг оптимизации выглядит следующим образом

$$\tilde{\omega}_{ij}^{(n)} = \omega_{ij}^{(n)} - \frac{\partial \mathcal{L}_m}{\partial \omega_{ij}^{(n)}}$$

Функции потерь

Выбор функции потерь ограничен только нашим воображением и особенностями поставленной перед нами задачи.

- квадратичная ошибка (mse)

$$\mathcal{L} = \frac{1}{2}(y - \tilde{y})^2$$

- абсолютная ошибка (mae)

$$\mathcal{L} = |y - \tilde{y}|$$

- logloss

$$\mathcal{L} = -y \ln \tilde{y} - (1 - y)\ln(1 - \tilde{y})$$

Метод обратного распространения ошибки

В общем случае, искать градиент - это довольно дорогое занятие. У нас огромное количество весов и вычислять для них всех градиент от нашей функции ошибок весьма неудобно.

Выше мы видели, что стохастический градиентный спуск и обычный градиентный спуск отличаются только ссумирование. В обоих случаях берется одна и та же производная от функции потери. Поэтому для простоты рассмотрим функцию потери на одном объекте.

Пусть мы выбрали функцию потерь

$$\mathcal{L} = \frac{1}{2}(y - \tilde{y})^2$$

и функцию активации

$$f(z) = \left(\frac{1}{1 + e^{-z}}\right)$$

Теперь нам нужно лишь найти все производные

$$\frac{\partial \mathcal{L}}{\partial \omega_{ij}^{(n)}}$$

Рассмотрим обычное взятие производной сложной функции

$$\frac{\partial \mathcal{L}}{\partial \omega_{ij}^{(n)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(n)}} \frac{\partial z_j^{(n)}}{\partial \omega_{ij}^{(n)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(n)}} y_i^{(n-1)}$$

где

$$\frac{\partial z_j^{(n)}}{\partial \omega_{ij}^{(n)}} = \frac{\partial (\sum_i \omega_{ij}^{(n)} y_i^{(n-1)})}{\partial \omega_{ij}^{(n)}} = y_i^{(n-1)}$$

Обозначим

$$\delta_j^{(n)} = \frac{\partial \mathcal{L}}{\partial z_j^{(n)}}$$

$$\vec{\delta}^{(n)} = \frac{\partial L}{\partial \vec{z}^{(n)}}$$

Теперь можем записать производную немного проще

$$\frac{\partial \mathcal{L}}{\partial \vec{\omega}_j^{(n)}} = \delta_j^{(n)} \vec{y}^{(n-1)}$$

Или совсем просто

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(n)}} = \vec{\delta}^{(n)} (\vec{y}^{(n-1)})^T$$

$$\tilde{\mathbf{W}}^{(n)} = \mathbf{W}^{(n)} - \vec{\delta}^{(n)} (\vec{y}^{(n-1)})^T$$

Мы видим, что градиент зависит от выходных данных с предыдущего слоя. Уже значительно лучше, но нам всё ещё нужно найти $\vec{\delta}^{(n)}$

$$\delta_j^{(n)} = \frac{\partial \mathcal{L}}{\partial z_j^{(n)}} = \frac{\partial \mathcal{L}}{\partial y_j^{(n)}} \frac{\partial y^{(n)}}{\partial z_j^{(n)}} = \frac{\partial \mathcal{L}}{\partial y_j^{(n)}} f'(z_j^{(n)})$$

где

$$y = f(z)$$

Немного отойдем назад и вспомним, где же $y_j^{(n)}$ встречается в нашей задаче. Это немного сложно представить, но логично предположить. Внутри $\mathcal{L}$ есть $y_j^{(n+1)}$ (выход следующего слоя), которые являются $y_j^{(n)}$. Попробуем перейти от производной по $y_j^{(n)}$ к $y_j^{(n+1)}$

$$\delta_j^{(n)} = f'(z_j^{(n)}) \sum_{k=1}^{J_{n+1}} \left[\frac{\partial \mathcal{L}}{\partial y_k^{(n+1)}} \frac{\partial y_k^{(n+1)}}{\partial z_k^{(n+1)}} \frac{\partial z_k^{(n+1)}}{\partial y_j^{(n)}} \right] =$$

$$= f'(z_j^{(n)}) \sum_{k=1}^{J_{n+1}} \left[\frac{\partial \mathcal{L}}{\partial y_k^{(n+1)}} f'(z_k^{(n+1)}) \omega_{jk}^{(n+1)} \right]$$

$$\delta_j^{(n)} = \frac{\partial \mathcal{L}}{\partial y_j^{(n)}} f'(z_j^{(n)})$$

В итоге, выражение

$$\delta_j^{(n)} = f'(z_j^{(n)}) \sum_{k=1}^{J_{n+1}} \left[\frac{\partial L}{\partial y_k^{(n+1)}} f'(z_k^{(n+1)}) \omega_{jk}^{(n+1)} \right]$$

в наших новых обозначениях примет вид для δ

$$\delta_j^{(n)} = f'(z_j^{(n)}) \sum_{k=1}^{J_{n+1}} \left[\delta_k^{(n+1)} \omega_{jk}^{(n+1)} \right]$$

таким образом, мы можем расчитать все δ зная лишь $\delta^{(N)}$

Давайте мы его найдем

$$\delta_j^{(N)} = \frac{\partial \mathcal{L}}{\partial y_j^{(N)}} f'(z_j^{(N)}) =$$

$$= (y_j^{(N)} - y_j) y_j^{(N)} (1 - y_j^{(N)})$$

где

$$f'(z) = \left(\frac{1}{1 + e^{-z}} \right)' = \frac{e^{-z}}{(1 + e^{-z})^2} =$$

$$= \frac{1}{1 + e^{-z}} \left(1 - \frac{1}{1 + e^{-z}} \right) = f(z)(1 - f(z)) = y(1 - y)$$

Введем новое обозначение (произведение Адамара), такое что

$$(\mathbf{A} \circ \mathbf{B})_{i,j} = (A)_{i,j} \cdot (B)_{i,j}$$

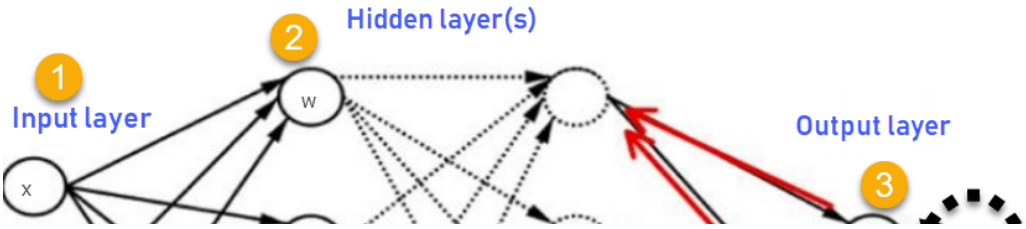
то есть мы просто по компонентно перемножаем элементы матриц

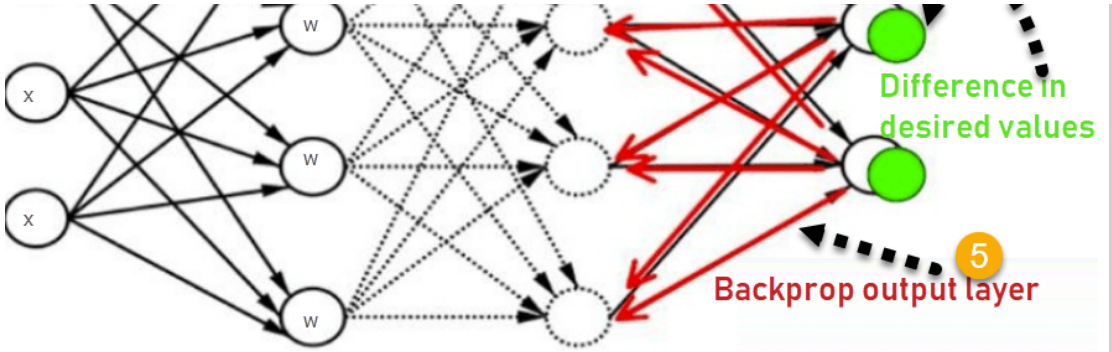
С этим обозначением наши выражения можно записать как

$$\vec{\delta}^{(N)} = \vec{y}^{(N)} \circ (\vec{1} - \vec{y}^{(N)}) \circ (\vec{y}^{(N)} - \vec{y})$$

$$\vec{\delta}^{(n)} = \vec{y}^{(N)} \circ (\vec{1} - \vec{y}^{(N)}) \circ ((\mathbf{W}^{(n+1)})^T \vec{\delta}^{(n+1)})$$

$$\tilde{\mathbf{W}}^{(n)} = \mathbf{W}^{(n)} - \vec{\delta}^{(n)} (\vec{y}^{(n-1)})^T$$





Batch

Вместо чистого стохастического градиентного спуска, мы можем набирать небольшой случайный набор данных (batch) и строить градиент, усредняя на нем.

Алгоритм

Теперь мы готовы описать алгоритм обратного распространения ошибки:

1. Проинициализировать сеть небольшими случайными значениями
2. Для входных параметров мы расчитываем значение выводов каждого слоя $\vec{y}^{(n)}$ (forward)
3. Рассчитываем ошибку $\vec{\delta}^{(N)}$ и затем от последнего слоя к первому расчитываем остальные $\vec{\delta}^{(n)}$ (backpropagation)
4. Корректируем веса $\mathbf{W}^{(n)}$
5. Перейти к шагу 2

Плюсы Backpropagation

- быстро вычисляется
- легко обобщается на любые функции активации, функции потерь и число слоев
- возможность online-обучения
- легко распараллеливается

Проблемы Backpropagation

- сходимость не всегда хорошая
- локальные минимумы
- взрыв градиента
- паралич сети
- переобучение

Softmax

В предыдущем случае всё получилось отлично, т.к. мы взяли специфическую функцию потерь и функцию активации, но что если у нас функция активации не ограничена?

Мы можем получить тогда произвольное значение целевого признака.

Нам бы хотелось получить на выходах нейронной сети нечто, что было бы очень похоже на вероятности

Идея в том, чтобы заменить последний слой нейронов, отвечающих за выход, специфическим softmax-слоем.

$$z_j^{(N)} = \sum_i \omega_{ij}^{(N)} y_i^{(N-1)}$$

$$y_j^{(N)} = \frac{e^{z_j^{(N)}}}{\sum_k e^{z_k^{(N)}}}$$

В этом случае

$$\sum_j y_j^{(N)} = 1$$

Регуляризация

При обучении, у нас нет никаких ограничений на веса, что может приводить к печальным последствиям. Неплохим решением является регуляризация весов.

$$L = \frac{1}{M}\sum_m \mathcal{L}_m + \lambda \frac{1}{M}\sum_m \mathcal{R}_m$$

L2-регуляризация

Самый простой вариант регуляризации - это просто сумма квадратов всех весов модели

$$\mathcal{R} = \sum_n \sum_i \sum_j (\omega_{ij}^{(n)})^2$$

$$\tilde{\mathbf{W}}^{(n)} = (1-\lambda)\mathbf{W}^{(n)} - \vec{\delta}^{(n)}(\vec{y}^{(n-1)})^T$$

Dropout

Довольно интересной техникой по борьбе с переобучением сети является метод случайного отключения нейронов.

Во время обучения(и только во время его), мы просто отключаем нейроны с некоторой вероятностью. Это эквивалентно тому, что мы просто умножаем $\vec{y}$ на некоторый вектор случайных величин $\vec{d}$.

При этом $d_j = 0$ с вероятностью p и $d_j = 1$ с вероятностью $q = 1 - p$.

Во время обучения у нас ничего не меняется (кроме домножения на случайный вектор)

$$\vec{y}^{(n)} = \vec{d}^{(n)} \circ f(\vec{z}^{(n)})$$

а вот во время нормальной работы сети из-за того, что в слое во время обучения было меньше нейронов, нужно отмасштабировать вход

$$\vec{y}^{(n)} = q f(\vec{z}^{(n)})$$

На практике обычно используют обратные значения, масштабируя значения во время обучения и не меняя значения во время работы сети

во время обучения

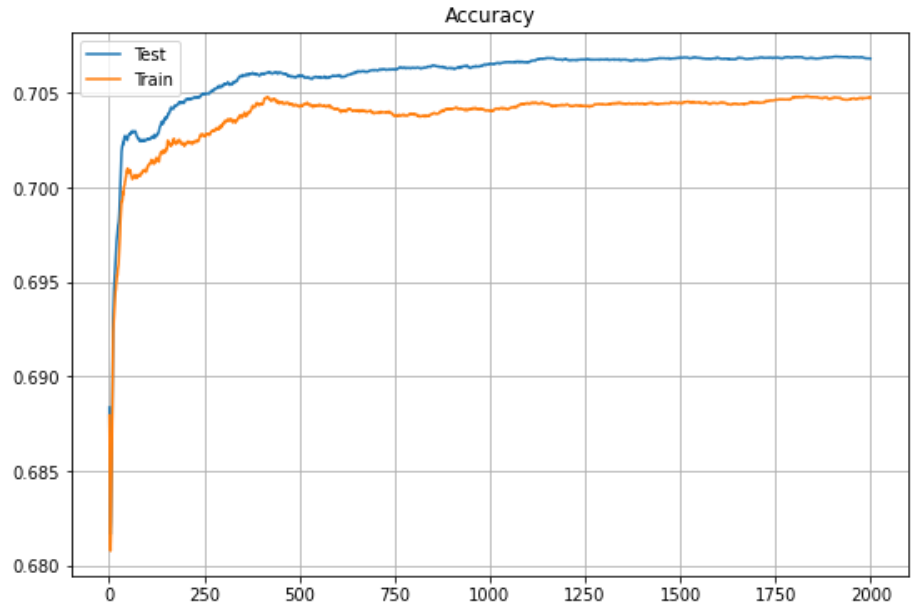
$$\vec{y}^{(n)} = \frac{1}{q}\vec{d}^{(n)} \circ f(\vec{z}^{(n)})$$

во время работы

In [71]:
```
net = MLPClassifier([25, 25], activation='logistic',
                    warm_start=True, alpha=0, max_iter=1,
                    solver='adam', tol=1e-12,random_state=42)

x = np.arange(1, 2000, 1)
y = []
y2 = []
ams = []
ams2 = []

for step in x:
    net.fit(X_train, y_train)

    y_p = net.predict(X_train)
    y.append(accuracy_score(y_train, y_p))
    ams.append(AMS(w_train, y_train, y_p))

    y2_p = net.predict(X_test)
    y2.append(accuracy_score(y_test, y2_p))
    ams2.append(AMS(w_test, y_test, y2_p))

    if step % 10 == 0:
        print("X = %6d, Acc = %.5f (%.5f)" % (step, y[-1], y2[-1]))
```

```
X =     10, Acc = 0.69212 (0.69092)
X =     20, Acc = 0.69718 (0.69505)
X =     30, Acc = 0.70035 (0.69823)
X =     40, Acc = 0.70262 (0.70008)
X =     50, Acc = 0.70269 (0.70083)
X =     60, Acc = 0.70297 (0.70056)
X =     70, Acc = 0.70294 (0.70056)
X =     80, Acc = 0.70253 (0.70068)
X =     90, Acc = 0.70246 (0.70088)
X =    100, Acc = 0.70254 (0.70105)
X =    110, Acc = 0.70263 (0.70136)
X =    120, Acc = 0.70279 (0.70129)
X =    130, Acc = 0.70290 (0.70139)
X =    140, Acc = 0.70339 (0.70179)
X =    150, Acc = 0.70364 (0.70201)
X =    160, Acc = 0.70404 (0.70224)
X =    170, Acc = 0.70421 (0.70259)
X =    180, Acc = 0.70439 (0.70240)
X =    190, Acc = 0.70446 (0.70233)
```

In [72]:
```
plt.title("Accuracy")
plt.plot(x, y, label="Test")
plt.plot(x, y2, label="Train")
plt.grid()
plt.legend()
plt.show()
```



In [73]:
```
plt.title("AMS")
plt.plot(x, ams, label="Test")
plt.plot(x, ams2, label="Train")
plt.grid()
plt.legend()
plt.show()
```