# Методы обучения без учителя

# Оглавление

# Обучение без учителя. Методы кластеризации

## 1.1 Задача кластеризации. Группы методов

На вход алгоритма кластеризации поступает множество объектов. Требуется разбить это множество объектов на группы таким образом, чтобы элементы внутри одной группы были похожи друг на друга, а элементы из разных групп отличались. Такие группы похожих объектов мы будем называть **кластеры**. Следует отличать методы кластеризации от методов классификации.

**Методы классификации** — это методы обучения с учителем. Это значит, что для каждого объекта нам известна его истинная метка принадлежности к классу. Затем, имея истинные метки, предсказания алгоритма и некоторую функцию потерь, алгоритмы классификации как-то подстраиваются так, чтобы допускать меньше ошибок на данных.

**Алгоритмы кластеризации** никак не используют информацию об истинных метках объектов и оперируют лишь похожестью объекта. Ну а **что такое похожесть, в каждом алгоритме кластеризации определяется по-своему**.

Зачем вообще нужна кластеризация? Основная её **цель заключается в том, чтобы выявить структуру в данных**. С помощью методов кластеризации мы можем автоматически найти группу похожих объектов, возможно, выделить аномалии, какие-то изолированные объекты, которые требуют дополнительного изучения или выбрасывания из данных, и кластеризация позволяет провести более детальный анализ самих кластеров. Иногда бывает полезно построить отдельные модели на каждом кластере вместо того, чтобы строить одну модель на всех данных и получить какой-то мусор.

Существует очень много методов кластеризации. Мы же с вами изучим три из них. Это методы кластеризации на основе прототипов, иерархические методы кластеризации и плотностные методы кластеризации.

**Методы разбиения на основе прототипов** заключаются в том, что кластеры характеризуются некоторым базовым элементом или прототипом. Например, в методе **k-средних** кластер характеризуется **центроидом** — центром масс объектов, из которых он состоит. Обычно в результате применения алгоритмов из этой группы мы получаем строгое разбиение всех наших объектов на кластеры, то есть одному объекту соответствует одна метка кластера.

**Иерархические алгоритмы** пошли дальше, и они позволяют получить целую структуру вложенных друг в друга кластеров. Иногда это бывает полезно для того, чтобы понять вообще структуру наших данных от начала до конца.

**Плотностные методы кластеризации** действуют совершенно иначе. В них кластер определяется, как область с большой плотностью точек, с большой плотностью объектов. Такая формулировка кластера позволяет выявлять произвольные формы кластеров и выделять объекты-выбросы, то есть элементы, вокруг которых нет других точек.

## 1.2 Метод k-средних

Начнем мы с изучения одного из самых популярных, вычислительно эффективных и не таких плохих методов кластеризации - метода к-средних. Возникает он из следующей оптимизационной задачи: у нас есть некоторое множество объектов, мы хотим найти в них $k$ кластеров $C_k$, каждый из которых характеризуется центроидом $\mu_k$. **Объект относится к соответствующему кластеру**, если его центроид находится ближе других центроидов к этому объекту, то есть

$$x_j \in C_k \Leftrightarrow \mu_k = arg\min_{\mu_j} ||x_j - \mu_j||^2$$

Отлично, давайте теперь **будем кластеризовывать объекты таким образом, чтобы минимизировать** следующий критерий - сумму квадратов расстояния между объектом и ближайшим к нему центроидом по всем кластерам, то есть значение

$$L(C) = \sum_{j=1}^{k} \sum_{x_j \in C_j} ||x_j - \mu_j||^2$$

Если взять производную по $\mu$, то становится понятно, что лучше всего выражать **центроид именно как центр масс объектов**, которые относятся к соответствующему кластеру, то есть

$$\mu_k = \frac{\sum_{x_j \in C_k} x_i}{|C_k|}$$

Итак, несмотря на то, что у нас есть формулировка критерия и мы знаем, как считать центроид. Для того, чтобы найти глобальный минимум этого функционала нужно перебирать все возможные разбиения объектов на $k$ кластеров, что нам конечно же не так интересно. Спасает нас **алгоритм k-средних, который позволяет найти локальный минимум этого критерия**. Состоит он из следующих шагов:

1. **Инициализация центроидов**
2. **Обновление состава кластеров**, то есть приписывание объекта к тому кластеру, центроид которого расположен ближе к этому объекту
3. **Обновление положения самих центроидов**, то есть пересчитывание положения центроидов, как центра масс объектов, которые лежат в соответствующем кластере
4. **Шаги 2 и 3 продолжаются до тех пор пока не выполнятся некоторые правила останова**, например, будет достигнуто какое-то максимальное количество итераций или пока кластеры не перестанут меняться

Результирующее разбиение на кластеры, которое получается после алгоритма k-средних в основном зависит от следующих факторов.

- Во-первых, **от начальной инициализации центроидов**. Оказывается, что на одним и тех же данных при одних и тех же $k$, но при разной инициализации мы можем получить совершенно разное разбиение на кластеры.
- Второй момент, это, собственно, определение **количества кластеров данных**, и этот вопрос совершенно нетривиальный.

Начнем с первого пункта: какие же **базовые варианты начальной инициализации центроидов** можно выделить?

- Первый, это **выбрать $k$ случайных объектов в наших данных в качестве центроидов** и уже с этих центроидов начинать следующие шаги алгоритма.
- Второй способ, который тоже показал себя на практике, это **использовать $k$ кластеров, полученных после применения иерархической кластеризации с методом Уорда**.
- И третий метод, который даже имеет собственное название, называется **k-means++**

Третий метод, k-means++, заключается в следующем. Первый центроид мы будем выбирать случайным образом среди всех точек в данных, которые у нас есть. Для каждой точки мы будем рассчитывать расстояние до ближайшего центроида, который ранее был уже нами инициализирован. Ну и в качестве следующего центроида мы будем выбирать также точку наших данных, но с вероятностью пропорциональной как раз этому расстоянию, которое мы посчитали на предыдущем шаге.

Второй важный фактор влияющий на результат применения алгоритма k-средних, это, собственно, параметр $k$ - количество кластеров, которые мы хотим получить в результате. **Как можно определить $k$?**

Во-первых, можно не использовать стандартный метод k-средних, а воспользоваться некоторой модификацией, которая позволяет автоматически определить это k. Такие модификациями являются, например, метод **X-means** или **intelligent k-means**.

Второй способ - воспользоваться некоторой мерой и качеством кластеризации. **Для каждого k и разбиения мы будем считать меру качества кластеризации** и в соответствии с этой мерой выберем лучшее разбиение и, соответственно, лучшее k.

И последний способ - это воспользоваться некоторыми эвристиками, одну из которых, **Метод Локтя**, мы с вами изучим далее. В чем заключается Метод Локтя?

Давайте для каждого k будем считать значение критерия k-means. Если расположить эти значения на графике, то мы получим убывающая функцию. Нам нужно **найти такое k, начиная с которого значение критерия k-means будет убывать не слишком быстро**. Этот эффект очень визуально похож на локоть и отсюда, собственно, название этого метода - Метод Локтя.

Важно понимать, что все эти **эвристики и меры качества в кластеризации носят лишь рекомендательный характер**, и, если вы действительно разбираетесь в данных, которые вы

хотите кластеризовать, то вам лучше опираться именно на эти знания, на знание вашей предметной области. Важно пытаться интерпретировать полученные кластеры, и тогда все эти меры качества и эвристики не будут играть для вас никакой роли.

**Давайте применим алгоритм k-средних на игрушечных данных и используем метод локтя, для того чтобы определить оптимальное k для соответствующего датасета**.

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

%matplotlib inline

plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (12,8)
```

Дополнительно воспользуемся библиотекой *sklearn.datasets*, для того чтобы **генерировать игрушечные данные**.

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=150, n_features=2, centers=4, random_state=1)
```

Таким образом мы сгенерировали 150 объектов, имеющие два признака и разделенные на четыре кластера.

Можно всё **визуализировать**, чтобы понять, как выглядят наши данные.

```
plt.scatter(X[:, 0], X[:, 1])
```

Теперь воспользуемся *sklearn.cluster*, для того чтобы **обучить на этих данных алгоритм k-средних**.

```
from sklearn.cluster import KMeans
```

Теперь **создадим экземпляр класса** *KMeans*.

```
kmeans = KMeans(n_clusters=2, init='k-means++', n_init=10, random_state=1)
```

**Обучим алгоритм**. Для этого воспользуемся методом *fit*.

```
kmeans.fit(X)
```

Теперь мы можем **получить метки кластеров** и также это дело визуализировать.

```
labels = kmeans.labels_
plt.scatter(X[:, 0], X[:, 1], c=labels)
```

Теперь давайте **поймем, какое k оказывается наилучшим для данного датасета**, и для этого мы используем Метод Локтя. Нам нужно итеративно для разных k применять метод k-средних и смотреть на значения критерия k-средних.

```
crit = []

for k in range(2, 8):
    kmeans = KMeans(n_clusters=k, random_state=1)
    kmeans.fit(X)

    crit.append(kmeans.inertia_)
```

Таким образом, в нашем списке будут храниться значения критерия качества k-means.

А теперь давайте это дело визуализируем.

```
plt.plot(range(2,8), crit)
```

И в данном случае оптимальным количеством кластеров для данного датасета по критерию локтя оказывается количество, равное четырем. И это тоже можно визуализировать.

```
kmeans = KMeans(n_clusters=4, init='k-means++', n_init=10, random_state=1)
kmeans.fit(X)
labels = kmeans.labels_
plt.scatter(X[:, 0], X[:, 1], c=labels)
```

## 1.3 Иерархическая кластеризация. Агломеративный алгоритм

Теперь мы рассмотрим алгоритмы иерархической кластеризации. Как было сказано ранее, иерархическая кластеризация позволяет получить не просто разбиение на кластеры, а целую иерархию вложенных кластеров. И оказывается, это сделать можно двумя способами.

- Первый способ — **агломеративный**. Мы начинаем с той ситуации, когда каждый объект является отдельным кластером. Затем на каждом шаге мы объединяем два наиболее близких кластера в один и останавливаемся тогда, когда все объекты оказываются в одном большом кластере.
- **Дивизивный подход** зеркален агломеративному. Мы начинаем с ситуации, когда все объекты объединены уже в один большой кластер, далее на каждом шаге большие кластеры мы бьем на два поменьше и останавливаемся тогда, когда каждый кластер состоит из одного объекта.

Так уж получилось, что агломеративный подход получил большую практическую популярность, чем дивизивный, и поэтому дальше мы сконцентрируемся именно на нем.

На первом шаге мы объединяем два наиболее близких объекта в один кластер. Дальше нам нужно как-то просчитать расстояние между этим кластером и всеми остальными кластерами. Оказывается, есть несколько способов пересчета расстояния между кластерами после их объединения. По-английски эти способы называются **Linkage**.

- Первый способ — **Single Linkage**, и в нем расстояние между кластерами вычисляется как минимальное между всеми парами объектов из разных кластеров.

- Второй способ — **Complete Linkage**. В нем расстояние между кластерами считается как максимальное расстояние между парами объектов из разных кластеров.
- **Average Linkage** — это просто среднее расстояние между всеми парами объектов из разных кластеров.
- **Centroid Linkage** — это расстояние между центроидами разных кластеров.
- **Ward Linkage** — это немного модифицированный Centroid Linkage с учетом размеров самих кластеров.

Продолжая нашу процедуру объединения, в результате мы получим нечто, называющееся **диаграммой вложения**. В принципе, она нам иллюстрирует иерархическую структуру вложенных кластеров на плоскости, но что, если у нас больше, чем два признака и слишком много объектов? Оказывается, есть другой, более элегантный способ визуализации иерархии, и он называется **дендрограмма**. На оси $X$ у дендрограммы расположены названия объектов, а на оси $Y$ — расстояние, на котором соответствующие объекты, кластеры, объединяются друг с другом.

Мы получили дендрограмму, и нам нужно как-то понимать, хорошая ли дендрограмма получилась, потому что от нее зависит конечное разбиение уже на сами кластеры. Для того чтобы научиться считать качество дендрограммы, нам нужно научиться считать некоторые величины.

Одной из таких величин является **кофенетическое расстояние**, и она считается между всеми парами исходных объектов. Кофенетическое расстояние между двумя объектами равно высоте дерева, на котором два объекта объединились в один кластер. Далее, имея массив кофенетических расстояний между всеми парами объектов и массив обычных попарных расстояний, мы можем посчитать корреляцию между этими массивами, и это и будет **кофенетическая корреляция**. Эвристическое правило звучит так: **у хорошей дендрограммы корреляция между попарным расстоянием и кофенетическим расстоянием будет высокой**. Таким образом, посчитав разные дендрограммы, мы можем понимать, какая из них хорошая, а какая не очень.

Итак, мы с вами рассмотрели алгоритмы агломеративной иерархической кластеризации.

Основными **плюсами** этих методов является то, что мы получаем целую иерархию кластеров вместо одного разбиения, мы имеем наглядное представление этой иерархии с помощью дендрограммы, и у нас есть множество различных способов пересчета расстояний после объединения в большие кластеры.

Но у этого метода есть свои **недостатки**. Во-первых, так как нам нужно хранить все попарные расстояния, то сложность этого алгоритма по памяти составляет $O(N^2)$. Более того, нам нужно каждый раз пересчитывать эти расстояния и находить среди них минимальное, и таким образом сложность по времени этого алгоритма равна $O(N^2 \log N)$, что довольно-таки нескромно.

**Давайте применим алгоритм иерархической кластеризации к небольшому объему данных, вытащим из этой иерархии кластеры и попытаемся их проинтерпретировать**.

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

```
%matplotlib inline

plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (12,5)
```

Воспользуемся небольшим датасетом, в котором объектами являются продукты питания, а их признаками являются энергетическая ценность в килокалориях, количество жиров и протеинов в граммах, количество кальция и железа в миллиграммах.

```
df = pd.read_csv('food.txt', sep=' ')
df.head()
```

**Первое, что стоит заметить, — это то, что все признаки в этом датасете имеют разную шкалу**, причем наибольшую шкалу имеет признак именно энергетической ценности, его шкала на порядок больше, чем шкалы всех остальных признаков. Это значит, что когда мы будем считать попарное расстояние между объектами, то наибольший вклад в это расстояние будет вносить признак энергетическая ценность. И нас это не интересует. Мы хотим, чтобы все признаки вносили одинаковый вклад в расстояние между объектами.

Есть несколько способов **привести признаки к единой шкале**. Один из таких способов — это вычесть из каждого признака его среднее значение и затем поделить на стандартное отклонение, что мы и сделаем.

```
X = df.iloc[:, 1:].values
X = (X - X.mean(axis=0))/X.std(axis=0)
```

Давайте проверим, что всё правильно. Посчитаем среднее значение результирующей матрицы по всем признакам в отдельности.

```
X.mean(axis=0)
```

Все средние значения либо очень близки к нулю, либо равны нулю. И теперь стандартное отклонение

```
X.std(axis=0)
```

По всем признакам в отдельности оно равно единице.

Теперь мы перешли к одинаковой для всех признаков безразмерной шкале, вклад их будет одинаковый в расчет расстояния.

Теперь можно **применять иерархическую кластеризацию**. Несмотря на то, что в библиотеке *skit-learn* есть инструменты для того, чтобы делать агломеративную кластеризацию, мы будем использовать инструментарий библиотеки *SciPy*.

```
from scipy.cluster.hierarchy import linkage, fcluster, dendrogram
```

Первый метод, который нам понадобится — это *linkage*.

```
Z = linkage(X, method='average', metric='euclidean')
```

В результате в переменной Z мы имеем табличку из четырех столбцов. Первые два столбца содержат в себе индексы объектов, которые на этом шаге будут объединяться. Третий столбец содержит

расстояние, на котором эти два объекта будут объединяться. И, наконец, четвертый столбец показывает нам, какой размер кластера получится после объединения на этом шаге.

**Получим визуализацию нашей дендрограммы**.

```
dend = dendrogram(Z, orientation='left', color_threshold=0.0, labels=df.Name.values)
```

Теперь хотелось бы из всей этой иерархии получить **разбиение на кластеры**. Для этого мы будем использовать метод *fcluster*.

```
label = fcluster(Z, 2.2, criterion='distance')
np.unique(label)
```

В результате мы получили шесть кластеров. Действительно, если посмотреть на дендрограмму, прикинуть, где находится порог 2.2, и провести линию, то как раз понятно, что это за шесть кластеров у нас получилось.

Теперь давайте их все **проинтерпретируем**. Объектов у нас не так много, поэтому мы можем посмотреть на все объекты целиком.

```
df.loc[:, 'label'] = label

for i, group in df.groupby('label'):
    print('=' * 10)
    print('cluster {}'.format(i))
    print group
```

Мы видим перечисление тех объектов, которые попали в тот или иной кластер. Давайте попытаемся их проинтерпретировать. Например, в первый кластер попали два объекта — это мидии. Почему они выделились в отдельный кластер? Потому что при их низких значениях энергетической ценности и концентрациях протеинов и жиров у них довольно высокие концентрации кальция и железа. Второй кластер состоит из продуктов питания, богатых калориями и жирами. Третий кластер состоит из рыбных продуктов, и они отличны тем, что при не выделяющихся энергетической ценности, протеинах, жирах и железе довольно большая концентрация кальция. Четвертый кластер не выделяется ничем, это просто продукты питания, у которых довольно средние показатели по всем признакам. И, наконец, у нас есть изолированные кластеры пять и шесть, которые отличны по какому-то одному из признаков, например, последний объект, сардины, имеет очень большое значение в концентрации кальция.

## 1.4 DBSCAN

**DBSCAN относится к методам плотностной кластеризации**. То есть он выделяет плотные кластеризации — это области высокой концентрации точек, разделенные областью низкой концентрации точек. Особенностью алгоритмов плотностной кластеризации является то, что **кластеры, полученные в результате, могут иметь произвольную форму**. И вторая особенность заключается в том, что **алгоритм позволяет выявить выбросы данных**, которые, например, можно выкинуть.

Введем следующие определения и обозначения. Через $\epsilon$ мы обозначим размер окрестности каждой точки, $minPts$ — это некоторое пороговое значение количества объектов, которые будут лежать в этой $\epsilon$-окрестности, $N_\epsilon(p)$ — это множество точек, которое лежит в этой $\epsilon$-окрестности, включая саму точку $p$.

Точка p называется **непосредственно плотно-достижимой** из точки q, если $p \in N_\epsilon(q)$ и $|N_\epsilon(q)| \geq minPts$

Точка p **плотно-достижима** из точки q, если существует такая последовательность $p, p_1, p_2, ..., p_{n-1}, q$ в которой каждая точка непосредственно плотно-достижима из предыдущей.

И точка p называется **плотно-связанной** с точкой q, если существует точка o, такая, что, p и q плотно-достижимы из o.

Тогда **кластером** мы будем называть максимальное множество плотно-связанных точек окрестности точки.

Также можно выделять некоторые типы точек. Например, точка p будет называться **core-объектом**, если количество объектов в ее $\epsilon$-окрестности больше, чем $minPts$.

Точка p может называться и **граничным объектом**, если количество объектов в ее $\epsilon$-окрестности меньше, чем $minPts$, но существует какой-то core-объект, расстояние от которого до этой точки меньше, чем $\epsilon$, то есть он лежит в этой $\epsilon$-окрестности.

И точка p может называться **шумовым объектом**, если она не граничный объект и не core-объект.

Как же выглядит сам **алгоритм**? На вход ему подаются, собственно, данные, minPts и $\epsilon$ - наши параметры.

1. Вначале мы инициализируем список непосещенных точек $NV$, счетчик для метки кластеров $j$ и множество шумовых объектов.

2. Начинаем итерировать по непосещенным объектам.

3. Если мы его еще не посетили, то выкидываем его из списка $NV$ и рассматриваем его $\epsilon$-окрестность.

4. Если количество объектов в этой $\epsilon$-окрестности недостаточно большое, то есть меньше minPts, то мы пока помечаем этот объект как шум. Если же $\epsilon$-окрестность достаточно плотная, то мы начинаем расширять новый кластер.

5. Как происходит расширение.

   (a) Во-первых, мы текущий объект добавляем в наш кластер.

   (b) Далее мы рассматриваем все объекты, которые находятся в $\epsilon$-окрестности этой точки.

   (c) Если мы эту точку еще не посещали, то выкидываем ее из списка непосещенных объектов, смотрим уже на ее $\epsilon$-окрестность, и если она оказывается достаточно плотной, то расширяем список объектов, которые мы хотим еще изучить, этим множеством точек.

   (d) И если у нас точка еще не принадлежит никакому кластеру, то, соответственно, мы добавляем ее в текущий кластер.

Таким образом, мы с вами изучили алгоритм DBSCAN.

Какие же основные **преимущества** он нам предлагает? Во-первых, он сам определяет количество кластеров. Нам нужно задать только параметр minPts и $\epsilon$, и дальше сколько кластеров получится, столько получится. Кластеры могут иметь произвольную форму, и алгоритм DBSCAN также позволяет выявлять выбросы данных.

Однако у DBSCAN есть определенный **недостаток**, и он связан с тем, что он не работает в том случае, когда кластеры имеют разную плотность.

**Давайте применим алгоритм DNSCAN** для кластеризации географических данных, и в вместе с этим изучим способ определения значений параметров $\epsilon$ и minPts.

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

%matplotlib inline

plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (12,5)
```

У нас есть небольшой датасет на 14000 точек, для каждой из которых известна ее координата по широте и долготе.

```
df_geo = pd.read_csv('geo_data.txt', sep='\t', header=None, names=['lat', 'lon'])/10000
df_geo.head()
```

Для того чтобы визуализировать эти точки, будем использовать некоторое дополнение к библиотеке *Matpotlib*, которое скорее всего у вас не установлено, и вам нужно будет это сделать самостоятельно.

```
import mpl_toolkits.basemap as bm
```

Функция, которая уже рисует точки на карте.

```
def plot_geo(lat, lon, labels=None):
    try:
        lllat, lllon = lat.min()-1, lon.max()+1
        urlat, urlon = lat.max()+1, lon.min()-1

        plt.figure(figsize=(10, 10))

        m = bm.Basemap(
            llcrnrlon=lllon,
            llcrnrlat=lllat,
            urcrnrlon=urlon,
            urcrnrlat=urlat,
            projection='merc',
            resolution='h'
        )
```

```
        m.drawcoastlines(linewidth=0.5)
        m.drawmapboundary(fill_color='#47A4C9', zorder=1)
        m.fillcontinents(color='#EBC4D8',lake_color='#47A4C9', zorder=2)

        parallels = np.linspace(lllat, urlat, 10)
        m.drawparallels(parallels,labels=[1,0,0,0],fontsize=10)
        # draw meridians
        meridians = np.linspace(urlon, lllon, 10)
        m.drawmeridians(meridians,labels=[0,0,0,1],fontsize=10)

        m.scatter(lon, lat, latlon=True, cmap=plt.cm.jet, zorder=3, lw=0, c=labels)
    except:
        print('something went wrong')
        plt.scatter(x=lon, y=lat, c=labels, cmap=plt.cm.jet)
        plt.axis('equal')


plot_geo(df_geo.lat.values, df_geo.lon.values)
```

**Способ оценки параметров для DBSCAN** заключается в следующем. Вначале нам нужно понять, какой размер кластера мы будем считать подходящим, чтобы он был выделен, то есть нам нужно заранее указать значение minPts. Ну например, по каким-то соображениям мы считаем, что минимум точек в кластере должно быть равно 20. Теперь исходя из этого нужно понять, какое же значение $\epsilon$ адекватно выставить. Сделать это нужно следующим образом: нужно посчитать расстояние для каждой точки от ее minPts-нутого ближайшего соседа. Далее эти расстояния сортируются по возрастанию и визуализируются. И по полученному графику мы можем примерно прикинуть, какой $\epsilon$ нам стоит использовать.

Для начала переведём наши данные из широты и долготы в радианы.

```
X = df_geo.values
X = np.radians(X)
```

Теперь, для того чтобы посчитать расстояние до minPts-нутого ближайшего соседа, воспользуемся классом *NearestNeighbors*из библиотеки *sklearn*.

```
from sklearn.neighbors import NearestNeighbors
model = NearestNeighbors(n_neighbors=20, algorithm='ball_tree', metric='haversine')
model.fit(X)

dist, _ = model.kneighbors(X, n_neighbors=20, return_distance=True)
```

Массив dist получился на самом деле размера: количество точек умноженное на 20, потому что у нас 20 ближайших соседей. Соответственно, нам нужен только 20-й столбец этого массива, что мы и выделим.

```
dist = dist[:, -1]
```

Теперь этот массив мы отсортируем по возрастанию и визуализируем.

```
dist = np.sort(dist)
plt.plot(dist)
```

На этом графике по оси $X$ у нас идет просто число от 0 до 13467, а по оси $Y$ отложено расстояние до minPts-нутого ближайшего соседа. Так вот **способ оценивания $\epsilon$ по этому графику заключается в том, что мы должны выбрать такой $\epsilon$, на котором этот график начинает резко возрастать**.

Аргументация этого метода заключается в том, что если мы, например, выберем $\epsilon = 0.005$, то это значит, что все те точки, которые находятся по правую сторону от этой точки, будут либо граничными, либо выбросами. Ну и далее, в зависимости от того, сколько мы хотим чтобы было выбросов, мы как-то пытаемся прикинуть значение $\epsilon$.

Установим $\epsilon = 0.002$.

```
eps = 0.002
```

И теперь будем **кластеризовывать объекты с помощью DBSCAN**.

```
from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps=eps, min_samples=20, metric='haversine', algorithm='ball_tree')
dbscan.fit(X)
```

Сохраним **метки полученных кластеров**.

```
labels = dbscan.labels_
```

Посмотрим, **сколько же кластеров у нас получилось и каковы их размеры**.

```
pd.Series(labels).value_counts()
```

Те объекты, которые помечены как выбросы, имеют метку -1.

Теперь давайте **визуализируем ту же самую карту**, с тем исключением, что мы выкинем оттуда объекты, которые являются выбросами.

```
idx = labels!=-1
plot_geo(df_geo.loc[idx, 'lat'].values, df_geo.loc[idx, 'lon'].values, labels=labels[idx])
```

Мы видим, что карта получилась более чистой, чем она была. Мы удалили довольно много выбросов. А дальше остается **анализировать эти кластеры и переподбирать значения параметра $\epsilon$ по этому графику**.

## 1.5 Оценки качества кластеризации

Перейдем к изучению оценок качества кластеризации. Меры качества кластеризации можно разделить на две группы.

Первая группа - это **качество по отношению к некоторому эталонному разбиению**. Для некоторых наборов данных уже известно истинное разбиение на кластеры и мы хотим просто сравнить полученную кластеризацию с этим истинным разбиением. Такие методы оценок называются

**External measures**, и по-русски их будем называть **мерой качества разбиения**.

Вторая группа мер - это **меры качества по отношению к некоторым нашим представлениям о том, что такое хорошая кластеризация**. По-русски я будем их называть **мерой валидности разбиения**. А по-английски они называются **Internal measures**.

Первая мера качества по отношению к эталонному разбиению - это **Rand Index**.

$$Rand(\pi^*, \hat{\pi}) = \frac{a+d}{a+b+c+d}$$

где $\pi^*$ - полученное разбиение на кластеры, $\hat{\pi}$ - эталонное разбиение,
$a$ - количество пар объектов, находящихся в одинаковых кластерах в $\pi^*$ и в $\hat{\pi}$,
$b(c)$ - количество пар объектов в одном и том же кластере в $\pi^*(\hat{\pi})$, но в разных в $\hat{\pi}(\pi^*)$,
$d$ - количество пар объектов в разных кластерах в $\pi^*$ и в $\hat{\pi}$.

Можно записать Rand Index через True Positive, False Positive, True Negative и False Negative и получить подоюную формулу

$$Rand(\pi^*, \hat{\pi}) = \frac{TP+TN}{TP+FP+FN+TN}$$

Почему мы рассматриваем именно пары объектов? Дело в том, что сама метка кластера не играет никакой роли (можно получить одно и то же логическое разбиение точек, у которых будут различные наборы меток). Значит сравнивать метки попросту не имеет смысла.

Однако, индекс Рэнда не лишен недостатков и одним из таких недостатков является то, что для случайных разбиений мы можем получить довольно высокое значение индекса Рэнда. Искоренить этот недостаток призвана **корректировка (Adjusted Rand Index)**. Выглядит она следующим образом

$$ARI(\pi^*, \hat{\pi}) = \frac{Rand(\pi^*, \hat{\pi}) - \mathbb{E}(Rand(\pi^*, \hat{\pi}))}{\max(Rand(\pi^*, \hat{\pi})) - \mathbb{E}(Rand(\pi^*, \hat{\pi}))}$$

Также, для измерения качества разбиения можно считать уже известные вам меры вида Precision, Recall и F-меры. В данном случае их формулы совершенно не меняются. Меняется просто способ расчета True Positive, False Positive и так далее.

Теперь изучим **меры валидности кластеров**. Как я уже сказал, мера валидности кластеров измеряется по отношению к нашему представлению о том, что такое хорошая кластеризация.

Что такое хорошая кластеризация? Во-первых, **объекты внутри одного кластера должны быть компактными**, то есть очень похожи друг на друга, и, при этом, **разные кластеры должны располагаться далеко друг от друга**, то есть быть максимально не похожими. И разные меры качества считают вот эти критерии по-разному, и поэтому их очень много. Эти меры можно использовать для, например, подбора параметров алгоритмов, то есть вы прогоняете один и тот же алгоритм кластеризации с разными параметрами, смотрите на меры и выбираете наилучшую настройку алгоритма по отношению к этой мере.

Первая мера - это **Silhouette**. Пусть дана кластеризация на K кластеров и объект $x_i$ попал в кластер $C_k$. Тогда для этого объекта мы можем посчитать следующие величины.

Во-первых, это **среднее расстояние от объекта до всех остальных объектов его же кластера** $a(i)$.

Вторая величина - $b(i)$ - это **минимальное среднее расстояние от этого объекта до объекта какого-то кластера** $C_j$.

Таким образом, как раз **компактность у нас выражается величиной** $a(i)$**, а разделимость** $b(i)$. Тогда

$$silhouette(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

Вторая мера - это **Dunn Index**. Для того чтобы ее получить, нам нужно научиться считать **расстояние между кластерами** - и мы его определим как минимальное расстояние между объектами из двух разных кластеров - $\delta(C_k, C_l)$. И вторая величина - это **диаметр кластера**, то есть мы будем считать уже максимальное расстояние между всеми парами объектов, но уже внутри одного кластера - $\Delta(C_k)$. И тогда индекс будет равен

$$DI = \frac{\min_{k \neq l} \delta(C_k, C_l)}{\max_k \Delta(C_k)}$$

И ещё один индекс - это **Davies-Bouldin Index**. Для того, чтобы его посчитать, нам нужно научиться считать **разброс данных внутри кластера**. Это будет просто среднее расстояние между объектами внутри кластера и центроидом этого кластера.

$$S(C_k) = \frac{1}{|C_k|} \sum_{x_i \in C_k} d(x_i, \mu_k)$$

Тогда индекс равен

$$DB = \frac{1}{K} \sum_k \max_{l \neq k} \frac{S(C_k) + S(C_l)}{d(\mu_k, \mu_l)}$$

# Методы понижения размерности

## 2.1 Введение. Мотивация

Итак, представим, что у нас есть такая задача. Дано исходное признаковое описание объекта в виде матрицы размера $N \cdot D$, и требуется получить новое признаковое описание, тоже в виде матрицы, но уже размера $N \cdot d$, где d должно быть сильно меньше, чем D.

Эту задачу можно решать двумя способами. Первый способ — это способы отбора признаков, и второй способ — это методы понижения размерности.

Чем же эти методы отличаются? В методе отбора признаков мы просто выбираем какое-то подмножество исходных столбцов нашей матрицы, их оставляем, а все остальное выкидываем. В методах же понижения размерности мы прогоняем наши признаки через некоторое особенное преобразование таким образом, что на выходе остается только d столбцов исходной матрицы.

Какие же могут быть мотивации использования методов понижения размерности?

Первая мотивация — это так называемое **проклятие размерности**. Это такой феномен, который в разных областях проявляет себя по-разному. В нашем случае он проявляет себя с двух сторон.

**Первая сторона** заключается в следующем. Давайте представим, что у нас есть признак, который принимает одно из трех значений. Тогда для того, чтобы описать такое признаковое пространство, нам будет достаточно трех объектов в нашем наборе данных. Добавим второй признак, который пока тоже будет принимать три значения. Тогда для того, чтобы описать такое признаковое пространство, будет достаточно уже девяти объектов. Добавим третий признак — нужно будет 27 объектов и так далее. То есть мы видим, что с увеличением количества признаков количество объектов, необходимое для того, чтобы описать такое признаковое пространство, возрастает экспоненциально. Понятное дело, что наши dataset ограничены, и они просто не могут иногда описывать полностью наше признаковое пространство. В свою очередь это приводит к тому, что **алгоритмы переобучаются, и их качество падает с увеличением количества признаков**.

Вторая сторона проклятия размерности — это геометрическая его сторона, то есть оказывается, что **с увеличением количества признаков объекты становятся одинаково похожи или не похожи друг на друга**. Это является бичом таких методов, как метод ближайших соседей, например.

Вторая причина, по которой можно использовать методы понижения размерности — это **визуализация многомерных данных**. Ну действительно, если у нас будет какое-то умное преобразование, которое сохраняет структуру исходных данных, но переводит их в пространство размера

два, то на это будет гораздо приятнее смотреть, чем смотреть на все возможные пары исходных признаков.

И третья очевидная причина — это **экономия ресурсов**, то есть хранить d столбцов гораздо экономичнее, чем хранить D столбцов.

## 2.2 Метод Главных Компонент (Principal Component Analysis)

Итак, перейдем к изучению метода главных компонент. Мы хотим получить новое признаковое пространство со следующими условиями.

- Во-первых **новые оси координат, которые в дальнейшем будут называться главными компонентами должны быть ортогональны друг к другу**.
- Во-вторых, для того чтобы получить значение новых признаков нам нужно домножить наши исходные признаки на некоторые коэффициенты, то есть **новые признаки будут являться линейной комбинации исходных признаков**. И вот эти коэффициенты нам как раз надо найти.
- И третье условие - **новые признаки должны сохранить как можно больше дисперсии, как можно больше изменчивости и вариативности наших исходных данных**.

Для того, чтобы выводить главные компоненты, нам нужно вспомнить о том, что такое задача о собственном векторе, о собственном числе матрицы и о том, что такое ковариация.

Начнем с собственного числа, с собственного вектора матрицы. Пусть у нас есть некоторая квадратная матрица $A$. Рассмотрим такое соотношение

$$A\vec{v} = \lambda\vec{v},$$

где $\vec{v} \neq \vec{0}$.

$\vec{v}$ будет называться **собственным вектором** матрицы $A$, а, собственно, число $\lambda$ будет называться **собственным числом** матрицы $A$.

Собственных векторов и собственных чисел для матрицы может быть много. Более того, для одного собственного числа может быть несколько собственных векторов.

**Ковариация** - это мера линейной зависимости двух случайных величин, но если вы знаете, что такое корреляция, то что такое ковариация вам тоже будет просто понять. **Корреляция** - это просто нормированная ковариация. То есть мы можем посчитать матрицу ковариаций между всеми парами признаков.

Оказывается, что матрица ковариаций обладает довольно приятными свойствами.

Во-первых, **ее собственные числа все больше нуля** и заранее договоримся, что собственные числа мы будем упорядочивать по убыванию.

Кроме того **собственные вектора при разных собственных числах будут ортогональны друг к другу**. То есть их скалярное произведение будет равно нулю.

Давайте начнем выводить компоненты. Вспомним, что мы хотим получить линейное преобразование исходных признаков. Записать это можно в виде произведения

$$Z = XA = X \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}$$

где $X$ - матрица исходных признаков, а в матрице $A$ по столбцам расположены вектора для каждой главной компоненты.

Мы хотим **максимизировать вариативность внутри главной компоненты**, то есть дисперсия по новым признакам должна быть максимальной.

$$Var(\vec{z}_i) = \vec{a}_i^T S \vec{a}_i \to \max$$

И вместе с этим мы требуем **ортогональность наших новых признаков**, то есть ковариация должна быть нулевой.

$$Cov(\vec{z}_j, \vec{z}_k) = \vec{a}_j^T S \vec{a}_k = 0$$

Начнем с первой главной компоненты

$$\vec{z}_1 = \vec{a}_1^T X$$

Для этого **запишем задачу условной оптимизации**. Мы будем максимизировать дисперсию, дополнительно потребовав, чтобы норма вектора была равна 1.

$$\begin{cases} \vec{a}_1^T S \vec{a}_1 \to \max_{\vec{a}_1}, \\ \vec{a}_1^T \vec{a}_1 = 1, \end{cases}$$

Используя метод Лагранжа, получим, что $\vec{a}_1$ - это какой-то собственный вектор матрицы $S$ и $\nu$, множитель Лагранжа, при нём - это какое-то собственное число. Вновь подставив это соотношение в целевой функционал мы получим, что мы должны максимизировать $\nu$

$$\vec{a}_1^T S \vec{a}_1 = \nu \vec{a}_1^T \vec{a}_1 = \nu \to \max$$

То есть $\nu$ должно быть наибольшим собственным числом матрицы $S$.

Дополнительно отметим, что $\nu$ или $\lambda_1$ - **первое собственное число равно дисперсии первой главной компоненты**.

Дальше будем выводить вторую главную компоненту. Аналогичным образом окажется, что ответом в этой задаче будет собственный вектор при втором по величине собственном числе матрицы $S$.

В итоге **линейное преобразование $A$ задаётся собственными векторами матрицы ковариаций**.

**Давайте еще раз проговорим - в чём заключается метод главных компонент**.

Итак, на вход к нему подаются исходные данные в виде матрицы $X$ и количество компонент $d$, которые мы хотим получить.

1. Первым делом мы наши **данные центрируем**, то есть из каждого столбца вычитаем его среднее значение. **Дополнительно мы можем их еще отшкалировать** и это может быть полезно, если шкалы наших признаков отличаются друг от друга.

2. Дальше мы **считаем матрицу ковариаций** $S$ **и находим** $d$ **наибольших собственных чисел и соответствующих собственных векторов данной матрицы**.

3. И в конце концов мы **составляем матрицу преобразования** - матрицу состоящую из этих собственных векторов - и получаем новое признаковое описание.

**Сколько же компонент стоит нам выводить?** Мы поняли, что $\lambda_j$, $j$-ое собственное число матрицы $S$, равно дисперсии, $j$-ой главной компонентой. Таким образом, мы можем посчитать долю дисперсии, которая объясняет это главная компонента, и также можем посчитать долю дисперсии, которая объясняется первыми компонентами, которые мы вывели.

Обычно выставляют некоторый порог типа 80%. То есть, если первые k компонент у нас будут выделять 80% дисперсии в наших данных, то на этом мы можем остановиться и перестать искать собственные вектора, собственные числа.

## 2.3 Сингулярное разложение матрицы и связь с PCA

Рассмотрим одно из самых важных матричных разложений - **сингулярное разложение**, и поймём, как оно связано с методом главных компонент.

Оказывается, что любую матрицу $X$ можно представить в виде произведения трех матриц: $X = U\Sigma V^T$

$U$ - матрица, состоящая из собственных векторов матрицы $XX^T$.

$V$ - матрица, состоящая из собственных векторов матрицы $X^TX$.

$\Sigma$ — это диагональная матрица, состоящая из так называемых сингулярных чисел, которые, в свою очередь, равны корню из соответствующих собственных чисел собственных векторов.

Мы можем сделать **неполное сингулярное разложение** $X_k$ матрицы $X$. Для этого в матрице $\Sigma$ мы будем рассматривать только k наибольших сингулярных чисел, в матрице $U$ будем рассматривать только первые k ее столбцов, а в матрице $V^T$ только первые k ее строк. Таким образом, мы получим некоторую аппроксимацию матрицы $X$.

Оказывается, такое неполное сингулярное разложение обладает некоторыми приятными свойствами. Например, если мы рассмотрим **матрицу ошибок аппроксимации**

$$E(\tilde{X}) = X - \tilde{X}$$

То оказывается, что **неполное сингулярное разложение является лучшей аппроксимацией в смысле**

$$X_k = arg\min_{\tilde{X}} \sqrt{\sum_{i,j} E_{ij}^2(\tilde{X})}$$

Сингулярное разложение имеет довольно широкий спектр применения, и, в частности, в дальнейшем мы изучим, как оно применяется в рекомендательных системах, но сейчас мы поймем, как оно связано с методом главных компонент, то есть в задаче сжатия признакового пространства.

Есть два пути вычисления в методе главных компонент. Первый путь мы уже с вами прошли. Мы берём исходную матрицу X, находим её матрицу ковариаций, считаем k собственных векторов при k наибольших собственных числах, и переводим наши исходные признаки в новое признаковое пространство. Мы также можем заметить, что в сингулярном разложении также используются собственные векторы матрицы ковариации. В данном случае они у нас будут лежать в матрице V.

Таким образом, чтобы сделать метод главных компонент с помощью сингулярного разложения, мы должны посчитать неполное сингулярное разложение до k-го сингулярного числа, получить ту же матрицу коэффициентов, которая будет лежать в матрице V, и перевести наши исходные признаки в новое признаковое пространство с помощью этой матрицы.

## 2.4 Применение PCA на данных

На этом занятии мы с вами применим метод главных компонент тремя разными способами и заодно проверим как он влияет на качество задачи и классификации.

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

%matplotlib inline

plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (12,5)
```

Мы будем использовать dataset про вино. Он состоит из одного целевого признака качества и 11 остальных признаков, которые как то характеризуют вино.

```
df_wine = pd.read_csv('winequality-red.csv', sep=';')4
df_wine.loc[:, 'quality_cat'] = (df_wine.quality > 5).astype(int)
df_wine = df_wine.drop('quality', axis=1)

X = df_wine.iloc[:, :-1].values
y = df_wine.iloc[:, -1].values

df_wine.head()
```

Мы его перевели из разряда задачи регрессии в задачу классификации. То есть мы будем с вами отличать вино хорошее от вина плохого по его каким-то характеристикам.

Итак, первый способ, которым мы будем применять, это коробочный метод, то есть мы возьмем **готовое решение из библиотеки** *sklearn*. Метод главных компонент у нас лежит в разделе *decomposition*, и тут делается все точно так же, как мы всегда раньше делали

```
from sklearn.decomposition import PCA

pca = PCA(n_components=6)
pca.fit(X)
```

Для того чтобы получить уже представление о наших объектах в сжатом признаковом пространстве, используем метод *transform*.

```
Z = pca.transform(X)
```

Что у нас лежит в *pca*? Во-первых, коэффициенты, которые необходимо использовать для того, чтобы перейти от исходного признакового пространства в сжатое.

Так же в *pca* лежит атрибут *exlplained_variance_* и *explained_variance_ratio_* - Вот, как раз, доли объяснённой дисперсии и в принципе дисперсия для каждой из компонент.

Теперь попробуем применить метод главных компонент **с помощью сингулярного разложения**. Сингулярное разложение у нас лежит в модуле *numpy*.

```
from numpy.linalg import svd

u, s, vt = svd(X_, full_matrices=0)
S = np.diag(S)

X_svd = u.dot(S).dot(vt)
```

Таким образом мы возьмем *vt*, возьмем оттуда 6 первых строк, потому что, у нас всего 6 компонент было изначально и транспонируем результат. Далее сделаем преобразование *Z_svd*.

```
v = vt[:6, :].T
Z_svd = X_.dot(v)
```

При сравнении результата с предыдущим способом получается всё одно и то же с точностью до знака.

Теперь сделаем метод главных компонент **с помощью собственных чисел и собственных векторов матрицы ковариаций**. Для начала посчитаем саму матрицу ковариаций, а затем найдем собственные вектора и собственные числа этой матрицы.

```
from numpy.linalg import eig

C = X_.T.dot(X_)
lambd, W = eig(C)
```

Вернемся к **объясненной дисперсии**. Напомним, что она лежит в *pca.explained_variance_ratio_*, и теперь посчитаем всё то же самое через собственные числа матрицы ковариации.

```
lambd/lambd.sum()
```

Также, обратите внимание, что если мы нарисуем накопительный график: доля объясненной дисперсии с каждой компонентой, то получится довольно странный результат. Оказывается у нас

90% всей дисперсии объясняется уже первой компонентой. Значит ли это, что нам нужно использовать только один какой-то признак? Нет, необязательно. Дело в том, что у нас признаки изначально в разной шкале. Поэтому **рекомендуется, кроме центрирования матрицы, ещё её нормировать, то есть делить на стандартные отклонения**.

Итак, теперь посмотрим, **как влияет метод главных компонент на качество классификации**. Наш базовый *Pipeline* будет состоять из двух этапов. Сначала мы признаки центрируем, потом применим логистическую регрессию и посмотрим, какой получим результат без метода главных компонент.

```
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import StratifiedKFold, cross_val_score

model_baseline = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', LogisticRegression()
])

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=123)
base_score = cross_val_score(model_baseline, X, y, scoring='accuracy', cv=cv).mean()
```

Теперь добавим в наш *Pipeline* метод главных компонент. Это довольно просто сделать. То есть теперь наш *Pipeline* будет выглядеть так.

```
k = range(1, 12)
scores = []

for n in k:
    model = Pipeline([
        ('scaler', StandardScaler()),
        ('pca', PCA(n_components=n)),
        ('clf', LogisticRegression())
    ])
    scores.append(cross_val_score(model, X, y, scoring='accuracy', cv=cv).mean())
```

И давайте это нарисуем.

```
plt.plot(k, scores)
plt.hlines(base_score, 1, 12)
```

Что мывидим? Что на самом деле так и бывает на практике, **метод главных компонент редко дает какой-то прирост качества, но и при этом с небольшим уменьшением качества признаков, качество падает тоже не сильно**.

## 2.5 Многомерное шкалирование

Продолжим изучение методов понижения размерности. И на этот раз мы **рассмотрим методы многократного шкалирования и алгоритм t-SNE**. Итак идея ровно та же: мы знаем, что признаковое пространство может быть очень многочисленным и нам хочется перейти к сжатому пространству, например, для визуализации или для предварительного анализа данных и при этом мы хотим сохранить структуру в данных. Что значит сохранить структуру - в методе многомерного шкалирования и в методе t-SNE определяется по-разному.

Начнем с **метода многомерного шкалирования**. Итак, нам дано исходное признаковое пространств, и представления объектов в этом пространстве, и, самое главное, расстояние между объектами в исходном признаковом пространстве. Мы **хотим для каждого объекта $X_i$ найти такое представление в сжатом признаковом пространстве, что расстояние в исходном будет похоже на расстояние в сжатом признаковом пространстве**. При этом делается оговорка, что расстояние в сжатом признаковом пространстве будет евклидовым. В результате мы получим сжатую матрицу представлений объектов. Естественно сохранить расстояние в исходном и в сжатом признаком в пространстве получается далеко не всегда.

Как работают методы многомерного шкалирования? Обычно они оптимизируют функционал, который называется **стрессом**. В базовом виде стресс можно представить, как сумма квадратов отклонений расстояний в исходным признаковом пространстве и расстояний в полученном признаком пространстве.

$$S(Y) = \sum_{i<j}(\delta_{ij} - d_{ij}(Y))^2$$

Стресс функцию можно модифицировать, то есть либо взять какое-то преобразование исходных расстояний, либо рассматривать относительный стресс.

Для диагностики методом многомерного шкалирования используется **диаграмма стресса**. Здесь на оси $X$ отложены все попарные расстояния в исходном признаковом пространстве, а на оси $Y$ отложены полученные попарные расстояния в сжатом признаковом пространстве. **Таким образом, если на ваших данных алгоритм многомерного шкалирования работает хорошо, то и все точки на этой диаграмме должны быть сконцентрированы на его диагонали**.

Выделяют три основных подхода к решению задачи многомерного шкалирования.

- **Классический** (classical MDS)
- **Метрический** (metric MDS)
- **Неметрический** (non-metric MDS)

Начнем с **классического**. Итак, в нем на задачи накладывается следующее ограничение. Во-первых, исходное расстояние мы будем считать евклидовым. Во-вторых, полученные представления в сжатом признаковом пространстве должны быть центрированы. То есть среднее значение каждого из признаков должно быть равно нулю.

Начинают с того, что рассматривают матрицу граммы. **Матрица Грамма** - это матрица попарных скалярных произведений $B = YY^T$ в сжатом признаковом пространстве. Эта матрица симметрична и положительно определенная, это довольно хорошие свойства матрицы, которую мы будем использовать в дальнейшем.

Кроме того можно показать, что если расстояние между объектами евклидово, то и матрицу Грамма можно выразить по следующей формуле (точка на месте индекса означает суммирование по всем элементам в этом индексе).

$$b_{ij} = -\frac{1}{2}(d_{ij} - d_{i.} - d_{.j} + d_{..}),$$

где $d_{ij} = d(y_i, y_j)$.

Оказывается, **если матрица такая хорошая: симметричная, положительная, определенная, то ее можно диаганолизировать**, то есть представить в виде произведения следующих матриц $B = A\Lambda A^T$, где $A$ - матрица собственных векторов $YY^T$, $\Lambda$ - диагональная матрица с собственными числами $YY^T$ на диагонали.

Отсюда и можно выразить **искомое сжатое представление** $Y = A\Lambda^{\frac{1}{2}}$

Многие могут заметить, что у этого метода есть общие черты с методом, который мы изучали ранее, с методом главных компонент. И действительно, **при данных ограничениях результат применения классического многомерного шкалирования будет идентичен применению метода главных компонент**.

Что касается **неметрического и метрического подхода** к многомерному шкалированию, то здесь уже не накладывается никаких ограничений на исходное расстояние, более того, мы можем брать некоторое монотонное преобразование этих расстояний.

Отличие же метрического и неметрического подхода заключается в том, что **в метрическом подходе мы пытаемся приблизить именно значение попарных расстояний, а в неметрическом подходе нам важен порядок**. То есть, если расстояние в исходном пространстве между объектами $X_i$, $X_j$ будет меньше, чем расстояние между объектами $X_k$, $X_m$, то и в сжатом признаковом пространстве это соотношение должно выполняться.

Обычно это **делается итеративными алгоритмами**, которые работают по одной следующей схеме:

- либо они оптимизируют координату одного объекта при фиксированных положениях остальных объектов,
- либо они перемещают все объекты целиком при фиксированных параметрах преобразования, а потом на втором шаге обновляют параметры самого преобразования минимизируя стресс функцию.

Один из наиболее известных и популярных алгоритмов, который используется в методе многомерного шкалирования, называется **SMACOF - Scaling by MAjorizing a COmplicated Function**, и он также реализован в *sklearn*.

## 2.6 t-SNE

Теперь **рассмотрим метод t-SNE - t-distributed Stocastic Neighbor Embedding**. Это не метод многомерного шкалирования, то есть **полученное расстояние в сжатом признаком в пространстве скорее всего никак не будет соотносится с расстояниями в исходном признаком в пространстве**. Вместо этого метод пытается перенести окрестность каждой точки из исходного пространства в сжатое. Как он это делает?

Давайте посмотрим «в глаза» следующей формуле.

$$p(i|j) = \frac{\exp\{-(x_i - x_j)^2/\sigma_j^2\}}{\sum_{k \neq j} \exp\{-(x_k - x_j)^2/\sigma_j^2\}}$$

Формула кажется довольно тяжелой, но на самом деле она интерпретируется следующим образом: это вероятность выбрать объект $x_i$ при гауссовом распределении с центром в объекте $x_j$ и дисперсией $\sigma_j^2$ в квадрате. $\sigma_j^2$ - это параметр, который явным образом задается пользователем вручную.

Мы также видим, что это мера схожести, она не симметрично. Для того, чтобы ее симметризовать, мы сделаем следующее преобразование:

$$p(i, j) = \frac{p(i|j) + p(j|i)}{2N}$$

Что касается схожести в целевом признаком пространстве, то здесь получается все примерно то же самое, только вместо экспонент у нас возникает формула для распределения стюдента с одной степенью свободы.

$$q(i|j) = \frac{g(|y_i - y_j|)}{\sum_{k \neq j} g(|y_i - y_j|)}$$

где $g(z) = \frac{1}{1 + z^2}$ - t-распределение Стьюдента с 1 степенью свободы.

**В изначальном методе - SNE - и там, и там использовались экспоненты, то есть гауссовское распределение, но оказывается, что это распределение стьюдента работает гораздо лучше**.

Итак, у нас есть 2 вероятностных распределения и нам нужно как-то померить похожесть этих распределений, и желательно сделать одно максимально похожее на другое. Для того, чтобы **измерить похожесть вероятностных распределений** используется мера, которая называется **дивергенция Кульбака-Лейблера**.

$$L(Y) = KL(P||Q) = \sum_i \sum_j p(i, j) \log \frac{p(i, j)}{q(i, j)} \to \min_Y$$

**Чем сильнее отличается одно распределение от другого, тем больше будет величина дивергенции**.

**Как же мы будем находить координаты этих объектов в сжатом признаков пространства**? С помощью метода градиентного спуска. То есть у нас есть единственный неизвестный параметр, это координата каждой точки $y_i$, и по ней мы будем брать градиент, частные производные и обновлять эти координаты.

Несмотря на свою простоту и вроде бы элегантность **метод t-SNE может быть нестабильным, может получить разные результаты при разных изначальных положениях точек и при равном количестве итераций градиентного спуска**. Более того, его стоит применять с опаской.

Во-первых, нельзя никак ориентироваться на размеры полученных кластеров в сжатом признаком в пространстве. Нельзя так же никак интерпретировать расстояние между кластерами в сжатом признаком пространстве. Более того, бывают случаим когда при применении метода t-SNE к полностью шумовым данным, в результате мы можем получить, увидеть вернее, структуру, хотя на самом деле никакой структуры в шуме нет.

## 2.7 Применение t-SNE на данных

Давайте применим алгоритм t-SNE к многомерным данным и посмотрим, как влияет гиперпараметр перплексии на полученный результат.

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

%matplotlib inline

plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (12,5)
```

Данные, которые мы будем использовать - это данные с изображениями цифр, которые по умолчанию уже есть в *sklearn*.

```
from sklearn.manifold import TSNE
from sklearn.datasets import load_digits

digit = load_digits()
```

Получим изображение:

```
img = digit.images
img.shape
```

Видим, что там лежит многомерный массив - много изображений размера 8 x 8 пикселей.

Давайте на какое-нибудь из них посмотрим.

```
plt.imshow(img[0], cmap=plt.cm.Greys_r)
```

Первое изображение у нас соответствует цифре ноль.

Теперь, **для того чтобы подать эти данные на вход алгоритма, нужно представить их в привычном нам виде, то есть в виде матриц объект-признак**. В данном случае каждым признаком здесь будет значение интенсивности в каждом пикселе.

```
X = img.reshape(-1, 64)
```

И теперь уже можно отправлять её в алгоритм t-SNE.

```
tsne = TSNE(n_components=2, perplexity=30.)
tsne.fit(X)
```

А теперь давайте посмотрим на представление объектов в сжатом признаковом пространстве. Для этого обратимсяк атрибуту *embedding*.

```
Z = tsne.embedding_

y = digit.target
plt.scatter(Z[:, 0], Z[:, 1], c=y)
```

Мы видим, что объекты, которые относятся к одним и тем же цифрам, они и в сжатом признаковом пространстве находятся близко друг к другу.

Теперь давайте посмотрим, **на что влияет гиперпараметр перплексии**.

```
permplex = [1, 3, 5, 10, 30, 50]

for p in permplex:
    tsne = TSNE(n_components=2, perplexity=p)
    tsne.fit(X)
    Z = tsne.embedding_
    fit, ax = plt.subplots(1, 1)
    ax.scatter(Z[:, 0], Z[:, 1], c=y, cmap=plt.cm.spectral_r)
    ax.set_title('Perplexity {}'.format(p))
```

Что мы наблюдаем? Что для каких-то небольших значений гиперпараметра перплексии мы наблюдаем очень много небольших кластеров. В то время как, если значение перплексии стоит достаточно большое - 10, 30, 50 - мы видим, что количество кластеров резко падает, но при этом они становятся сами по себе очень большими. Таким образом, перплексия регулирует то, на что именно t-SNE обращает внимание: на локальную структуру или на глобальную структуру. **С увеличением перплексии фокус на глобальную структуру данных становится больше, а с уменьшением, наоборот, меньше**.

# Рекомендательные системы

## 3.1 Рекомендательные системы

Давайте изучим, какая мотивация стоит за рекомендательными системами; как формулируются задачи, которые рекомендательные системы должны решать; а также какие факторы стоит учитывать при построении рекомендательных систем.

Итак, люди — это потребители контента, товаров и услуг. И понятное дело, что каталог этих товаров и услуг очень большой. Очень много фильмов, очень много книг, очень много игр и песен, и все на свете попробовать невозможно. Соотвественно, хочется как-то фильтровать этот каталог. Как это можно делать?

Если у вас есть друзья, вы можете поинтересоваться их мнением. Проблема заключается в том, что вкусы вас и ваших друзей, скорей всего, будут отличаться. Можно заморочиться и почитать обзоры на нужную тему. Но. скорей всего, вы потратите на это очень много времени, а результат может быть неудовлетворительным. В конечном счете хотелось бы иметь некоторую систему, которая учитывала бы ваши вкусы, и на основе этого строила бы рекомендации. То есть нужна рекомендательная система.

Немножко погрузимся в историю. Оказывается, что рекомендательная система — это довольно молодая область. Где-то в 2006 году стартовал конкурс по анализу данных, который назывался Netflix Prize, который проводила компания Netflix. Призовой фонд был 1 млн долларов, и участники должны были восстанавливать рейтинги пользователей к фильмам. Рейтинг выражался в виде звездочки, от одной звездочки до пяти звездочек. И ошибка (мера качества в этом конкурсе) измерялась как среднее квадратичное отклонение предсказания рейтинга от фактического значения рейтинга. Конкурс шел довольно долго, и где-то по прошествии трех лет только удалось выявить победителя этого конкурса. Где-то одновременно с началом Netflix Prize началась запускаться ежегодная международная конференция по рекомендательным системам RecSys, которая идет до сих пор и каждый раз являет новые и новые продвижения в мире рекомендательных систем.

Итак, как все выглядит с формальной точки зрения? У нас есть пользователи, есть товары, и пользователи ставят оценку товару. Оценка может быть разной. Она может быть бинарной, например: «нравится — не нравится» или «купил — не купил». Она может быть в виде рейтинга, то есть количество звезд. Или это может быть какая-то неявная оценка, например количество потраченного времени на просмотр фильма. Так или иначе, мы переходим к такой матрице пользователей на товары. Соотвественно, на пересечении строки и столбца будет стоять оценка рейтинга. Но на самом деле в большей части этих ячеек рейтинга стоять никакого не будет, потому что опять же люди не могут посмотреть все на свете и проставить какой-то свой респонс по поводу товара.

Таким образом, первой **задачей рекомендательной системы является заполнение этих самых пропусков в этой матрице**.

А вторая задача — это **каждому пользователю предоставить релевантную его интересам рекомендацию**.

И оказывается, что **эти две задачи в некотором смысле независимы друг от друга**, то есть хорошее восстановление рейтингов далеко не всегда дает хорошую рекомендацию пользователю. Более того, при рекомендации иногда можно учитывать некоторое экономическое предпочтение продавцов, то есть людей, которые владеют этими товарами и услугами и продают их. Иногда им хочется в рекомендациях давать товары, которые имеют большую ценность, соответственно, повышать свою маржу.

При построении рекомендательных систем также стоит учитывать такой эффект, как **Learning Loop**. Это когда дальнейшее обучение рекомендательных систем основывается на рекомендациях, которые были произведены ранее. И таким образом, система не изучает никаких новых зависимостей, зацикливается сама в себе и не дает нужного результата в будущем. И такая **довольно известная проблема — это тривиальные рекомендации**, то есть рекомендации, которые основаны на каких-то бестселлерах, которые и так люди все знают и нечего их рекомендовать.

Какие другие важные факторы стоит учитывать при построении рекомендательных систем? Конечно же, это **проблема холодного старта**. Она возникает когда на портал заходит новый пользователь, который еще не проставлял никаких оценок фильму, или новый товар, про который тоже ничего не известно.

**Масштабируемость**. Это проблема, которая возникает из-за того, что на портале становится очень много пользователей и товаров, и, соответственно, все вычисления нужно производить очень быстро, поэтому нам нужна какая-то модель, которая будет работать очень быстро.

**Накручивание рейтингов** — это проблема, которая возникает из-за того, что некоторые силы влияют на проставление рейтингов к товарам, и таким образом они заставляют рекомендательные системы показывать эти товары в рекомендациях пользователям, хотя на самом деле товар может быть так себе.

**Неактивные пользователи** — это проблема, которая возникает, когда человек один раз зашел на портал, что-то один раз оценил или купил и больше никогда там не появлялся. Соответственно, это замедляет и засоряет результаты, такие моменты стоит фильтровать.

И есть другие проблемы, которые стоит тоже учитывать.

Какие же подходы к построению рекомендательных систем можно выделить?

Во-первых, это **методы коллаборативной фильтрации**. Если в двух словах, это методы, основанные на том, чтобы восстанавливать рейтинги пользователей на основе других пользователей, которые на них наибольше похожи.

Второй подход — это **латентные модели, латентные методы**. Опять же, если в двух словах, то в сущности пользователи и товар в этих методах переходят в некоторое скрытое признаковое пространство, при этом пытаются сохранить информацию об исходных рейтингах, которые пользователи товарам проставили. Более подробно мы их изучим дальше.

## 3.2 Методы коллаборативной фильтрации

Коллаборативная фильтрация делится на два подхода: user-based и item-based.

**User-based, или подход на основе похожих пользователей** заключается в том, что пользователи с похожими интересами будут покупать похожие товары.

В подходе **item-based**, мы отталкиваемся от товаров, то есть мы ищем товары, которые похожи на те, которые нравятся пользователю.

Нам для начала нужно ввести некоторые обозначения.

- Через $U$ мы будем обозначать множество пользователей;
- $I$ - множество товаров;
- $U_i$ - множество пользователей оценивших товар $i$;
- $I_u$ - множество товаров, оцененных пользователем $u$;
- $R_{ui}$ - это оценка, которую на самом деле дал пользователь $u$ товару $i$;
- $\hat{R_{ui}}$ - это прогноз этой оценки.

Итак, **user-based CF (Collaborative Filtering)**. Нам нужно посчитать некоторое сходство между пользователями $s$. Дальше, предположим мы хотим спрогнозировать рейтинг или оценку пользователя $u$ для товара $i$, который он раньше не купил, не посмотрел, не оценил. Далее мы находим пользователей, которые во-первых купили, посмотрели или оценили товар $i$ и среди них выбираем наиболее похожих на наши текущего пользователя пользователей $N(u)$. А дальше прогнозирование рейтинга делается как взвешенное среднее оценок этих похожих пользователей с учетом уровня их похожести.

$$\hat{R}_{ui} = \frac{\sum_{\nu \in N(u)} s_{u\nu} R_{\nu i}}{\sum_{\nu \in N(u)} |s_{u\nu}|}$$

Эту формулу также можно модифицировать, отличается тем, что из исходных рейтингов пользователей мы вычитаем их среднюю оценку. То есть, возникает некоторая поправка на оптимизм или пессимизм пользователя. Некоторые люди излишне пессимистичные, некоторые излишне оптимистичны. Эта формула призвана этот эффект сгладить.

**Как же выбирать множество пользователей наиболее похожих на текущего пользователя**?

Мы можем, во-первых брать всех пользователей. Этого никто не отменял. Мы можем заранее ограничивать количество наиболее похожих этих пользователей некоторым числом K. Ну или можем просто придумать какой-то порог на меру схожести между двумя пользователями.

**Как же можно мерить схожесть между двумя пользователями**? Два наиболее распространенных варианта, это корреляция или косинусная мера. В принципе формулы довольно каноничны. Единственное отличие заключается в том, что суммирование происходит не по всем товарам, а только по тем товарам, которые оба пользователя успели купить или оценить.

Что касается **item-based коллаборативной фильтрации**, то здесь всё ровно то же самое, только мы отталкиваемся от товаров.

Мы также считаем схожесть между товарами. Помимо уже изученных мер схожести: косинусные меры и корреляции, можно использовать более специфические. Например, вероятность того, что товар G, будет куплен вместе с товаром I, или некоторую зависимость совместной покупки товара I и G к независимой покупке товара I и товара G.

Таким образом мы с вами изучили два подхода к коллаборативной фильтрации. Это user-based CF и item-based CF. Стоит еще раз отметить, что **это эвристические подходы, то есть они не имеют никаких теоретических обоснований**. Мы лишь проверяем некоторую гипотезу, о том что похожие пользователи будут покупать похожие товары. Эту гипотезу, этот тот алгоритм легко реализовать, он довольно понятный, но тем не менее, его **реализация требует очень много вычислений**. То есть нужно постоянно хранить матрицу рейтингов и считать похожести для пользователей или для товаров, что может занять довольно много времени.

## 3.3 Методы с матричными разложениями

Рассмотрим **модели с латентными переменными**. Итак, идея этих методов заключается в том, что каждому пользователю и товару будет соответствовать некоторый вектор. Для пользователей мы будем его обозначать через $\vec{p}_u$, а для товаров - через $\vec{q}_i$. Вектор имеет фиксированную длину и выбирается не просто так, а таким образом, что их скалярное произведение должно быть похоже на оценку для соответствующего товара, соответствующим пользователем.

$\vec{p}_u$ иногда получается интерпретировать как заинтересованность пользователя в определенных категориях товаров, а $\vec{q}_i$ как принадлежность этих товаров к определенным категориям. Естественно, сделать это получается далеко не всегда.

Кроме того, это латентное пространство можно использовать для того, чтобы находить похожих пользователей или похожих товаров. Схематично это можно представить следующим образом. У нас есть исходная матрица рейтингов пользователей на товары и мы хотим ее преобразовать в виде произведения двух других матриц: матриц векторов пользователей и матрицы с векторами товаров.

Первая модель называется **Latent Factor Model**, оптимизирует она следующий функционал качества.

$$\sum_{u,i}(R_{ui} - \bar{R}_u - \langle \vec{p}_u, \vec{q}_i \rangle)^2 \to \min_{P,Q}$$

Как находятся собственно компоненты $\vec{p}_u$ и $\vec{q}_i$? С помощью метода стохастического градиентного спуска. То есть, на каждом шаге, мы случайным образом выбираем пару пользователей товар и обновляет координату каждого из этих векторов по методу градиентного спуска. Стоит отметить, что невязка считается только для тех элементов, для которых нам известно значение в матрице рейтингов. Для всех остальных мы ничего никак не можем посчитать.

Кроме того, эта **модель является довольно гибкой, то есть в функционал качества можно также добавить регуляризацию** на $\vec{p}_u$ и $\vec{q}_i$.

Вернемся к схематичному предоставлению латентных моделей. Многие могут заметить, что нечто подобное мы с вами наблюдали ранее, в частности, сингулярное разложение. Только там мы раскладывали нашу матрицу на произведения 3 матриц.

Единственная проблема с которой мы тут столкнемся: в нашей матрице рейтинга всё-таки есть пропущенные значения. Один из самых распространенных способов - это заменить пропущенные значения на ноль. Таким образом наша матрица получится очень сильно разреженная. Это будет нам только на руку, на самом деле, так как можно использовать некоторые высокоэффективные библиотеки для вычисления сингулярного разложения для разреженных матриц.

Если возвращаться к нотации по P и Q, то $P = U\Sigma^{\frac{1}{2}}$, $Q = \Sigma^{\frac{1}{2}}V^T$.

Кроме того, есть факторизация, которая называется **неотрицательной матричной факторизацией**. По сути это то же самое, только на матрицы P и Q накладывается условие неотрицательности. То есть, каждый элемент этой матрицы должен быть больше или равен нулю.

Таким образом мы с вами рассмотрели модели с латентными переменными, некоторые из которых связаны с настоящими матричными разложениями. **Обычно эти модели работают лучше и быстрее чем коллаборативная фильтрация**. Если вам интересна эта тема, вы можете дополнительно изучить метод факторизационных машин и тензорные разложение.

## 3.4 Матрица рейтингов и SVD

Теперь научимся получать матрицу рейтингов из исходных данных, применять метод сингулярного разложения к матрице рейтингов и считать похожесть пользователей по матрице рейтингов.

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm_notebook

%matplotlib inline

plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (12, 6)
```

Изначально у нас есть две таблицы. Первая таблица — это таблица с рейтингами, то есть у нас есть поля «Идентификатор пользователя», «Идентификатор фильма», «Рейтинг» и «Время, когда рейтинг был проставлен». Есть таблица с фильмами, то есть у нас есть «идентификатор фильма», «название фильма» и прочая информация про этот фильм.

```
filepath = './data/user_ratedmovies.dat'
df_rates = pd.read_csv(filepath, sep='\t')

filepath = './data/movies.dat'
df_movies = pd.read_csv(filepath, sep='\t', encoding='iso-8859-1')

df_rates.head()
df_movies.head()
```

Первое, что стоит заметить, - это то, **как распределены идентификаторы фильмов и пользователей**. Вначале, если посмотреть на минимальное и максимальное значения, то кажется, что у нас должно быть довольно много пользователей.

```
from sklearn.preprocessing import LabelEncoder

df_rates.userID.min(), df_rates.userID.max()
```

Но если просто посчитать количество уникальных значений, то их оказывается не так много.

```
df_rates.userID.nunique()
```

В дальнейшем **для удобства лучше кодировать эти идентификаторы числами от нуля до количества уникальных значений**, что для пользователя, что для фильмов. Делать мы это будем с помощью преобразователя, который называется *LabelEncoder*, он есть в *sklearn*.

```
enc_user = enc_user.fit(df_rates.userID.values)
enc_mov = enc_mov.fit(df_rates.movieID.values)
```

Мы для начала создадим экземпляр класса и обучим их. Обучать мы их будем на матрице рейтингов.

```
enc_user = enc_user.fit(df_rates.userID.values)
enc_mov = enc_mov.fit(df_rates.movieID.values)
```

И перед как мы сделаем переобозначение, еще из матрицы с фильмами удалим те фильмы, которые не встречаются в матрице рейтингов, просто для удобства.

```
idx = df_movies.loc[:, 'id'].isin(df_rates.movieID)
df_movies = df_movies.loc[idx]
```

Теперь **сделаем переобозначение**.

```
df_rates.loc[:, 'userID'] = enc_user.transform(df_rates.loc[:, 'userID'].values)
df_rates.loc[:, 'movieID'] = enc_mov.transform(df_rates.loc[:, 'movieID'].values)
df_movies.loc[:, 'id'] = enc_mov.transform(df_movies.loc[:, 'id'].values)

df_rates.head()
```

Теперь давайте **создадим саму матрицу**. Делать мы это будем с помощью разреженных матриц. Есть много разных форматов разреженных матриц, мы будем использовать так называемый координатный формат, то есть для того чтобы ее инициализировать, нужно указывать значение и координаты у матрицы по столбцам и строкам, где это значение должно храниться.

```
from scipy.sparse import coo_matrix, csr_matrix

R = coo_matrix((df_rates.rating.values, (df_rates.userID.values, df_rates.movieID.values)))
```

Теперь давайте **применим к этой матрице сингулярное разложение**. На разреженных матрицах есть специальный алгоритм *svds*, он лежит в библиотеке *SciPy*. Выход абсолютно такой же, как и в обычном сингулярном разложении на плотной матрице.

```
from scipy.sparse.linalg import svds

u, s, vt = svds(R, k=6)
```

Давайте в сжатом признаковом пространстве **для каждого фильма найдем десять его ближайших соседей и посмотрим, как это соотносится с реальной жизнью**. Для того чтобы найти ближайших соседей, будем использовать класс *NearestNeighbors*.

```
from sklearn.neighbors import NearestNeighbors

nn = NearestNeighbors(n_neighbors=10)
```

Чтобы **получить индексы ближайших соседей**, воспользуюемся методом *kneighbors*.

```
v = vt.T
nn.fit(v)

_, ind = nn.kneighbors(v, n_neighbors=10)
ind[:10]
```

Теперь в матрице *ind* содержатся индексы десяти ближайших соседей для каждой точки. Естественно, каждый фильм будет ближайший самому себе, поэтому первый столбец у нас такой особенный.

Теперь для интерпретации **достанем названия этих фильмов**, чтобы понять, действительно ли они такие близкие и действительно ли это соответствует реальной жизни.

```
movie_titles = df_movies.sort_values('id').loc[:, 'title'].values

cols = ['movie'] + ['nn_{}'.format(i) for i in range(1,10)]
df_ind_nn = pd.DataFrame(data=movie_titles[ind], columns=cols)
df_ind_nn.head()
```

Давайте посмотрим. Возьмем первый фильм - «История игрушек» («Toy story»), и все похожие на него ближайшие соседи - это действительно какие-то мультики, то есть вроде как похоже на правду.

Можно посмотреть, что будет с фильмом «Терминатор».

```
idx = df_ind_nn.movie.str.contains('Terminator')
df_ind_nn.loc[idx].head()
```

Теперь давайте научимся считать **похожесть между пользователями по матрице рейтингов**. Конечно, можно схитрить и из *sklearn* достать метод, который считает косинусное расстояние по заданной матрице. Сделает это он довольно быстро. В итоге мы получим матрицу попарных косинусных схожеств. Размером этой матрицы будет количество пользователей на количество пользователей.

```
from sklearn.metrics.pairwise import cosine_similarity as cosine_similarity

D = cosine_similarity(R)
```

Но мы помним, что **похожесть можно считать, только используя те фильмы, которые пользователь $u$ и пользователь $v$ посмотрели вместе**. То есть нам нужно как-то скорректировать эту меру схожести. Сделать это можно следующим образом.

```
from scipy.spatial.distance import cosine
from scipy.spatial.distance import pdist
from scipy.spatial.distance import squareform
from sklearn.metrics import pairwise_distances

def similarity(u, v):
    idx = (u != 0) & (v != 0)
    if np.any(idx):
        sim = -cosine(u[idx], v[idx])+1
        return sim
    else:
        return 0
```

Теперь для того чтобы посчитать такое вот наше кастомное косинусное расстояние, можно использовать метод *pdist*.

```
d = pdist(R.toarray(), metric=similarity)
```

Посмотрим, что лежит в переменной d. Это такой длинный вектор. Но для того чтобы перевести его в привычный вид, мы его прогоним через метод *squareform*.

```
D = squareform(d)
```

И вот на выходе у нас уже та же самая матрица попарных косинусных похожестей между объектами.