

Векторные представления слов

Цель занятия

В результате обучения на этой неделе:

- вы узнаете о формальной постановке задачи обработки естественного языка;
- научитесь предобрабатывать текстовые данные и извлекать из них признаки;
- научитесь строить информативные векторные представления слов.

План занятия

1. [Введение в NLP](#)
2. [Предварительная обработка текста](#)
3. [Извлечение признаков](#)
4. [Векторное представление слов \(Word Embeddings\)](#)
5. [Векторные представления слов. Визуализация. \(Word embeddings visualization\)](#)
6. [Визуализация в Python](#)

Конспект занятия

1. Введение в NLP

Поговорим о способах представления данных (слов) в некотором машиночитаемом векторном виде. То есть научимся строить информативное векторное представление — embedding.

NLP (Natural Language Processing, обработка естественного языка) — это одна из трех наиболее развивающихся областей искусственного интеллекта. Кроме NLP сейчас активно развиваются компьютерное зрение CV (Computer Vision) и обучение с подкреплением (RL).

Примеры применения:

- Все переводчики базируются на NLP.

- Классификации изображений.
- Нейросетевые доработки для фотографий.

Общие задачи NLP включают все задачи, в которых рассматриваются текстовые данные на входе. А также распознавание и генерацию речи.

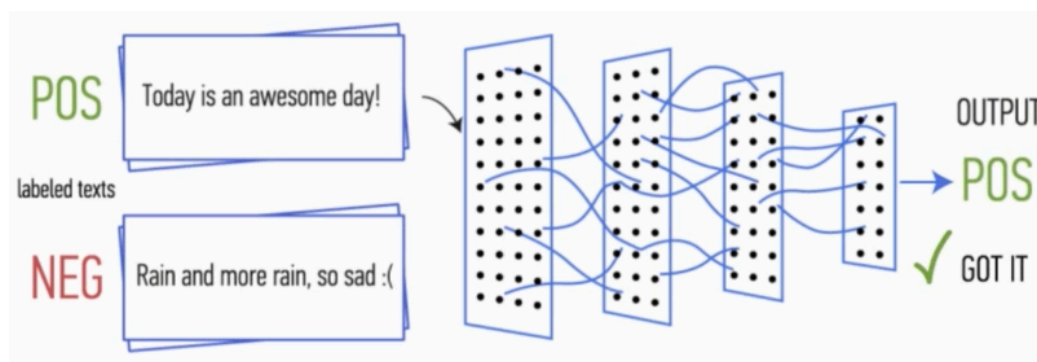
Мы рассмотрим следующие текстовые задачи NLP:

- Анализ сантимерта — анализ реакции человека по комментариям и высказываниям.
- Фильтрация спама. Ранее фильтрация спама происходила по механизму наивного байесовского классификатора вместе с «мешком слов».
- Детекция фейковых новостей.
- Предсказание темы.
- Предсказание хэштегов.

Это подмножество задач — задачи классификации текста.

Задача анализа сантимерта

Анализ сантимерта — анализ эмоциональной, тональной окраски текста.



Визуализация нейронной сети для анализа текста:

На вход нейронной сети поступает текст позитивного или негативного окраса, на выходе получаем предсказание метки класса.

Виды предсказаний текста

Предсказания текста могут быть нескольких видов:

- Бинарная классификация:
 - спам-фильтр;
 - анализ сентимента.
- Мультиклассовая классификация — например, категория продукта по его описанию.
- Классификация multi-label — предсказание хештега.

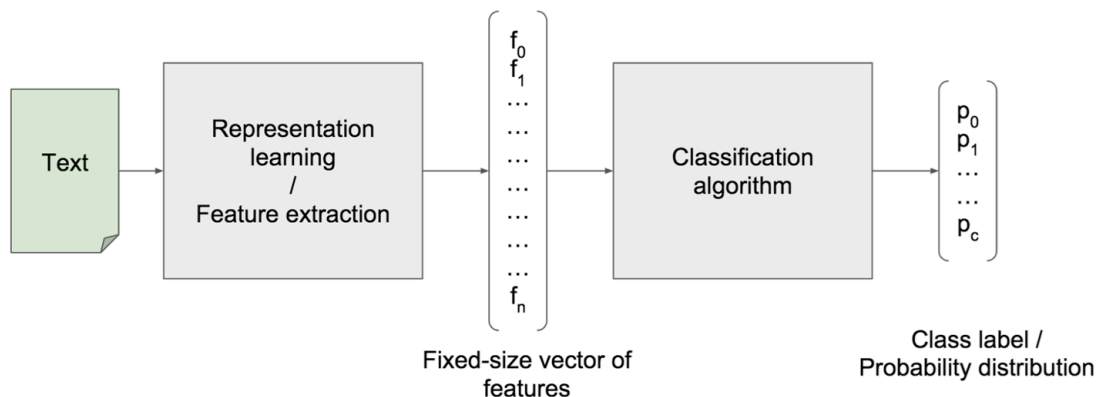
Нельзя путать мультиклассовую классификацию и multi-label. Для каждого твита или фотографии может присутствовать сразу несколько хештегов, каждый из которых появляется независимо от других. В мультиклассовой классификации мы выбираем одну категорию из фиксированного множества.

Существуют задачи, которые стоят на пересечении NLP и CV. Например:

- Генерация описания фотографии. Здесь нужно распознать изображение и сгенерировать текст.
- Составление текстового интерфейса для компьютерной игры. Взаимодействие с игрой — RL, генерация текстового интерфейса — NLP.

Общая классификация текста

Общий пайплайн работы с текстом, изображениями, видео:



На вход поступают данные, модель извлекает некоторые хорошие признаки и представляет их в виде вектора. Он проходит через классификатор (линейный классификатор, логистическая регрессия и т. д.), и мы получаем ответ. Например, в задаче классификации это вектор вероятностей, который гораздо информативней, чем просто метка класса.

При хорошем признаковом описании любой простой классификатор будет работать. Главный вопрос — как именно построить качественное признаковое описание.

Дальше рассмотрим, как от слова (последовательности букв) перейти к векторному представлению, чтобы вектор все еще хранил смысл слова.

2. Предварительная обработка текста

Когда мы переходим к работе с текстом на низком уровне, встает вопрос, как вообще объяснить машине, что такое слова, буквы и смысл слова. Для машины текст — последовательность символов. Для нее не понятно, что такое слово, она «видит» строку.

Токенизация — процедура разбиения данных на токены, в нашем случае это разбиение строки на слова, слоги, буквы, то есть минимальный «атомарный» уровень работы с текстом. Для машины слова нужно привести в нормальную форму. Иначе слова «собака» и «собаки» — это два разных слова. Пока мы не научились строить признаковые описания для однокоренных слов, нужно научиться нормировать слова.

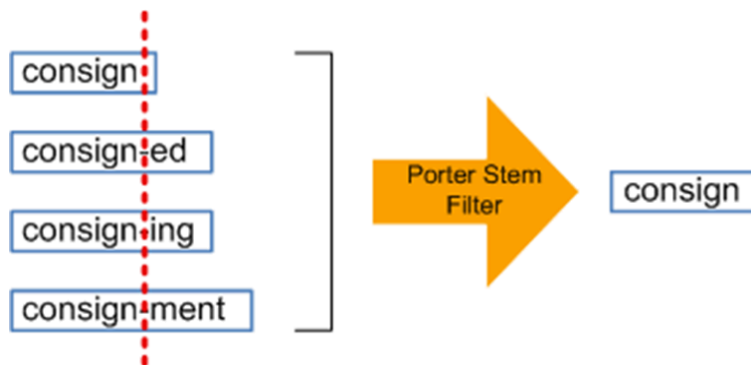
Строку можно разбить на разные элементы — слова, слоги, символы. Общие механизмы обработки будут сохраняться, но при этом у отдельного символа своего

смысла нет, у отдельного слога он уже может быть, у морфемы смысл уже есть, у пары слов смысла еще больше. Различные уровни разбиения текста на подмножества по-разному влияют на смысловую составляющую. Поэтому специалистами были предложены токены — атомарный уровень, на который мы бьем текст. Токен в виде слова, слога, символа, морфемы — это вопрос договоренности в той задаче, которую решаем на данный момент.

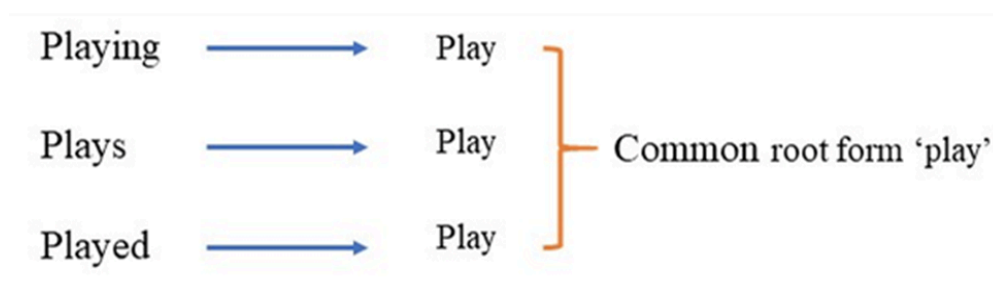
Пусть у нас есть некоторый текст. Разобьем его пока по пробелам, знакам препинания и переносам строк, то есть на слова.

Рассмотрим несколько подходов приведения слов в нормальную форму.

Стемминг (Stemming) — это отсечение с английского.



Лематизация. Для каждого слова хранится словарь, который позволяет из какой-то формы слова получить нормальную форму:



В случае лематизации могут быть какие-то формы слова, которые не попали в словарь, тогда мы не сможем привести в нормальную форму.

Стемминг

Существует множество подходов стемминга:

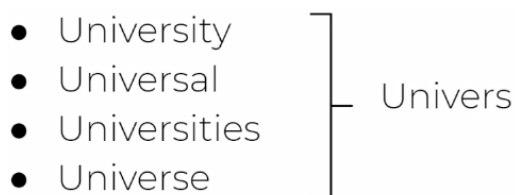
1. Porter stemmer. Был предложен в 1979 году.
2. Lancaster stemmer. Появился 1990 году. Очень агрессивен, отрубал большие кусочки слова, но достаточно прост для работы, поэтому был популярен.
3. Snowball stemmer (Porter 2). Наиболее популярный сейчас стемминг. Описывает почти те же самые правила, что и Porter stemmer, но работает немного по-другому.

Как работает Porter stemmer. Пусть у нас есть набор слов. Для каждого слова предлагаются следующие правила:

- Если видим последовательность SSES, то заменяем на SS (dresses – dress),
- IES – I (ponies – poni),
- S – <empty> (dogs – dog).

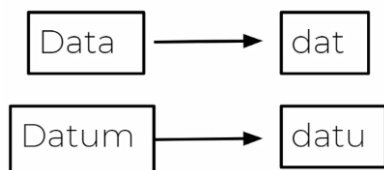
Проблемы подходов стемминга:

- **Проблема оверстемминга (overstemming).** Например:



Все слова будут приведены к слову Univers, хотя они имеют совершенно различные значения.

- **Андерстемминг (understemming).** Например:



Здесь одно и то же слово в единственном и множественном числе может быть приведено к разным нормализациям.

Лематизация

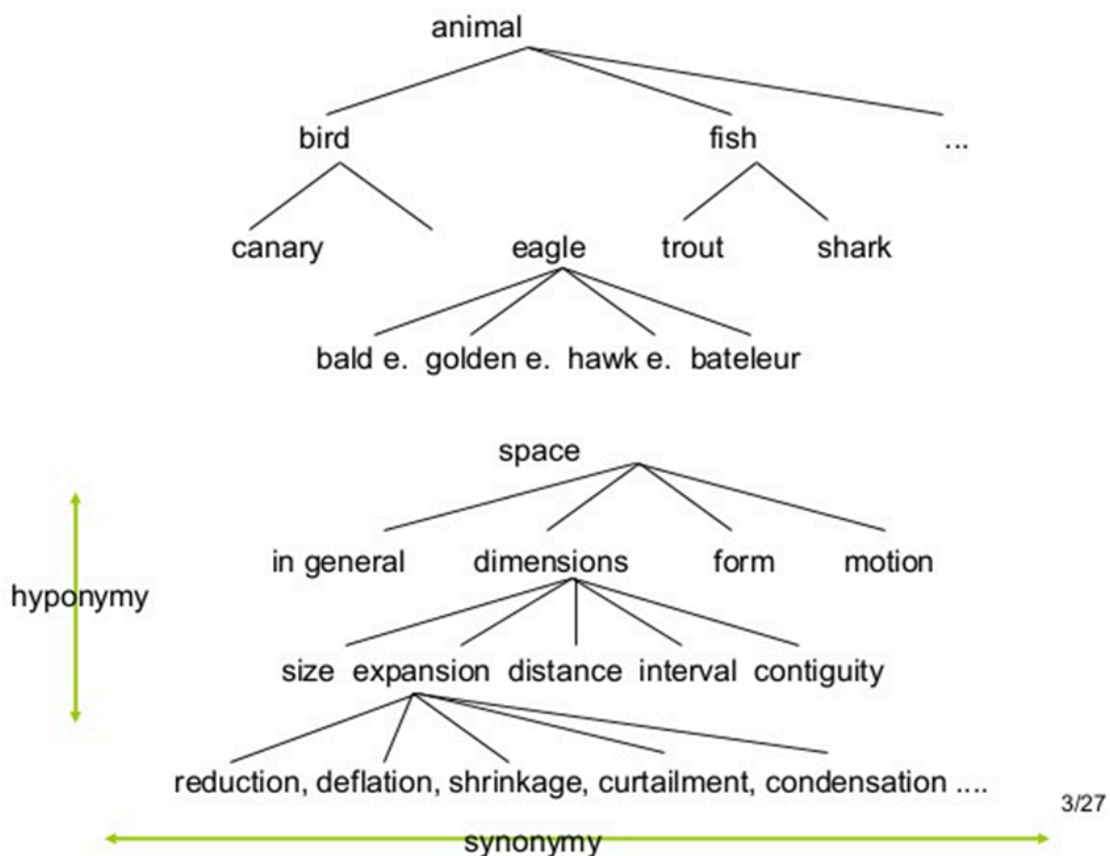
Для лематизации тоже существует много проблем. Нужно построить весь словарь со всеми возможными формами слов. Язык постоянно развивается, и в какой-то момент словарь может устареть, с ним будет сложно работать.

Для лематизации есть множество разных инструментов. Один из таких инструментов – NLTK (National Language Tool Kit), обработка естественного языка. Это отличный вариант для нормализации английского текста.

Лематизаторы стоят на **WordNet** базах данных.

WordNet

WordNet — начинание лингвистов, которые построили информативное представление языка, в котором не просто описание слов, а описание слов между собой. В результате был построен огромный граф:



3/27

По такому графу можно понять определение слова и иерархическое взаимодействие слов. Можно найти синонимы, антонимы, близкие по смыслу слова.

Несмотря на то что это был прорывной проект, было множество проблем. Граф у WordNet статический. А язык — принципиально динамическая сущность. Поэтому граф приходилось постоянно вручную дорабатывать. Это огромный труд. Поэтому поддерживать WordNet крайне дорого.

И более того, такие графы нужно создавать для каждого языка отдельно. Для разных языков есть различный объем ресурсов, который можно потратить.

Тем не менее WordNet качественно продвинул обработку естественного языка вплоть до 2000-х.

Инструменты для предобработки языка

- [NLTK](#):
 - `nltk.stem.SnowballStemmer`,
 - `nltk.stem.PorterStemmer`,
 - `nltk.stem.WordNetLemmatizer`,
 - `nltk.corpus.stopwords`.
- [BeautifulSoup](#) (for parsing HTML) — позволяет работать в HTML файлами.
- Regular Expressions (`import re`) — механизм регулярных выражений.
- [Pymorphy2](#) — библиотека Python.

Как же все-таки работать с текстом, если рассматривать построение признаков описаний для слов?

При разбиении текстов на токены мы не рассмотрели еще некоторые вопросы:

- Что делать с большими буквами? Для простоты все можно привести к нижнему регистру, но тогда могут пострадать имена собственные.
- Пунктуация. Можно считать, что пунктуация — отдельные токены.
- Сокращения (т. е., и т. д., и др.). Их нужно либо разворачивать некоторым словарем, либо считать отдельные сокращения отдельными токенами.
- Числа (даты, номера ID, номера страниц).

- Стоп-слова, артикли (the, is и т. д.) имеют смысл только в сочетании с другими словами, сами по себе смысл не несут. Чаще всего их просто выкидывают при обработке текста на первичном уровне.
- Тэги.

3. Извлечение признаков

Когда мы говорим о построении признакового описания для слов, мы говорим о представлении слова (токена) в виде вектора. Какой простейший способ представить слово в виде вектора? Конечно, количество слов в языке, то есть некоторый словарь фиксированной размерности.

Мешок слов

Простейшее представление слова в виде вектора — one-hot вектор. То есть вектор, в котором одна единица стоит на месте, соответствующем порядковому номеру слова в словаре, а на всех остальных стоят нули. Отсюда можно придумать представление всего текста в виде вектора.

Мы можем все one-hot векторы между собой просуммировать и получить один вектор размерности словаря, и теперь он будет описывать заданный текст. Такой подход называется **мешок слов (bag-of-words)**.

the dog is on the table

0	0	1	1	0	1	1	1
are	cat	dog	is	now	on	table	the

У этого подхода по векторизации из слов есть недостатки:

- Потерян порядок слов, а порядок слов может быть важен.
- Векторы, которые описывают слова и текст в целом, огромные. Размерность словаря популярного языка примерно 10 тыс. слов. И такие векторы будут очень сильно разреженными, это не очень удобно.

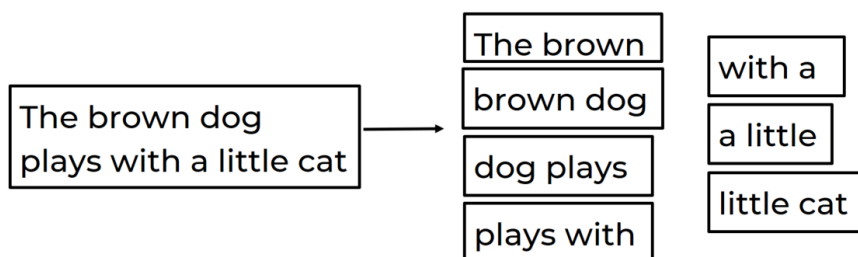
- Векторы ненормированные для текста в целом.

Метод мешка слов позволял использовать наивный байесовский классификатор для фильтрации спама. Пришедшее на почту письмо разбивали на токены, приводили в нормальную форму, и каждый признак рассматривался отдельно от всех остальных. Наивный байесовский классификатор с правильным априорным распределением слов работал довольно неплохо.

Мешок слов работал на простых задачах.

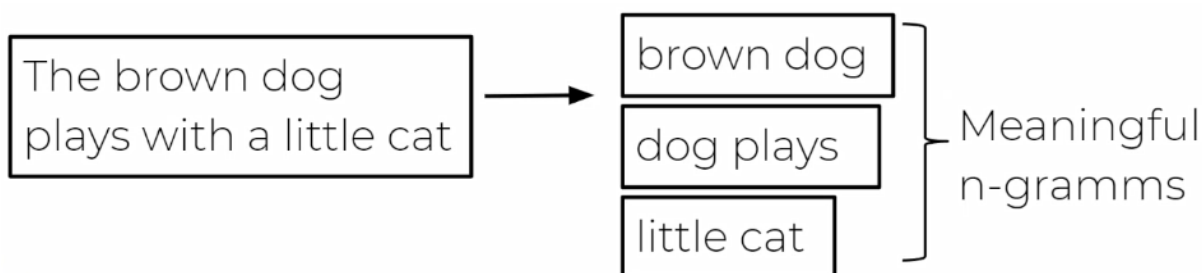
Как улучшить мешок слов

Теряя порядок слов, мы теряем еще больше информации. Можно попытаться сделать мешок слов более информативным. Попробуем вместо мешка слов использовать мешок пар слов.



Теперь можно улавливать порядок слов в парах. Но теперь размерность словаря из пар слов станет квадратичной.

Решение: зачем нам использовать все биграммы (n-грамма — последовательность из n слов)? Некоторые из них не несут смысла. Человечество придумало различные эвристики по выделению информативных n-грамм, которые в дальнейшем назвали **колокациями**.



В примере получаем три информативные биграммы.

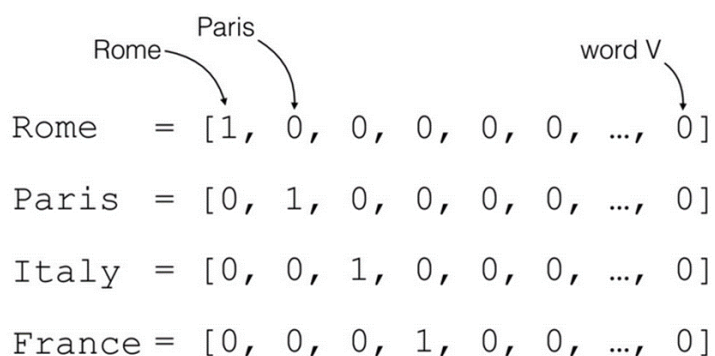
Конечно, для эвристик нет четких правил.

Рекуррентные нейронные сети способны улавливать порядок слов. В этом случае нужно построить информативное признаковое описание для каждого из слов.

4. Векторное представление слов (Word Embeddings)

One-hot вектор

Повторим концепцию one-hot вектора на примере.



Rome = [1, 0, 0, 0, 0, 0, ..., 0]

Paris = [0, 1, 0, 0, 0, 0, ..., 0]

Italy = [0, 0, 1, 0, 0, 0, ..., 0]

France = [0, 0, 0, 1, 0, 0, ..., 0]

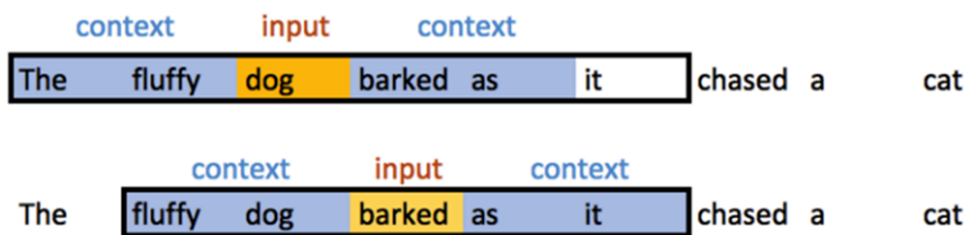
Для каждого из этих векторов можно утверждать:

- векторы очень большие;
- очень разреженные;
- у всех норма одинаковая;
- расстояние между векторами одинаковое, хотя должна быть корреляция между словами по смыслу.

Дистрибутивная семантика

Конечно, контекст влияет на слово, а слово влияет на контекст. Каждое слово может быть описано тем, в каком контексте оно находится.

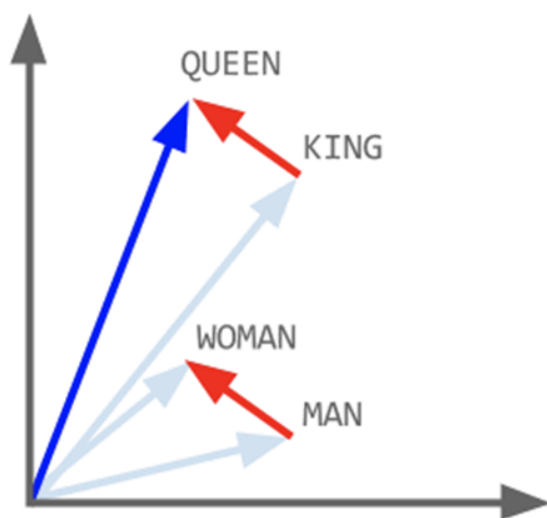
Введем понятие **скип-граммы** — последовательности токенов, где центральный токен выколот.



То есть попытаемся формально описать локальный контекст — соседей, которые окружают данное слово, предполагая, что именно ближайшие соседи в первую очередь влияют на выбранное слово.

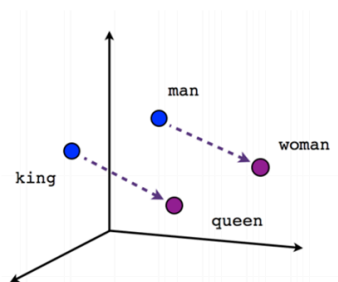
Эта идея позволит нам выучить векторное представление слов, и для каждого слова появится пара — слово и контекст, в котором слово находится.

So king = man + woman = queen!

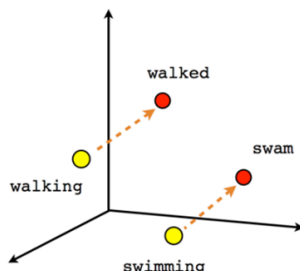


Word2vec

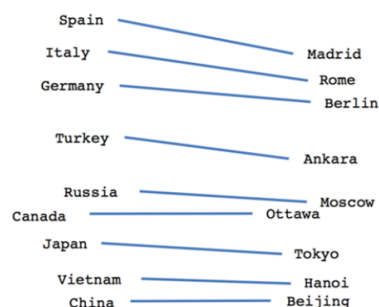
Word2vec позволяет построить векторное представление слов, где между векторами сохраняются семантические связи.



Male-Female



Verb tense



Country-Capital

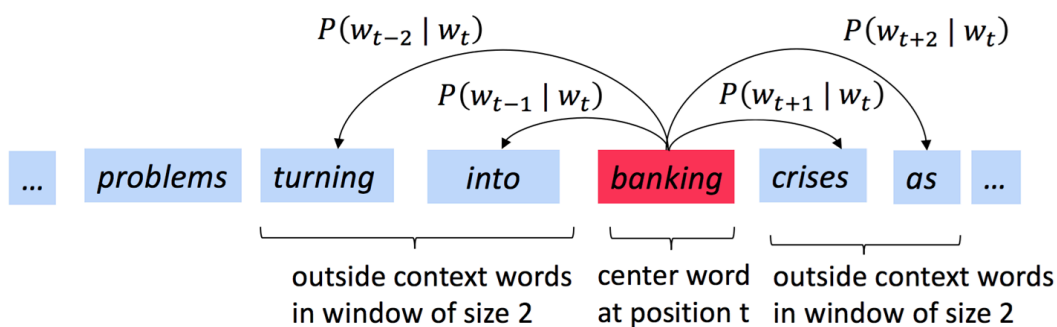
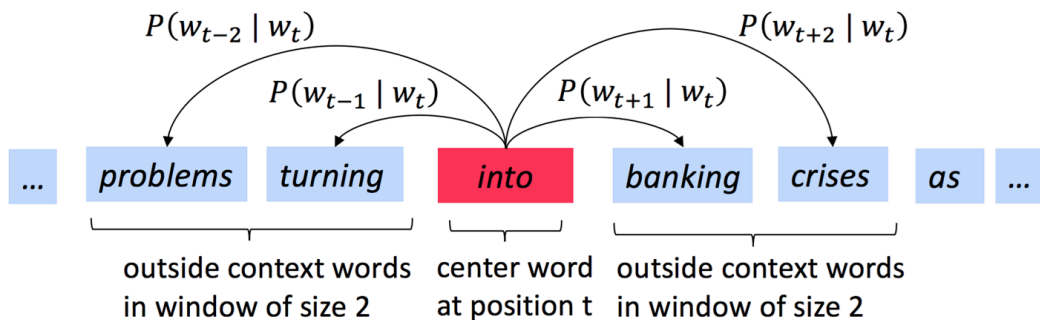
Пусть у нас есть текст, и теперь для каждого слова мы можем выбрать само слово и его локальный контекст. Получаем обучающую выборку:

Source Text	Training Samples
The quick brown fox jumps over the lazy dog. →	(the, quick) (the, brown)
The quick brown fox jumps over the lazy dog. →	(quick, the) (quick, brown) (quick, fox)
The quick brown fox jumps over the lazy dog. →	(brown, the) (brown, quick) (brown, fox) (brown, jumps)
The quick brown fox jumps over the lazy dog. →	(fox, quick) (fox, brown) (fox, jumps) (fox, over)

Мы можем породить обучающую выборку из всех текстов, доступных на данном языке. Для любого языка количество текстов огромное.

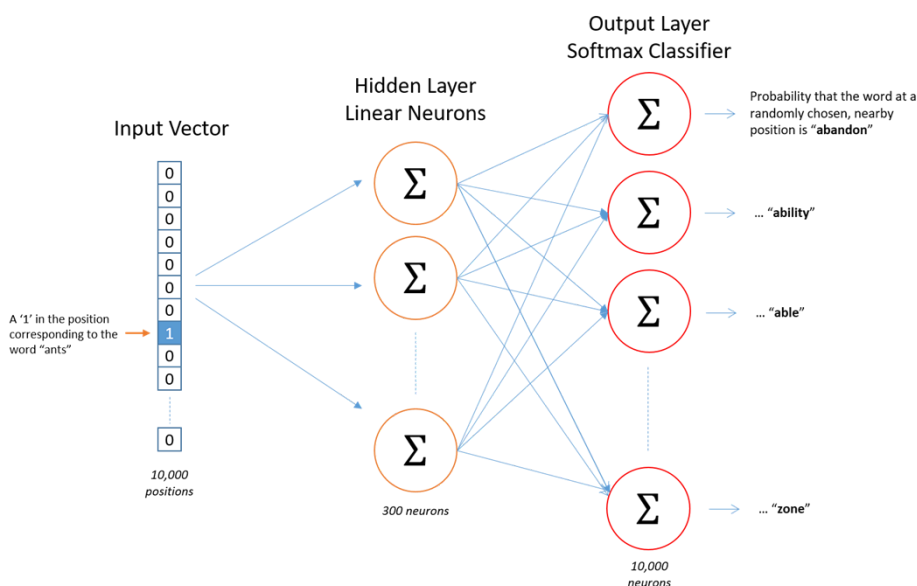
На полученной обучающей выборке будем тренировать классификатор — обучать его предсказывать, какие слова могут оказаться в контексте с выбранным словом.

По сути, будем моделировать следующие распределения:



Такие действия проводят для каждого из слов, используя весь текст для порождения обучающей выборки. Обучающие выборки получаются огромных размеров. Пробежаться по ним один раз потребует огромного количества времени.

Постараемся задать данный алгоритм на уровне архитектуры.



На входе есть one-hot вектор, затем производится линейное преобразование в маломерное представление (скрытый линейный слой). На основании полученного векторного представления есть линейный классификатор, который предсказывает, какое слово оказалось в контексте с данными словами.

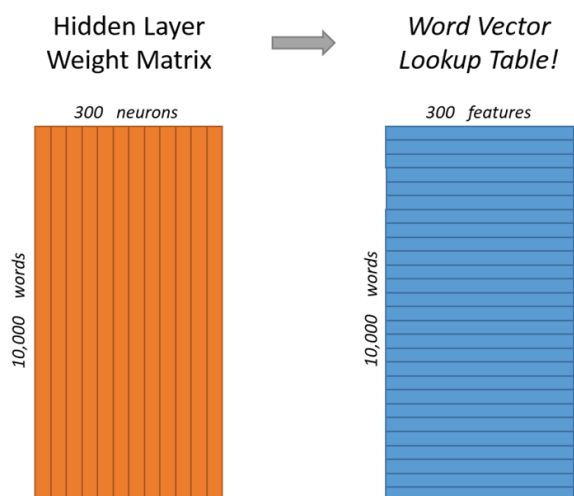
Теперь можно учить модель предсказывать, какие слова могут попасть в контекст к данным словам. И для каждого вектора мы предсказываем некоторое плотное множество представлений.

Верно ли, что для похожих слов контекст одинаковый? Да, на похожие слова будут производиться одинаковые предсказания, а значит, их векторные представления будут похожи друг на друга. То есть на уровне архитектуры мы заложили идею, что похожие по контексту слова должны получить похожее векторное представление. И нам даже не надо это как-то регулировать. Сам контекст сделает это за нас.

У слова в разных контекстах могут быть разные смыслы, поэтому модель не всегда может сойтись в своем предсказании. Но это и не важно. Нам достаточно, что модель может предсказать контекст лучше, чем случайным образом. У one-hot вектора вообще никакой информации не было.

Как работает метод Word2vec

У нас есть матрица весов первого линейного преобразования. При умножении one-hot вектора на нее мы получаем определенную строку. Поэтому получаем хэш-таблицу, которая для каждого элемента нашего словаря содержит векторное представление слова.

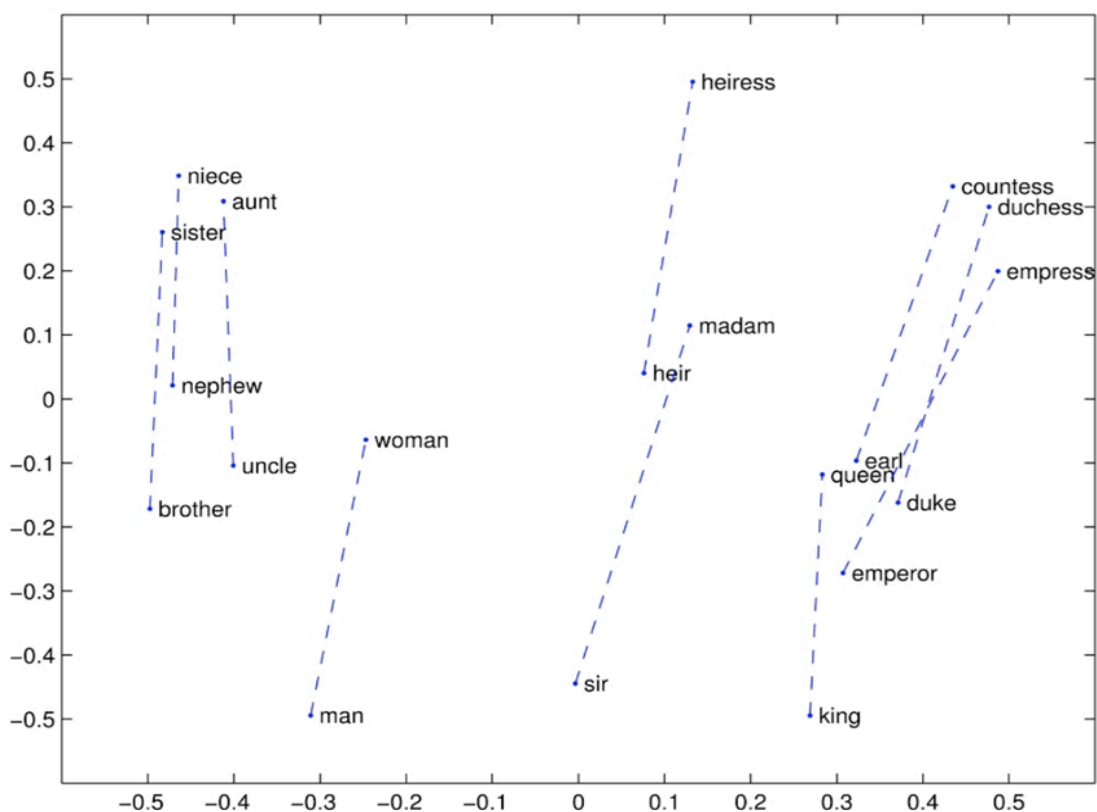


По сути, мы получаем словарь из человеческого в машинный язык. Его один раз обучили, и потом можно переиспользовать. Обычно так и делают, берут предобученные Word2vec откуда угодно.

У Word2vec есть проблема: придется выучить две матрицы по 3 млн весов каждая. Это дорого, и учить это проблематично.

5. Векторные представления слов. Визуализация. (Word embeddings visualization)

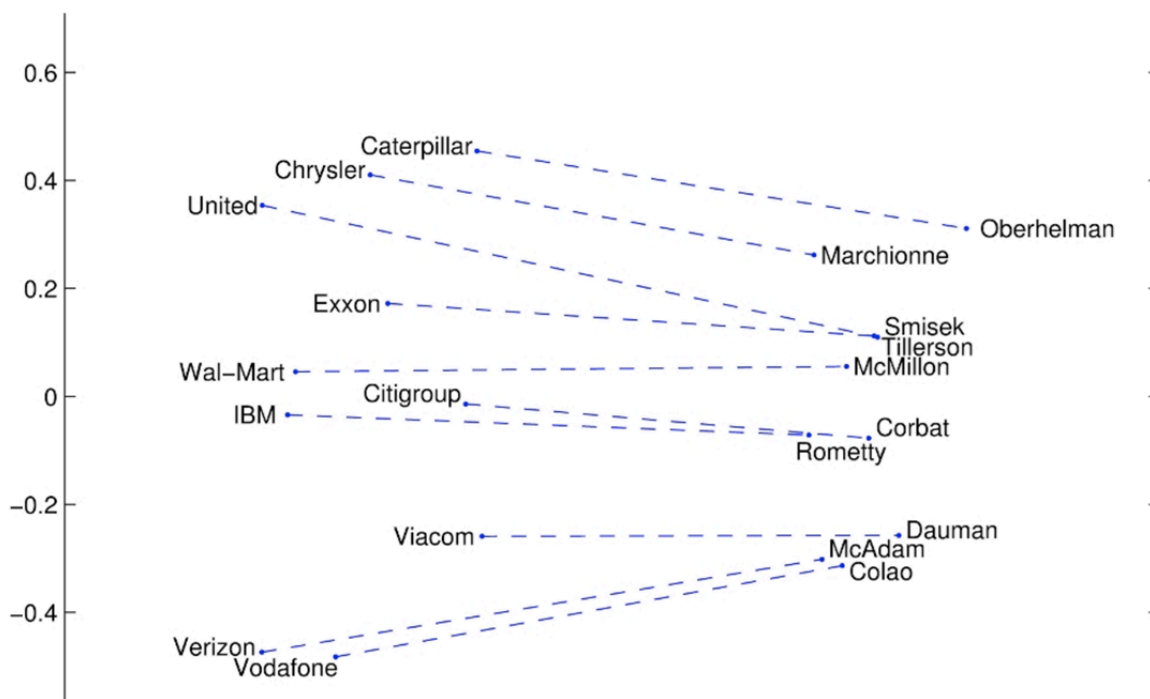
Пример 1. Рассмотрим пример визуализации векторных представлений.



С помощью PCA мы снизили размерность до двумерного случая. В данном визуальном представлении видим, что сверху находятся слова, соответствующие женскому роду, снизу — мужскому.

Получаем, что разница для одного и того же слова для мужского и женского рода практически константа.

Пример 2. Визуализация взаимосвязи между названием компании и именем его CEO.

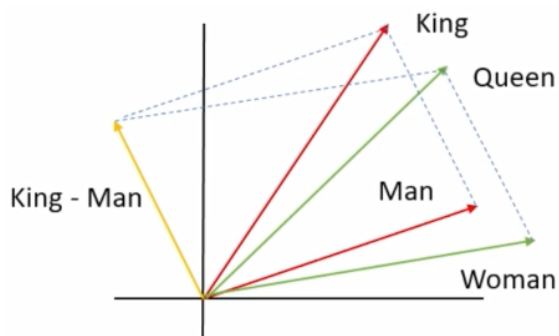


Визуализация получена на основе обучения из новостных заметок. Все компании находятся слева, CEO — справа. Расстояние между ними примерно одинаковое. На основании такой визуализации можно предсказать имя человека для определенной компании, если он появлялся где-то в новостях.

Метод подбора подходящего слова:

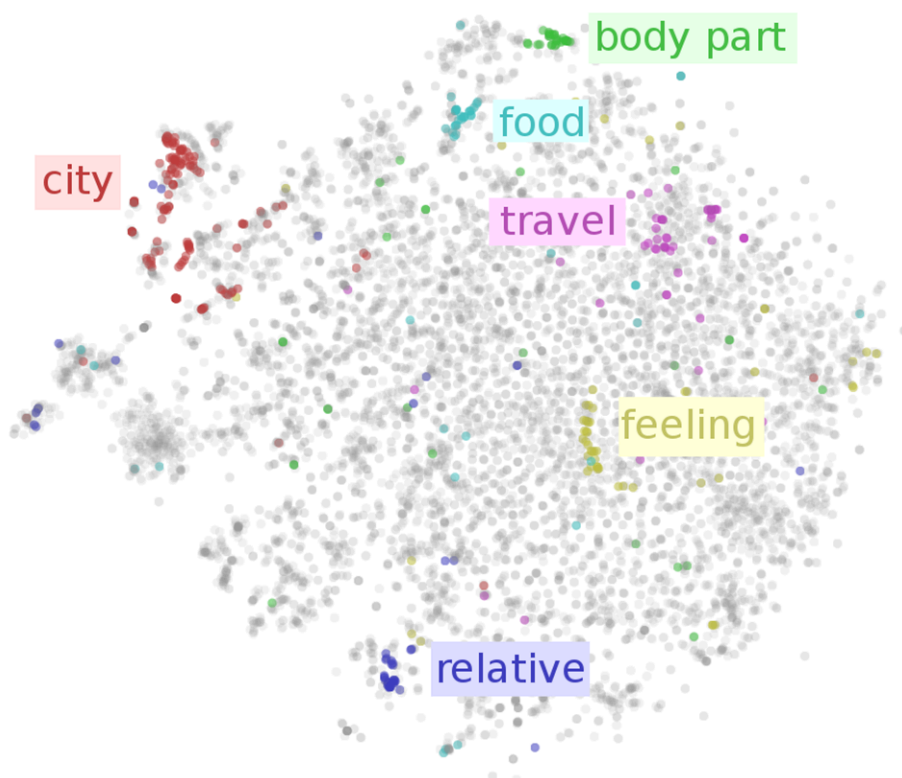
King - man + woman = queen
 $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$
 $x \quad y \quad y' \quad target$

$$\cos(x - y + y', target) \rightarrow \max_{target}$$



Мы находим, для какого слова из нашего словаря рассчитанный вектор ближе всего. Для определения близости используем не евклидово расстояние, а **косинусную меру близости**. Косинусная мера близости гораздо лучше подходит для нахождения разницы между словами, потому что она игнорирует нормы векторов. Чем чаще встречается слово, тем больше норма его вектора.

Пример 3. Слова были перенесены в векторное пространство, а затем снизили размерность до 2.



Слова, похожие по смыслу, находятся в одном облаке точек.

Таким образом, мы научились отображать слово в векторное пространство, при этом сохраняя его смысл и смысловую близость. Как только появляется некоторая мера близости между исходными объектами, мы можем использовать ее и построить векторное представление, которое в векторном поле будет показывать близкие объекты.

6. Визуализация в Python

Эмбединг — это векторное представление слов (токенов, фраз, морфем и т. д.).

Одно из векторных представлений слов — мешок слов (bag of words). Это простой и прямолинейный подход. Он делает подсчет, сколько раз какое слово встречалось в тексте.

У подхода много недостатков, так как он не учитывает:

- порядок слов;
- контекст слов;
- более дальние связи слов.

Улучшить векторное представление текстов можно с помощью подхода Word2vec, который «смотрит» на слово с точки зрения его контекста. Эмбединги слов получаются уже с точки зрения контекста.:

```
!pip install --upgrade nltk gensim boken umap-learn
```

nltk — библиотека для препроцессинга текста.

gensim — библиотека, в которой содержатся модели.

```
import gensim
```

```
import itertools
```

```
import string
```

```
import numpy as np
```

```
import umap
```

```
from nltk.tokenize import WordPunctTokenizer
```

```
from matplotlib import pyplot as plt
```

```
from IPython.display import clear_output
```

Возьмем данные — вопросы из quora, аналог вопросника mail.ru:

```
#download the data:

!wget https://www.dropbox.com/s/obaitrix9jyu84r/quora.txt?dl=1 -O
./quora.txt -nc

#alternative download link: https://yadi.sk/i/BPQrUu1NaTduEw
```

```
data = list(open("./quora.txt", encoding="utf-8"))
data[5]
```

```
>>> 'What can I do to improve my immune system?\n'
```

Мы будем «смотреть» на слово и его контекст.

Датасет довольно большой:

```
data
```

Чтобы правильно сделать векторное представление, нам нужно разбить слово и предложение на токены. Каждому токenu будет даваться какое-то векторное представление.

```
tokenizer = WordPunctTokenizer()

print(tokenizer.tokenize(data[50]))
```

```
>>> ['What', 'TV', 'shows', 'or', 'books', 'help', 'you', 'read',
'people', '"', 's', 'body', 'language', '?']
```

Токенизируем все наши данные:

```
# TASK: lowercase everything and extract tokens with tokenizer.

# data_tok should be a list of lists of tokens for each line in
data.
```

```
data_tok = [tokenizer.tokenize(ind.lower()) for ind in data]
```

```
data_tok
```

Теперь у нас образовался двойной массив. Каждый входящий массив — токенизированное предложение.

Из токенизированных массивов мы можем снова воссоздать предложение:

```
' '.join(data_tok[0])
```

```
>>> "can i get back with my ex even though she is pregnant with  
another guy ' s baby ?"
```

```
assert all(isinstance(row, (list, tuple)) for row in data_tok),  
"please convert each line into a list of tokens (strings)"  
  
assert all(all(isinstance(tok, str) for tok in row) for row in  
data_tok), "please convert each line into a list of tokens  
(strings)"  
  
is_latin = lambda tok: all('a' <= x.lower() <= 'z' for x in tok)  
  
assert all(map(lambda l: not is_latin(l) or l.islower(), map(''  
''.join, data_tok))), "please make sure to lowercase the data"
```

Теперь можно перейти к Word2vec. В Word2vec есть два основных подхода:

- мы «скользим» по предложению и записываем слово-таргет, которое по центру. Контексты справа и слева учитываем с какими-то весами — **CBow** (Continious Bag of Words). Затем по контексту предсказываем слово.
- **Skip-gram** — мы берем слово и предсказываем по нему контекст.

```
from gensim.models import Word2Vec

model = Word2Vec(data_tok,

                  vector_size=32, # embedding vector size

                  min_count=5, # consider words that occurred at
least 5 times

                  window=5).wv # define context as a 5-word
window around the target word
```

Как узнать, в каком режиме модель будет предсказывать? Для этого у модели есть атрибут **.sg**. У этого атрибута есть два варианта – 0 или 1:

- 0 – CBow;
- 1 – Skip-gram.

```
Word2Vec(data_tok,

          vector_size=32, # embedding vector size

          min_count=5, # consider words that occurred at
least 5 times

          window=5).sg
```

```
>>> 0
```

Посмотрим на эмбединги слов.

```
# now you can get word vectors !

model.get_vector('anything')
```

```
>>> array([-1.5752376 , -0.9731617 ,  1.8669678 ,  3.5249043 ,  2.3736055 ,
           0.9049323 , -0.9795948 , -5.6637373 ,  1.2144465 ,  2.1200514 ,
          -0.04951129,  2.6814644 ,  4.99559   ,  1.2394284 ,  2.6379015 ,
```

```
-0.54646885, -0.56735647, -1.0117594 , 0.26925406, -2.2996085 ,  
-2.1971612 , 0.0697102 , -2.2669172 , -0.88472766, -0.1711679 ,  
-1.0784123 , -0.33752438, 0.02864904, 0.11451391, -0.21015045,  
-2.2564306 , 1.0573425 ], dtype=float32)
```

Посмотрим, какие слова чаще всего встречаются с выбранным словом:

```
# or query similar words directly. Go play with it!  
model.most_similar('bread')
```

```
>>> [('meat', 0.9616428017616272),  
      ('corn', 0.9610626101493835),  
      ('cheese', 0.9532765746116638),  
      ('noodles', 0.9493104815483093),  
      ('soup', 0.9440537691116333),  
      ('egg', 0.9418217539787292),  
      ('milk', 0.941437304019928),  
      ('chicken', 0.9398934841156006),  
      ('beans', 0.9390753507614136),  
      ('toast', 0.936586856842041)]
```

Word2vec не единственный способ представления слов в векторном виде. Возьмем glove-twitter-25.

```
import gensim.downloader as api  
model = api.load('glove-twitter-25')
```

Посмотрим на слова, наиболее близко расположенные к словам `coder` и `money`, и наиболее далеко расположенные от слова `brain`:

```
model.most_similar(positive=["coder", "money"],  
negative=["brain"])
```

```
>>> [('realtor', 0.8265186548233032),  
      ('gfx', 0.8249695897102356),  
      ('caterers', 0.798202395439148),  
      ('beatmaker', 0.7936854362487793),  
      ('recruiter', 0.7892400026321411),  
      ('sfi', 0.784467339515686),  
      ('sosh', 0.7840632796287537),  
      ('promoter', 0.7838250994682312),  
      ('smallbusiness', 0.7786215543746948),  
      ('promoters', 0.77646803855896)]
```

Попробуем нарисовать наши данные.

Самые популярные слова, которые встречаются в нашем датасете:

```
model.key_to_index
```

```
words = list(model.key_to_index.keys())[:1000]  
# for each word, compute it's vector with model  
  
print(words[:101])  
  
word_vectors = np.asarray([model[x] for x in words])
```



```
>>> ['<user>', 'mi', 'much', '✓', 'kita', 'buat', 'looking', 'y',  
'kak', 'gusta']
```

Мы рассматриваем 1000 векторов и сопоставляем векторы длиной 25:

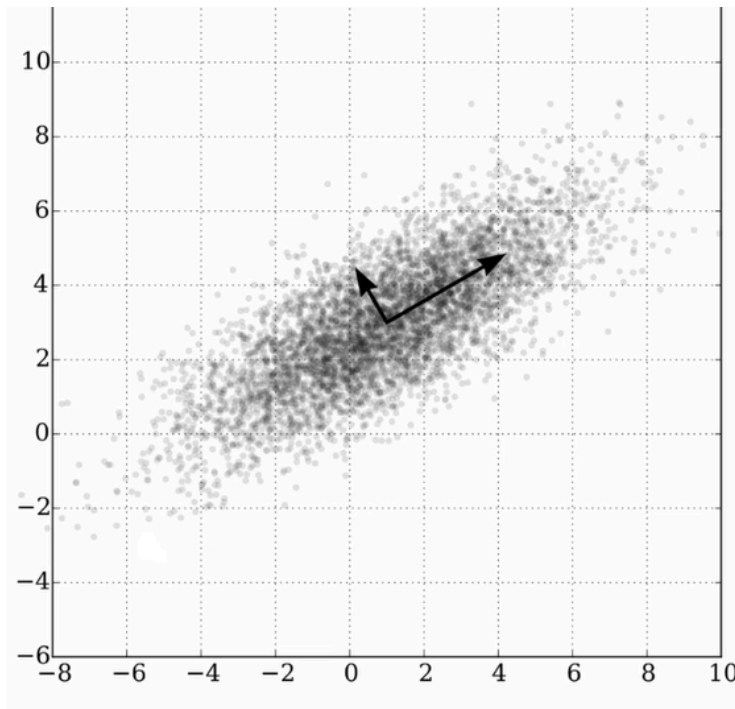
```
model[words].shape
```

```
>>> (1000, 25)
```

```
# for each word, compute it's vector with model  
word_vectors = model[words] # YOUR CODE
```

```
assert isinstance(word_vectors, np.ndarray)  
assert word_vectors.shape == (len(words), 25)  
assert np.isfinite(word_vectors).all()
```

25-мерное пространство представить очень сложно, но мы можем спроецировать это пространство на плоскость. Будем использовать метод PCA (Principal Component Analysis). Это статистический метод, который позволяет преобразовать систему координат так, чтобы каждая из главных осей описывала наибольший разброс данных.



По сути, метод PCA — поворот и сжатие. Мы отбрасываем те направления, вдоль которых расположено наименьшее количество информации. PCA помогает описать данные наиболее эффективно с наименьшими потерями.

С точки зрения оптимизации наша задача выглядит следующим образом:

$$\|(XW)\hat{W} - X\|_2^2 \rightarrow_{W, \hat{W}} \min$$

$$X \in R^{n \times m}$$

$$W \in R^{m \times d}$$

$$\hat{W} \in R^{d \times m}$$

Мы хотим найти такое низкоранговое приближение исходной матрицы X , чтобы минимизировать приведенный квадратичный функционал.

Напомним, что ранг матрицы — это количество линейно независимых строк или столбцов.

Мы сжимаем размерность 25 до 2.

```
len(word_vectors)
```

```
>>> 1000
```

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

pca = PCA(2)

scaler = StandardScaler()

# map word vectors onto 2d plane with PCA. Use good old sklearn
api (fit, transform)

# after that, normalize vectors to make sure they have zero mean
and unit variance

word_vectors_pca =
scaler.fit_transform(pca.fit_transform(word_vectors)) # YOUR CODE

# and maybe MORE OF YOUR CODE here :)
```

```
assert word_vectors_pca.shape == (len(word_vectors), 2), "there
must be a 2d vector for each word"

assert max(abs(word_vectors_pca.mean(0))) < 1e-5, "points must be
zero-centered"

assert max(abs(1.0 - word_vectors_pca.std(0))) < 1e-2, "points
must have unit variance"
```

```
import bokeh.models as bm, bokeh.plotting as pl
```

```
from bokeh.io import output_notebook

output_notebook()

def draw_vectors(x, y, radius=10, alpha=0.25, color='blue',
                 width=600, height=400, show=True, **kwargs):

    """ draws an interactive plot for data points with auxiliary
    info on hover """

    if isinstance(color, str): color = [color] * len(x)

    data_source = bm.ColumnDataSource({ 'x' : x, 'y' : y,
    'color': color, **kwargs })

    fig = pl.figure(active_scroll='wheel_zoom', width=width,
    height=height)

    fig.scatter('x', 'y', size=radius, color='color',
    alpha=alpha, source=data_source)

    fig.add_tools(bm.HoverTool(tooltips=[(key, "@" + key) for key
in kwargs.keys()])))

    if show: pl.show(fig)

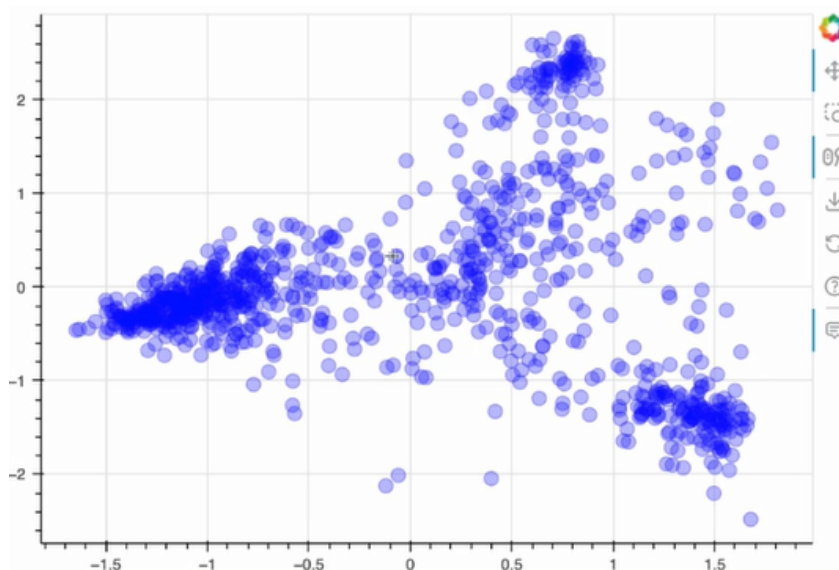
    return fig
```

```
len(words), len(word_vectors_pca)
```

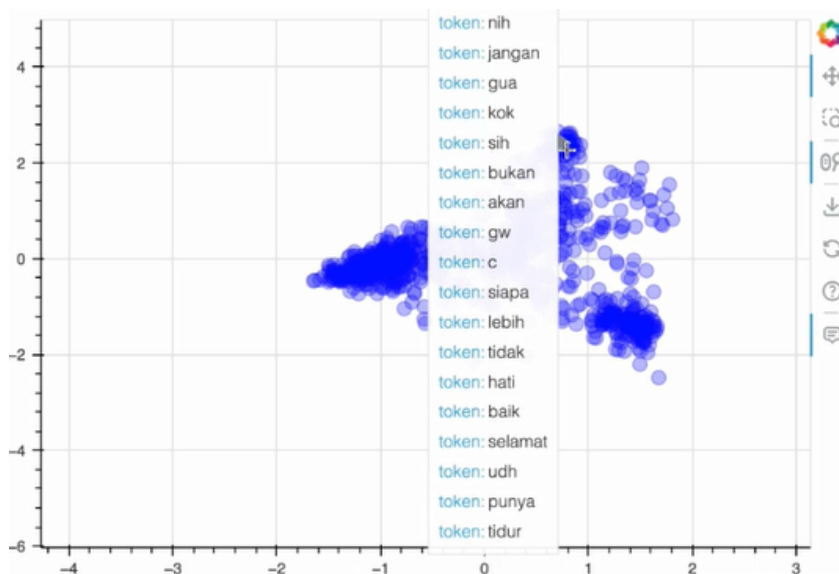
```
>>> (1000, 1000)
```

```
draw_vectors(word_vectors_pca[:1000, 0], word_vectors_pca[:1000, 1], token=words)
```

```
# hover a mouse over there and see if you can identify the clusters
```



Видим, что на рисунке есть некоторые кластеры — скопления объектов. Если навести на них стрелочку, можно увидеть, какие слова есть в этом кластере.



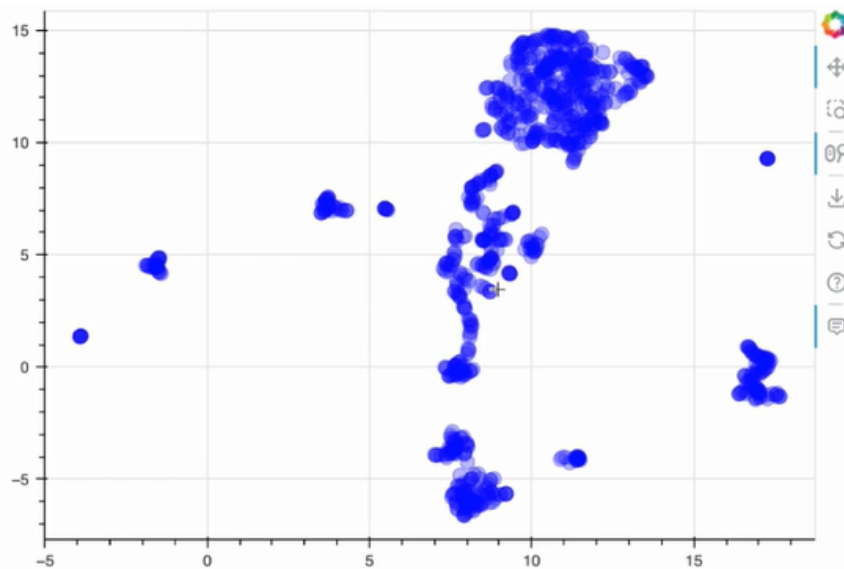
Кроме PCA есть и другие методы представления данных в двумерном пространстве.

Воспользуемся библиотекой UMAP:

```
embedding = umap.UMAP(n_neighbors=5).fit_transform(word_vectors)
```

```
draw_vectors(embedding[:, 0], embedding[:, 1], token=words)
```

```
# hover a mouse over there and see if you can identify the  
clusters
```



Здесь снова слова как-то разбросаны на кластеры.

Мы можем группировать на плоскости по кластерам не только слова, но и целые фразы. То есть можно строить эмбединги для целых фраз:

```
def get_phrase_embedding(phrase):  
    """  
    Convert phrase to a vector by aggregating it's word
```

```
embeddings. See description above.

"""

# 1. lowercase phrase

# 2. tokenize phrase

# 3. average word vectors for all words in tokenized phrase

# skip words that are not in model's vocabulary

# if all words are missing from vocabulary, return zeros

vector = np.zeros([model.vector_size], dtype='float32')

phrase_tokenized = tokenizer.tokenize(phrase.lower())

phrase_vectors = [model[x] for x in phrase_tokenized if
model.has_index_for(x)]

if len(phrase_vectors) != 0:

    vector = np.mean(phrase_vectors, axis=0)

# YOUR CODE

return vector
```

Посмотрим на полученные данные:

```
data[402687]
```

```
>>> 'What gift should I give to my girlfriend on her birthday?\n'
```

Вектор по-прежнему размера 25:

```
get_phrase_embedding(data[402687]).shape
```

```
>>> (25,)
```

Затокенизируем часть наших слов:

```
vector = get_phrase_embedding("I'm very sure. This never happened  
to me before...")
```

Составляем фразовые векторы:

```
# let's only consider ~5k phrases for a first run.  
chosen_phrases = data[:,len(data) // 1000]  
  
# compute vectors for chosen phrases and turn them to numpy array  
phrase_vectors = np.asarray([get_phrase_embedding(x) for x in  
chosen_phrases]) # YOUR CODE
```

```
assert isinstance(phrase_vectors, np.ndarray) and  
np.isfinite(phrase_vectors).all()  
  
assert phrase_vectors.shape == (len(chosen_phrases),  
model.vector_size)
```

Посмотрим на размерность:


```
phrase_vectors.shape
```

```
>>> (1001, 25)
```

Нарисуем наши фразы:

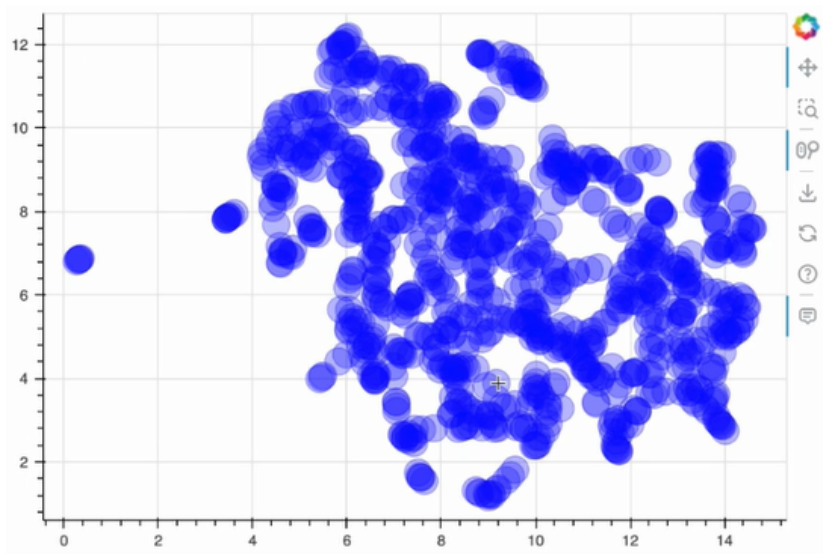
```
# map vectors into 2d space with pca, tsne or your other method  
of choice
```

```
# don't forget to normalize
```

```
phrase_vectors_2d =  
umap.UMAP(n_neighbors=3).fit_transform(phrase_vectors) #  
преобразовываем
```

```
# phrase_vectors_2d = (phrase_vectors_2d -  
phrase_vectors_2d.mean(axis=0)) / phrase_vectors_2d.std(axis=0)
```

```
draw_vectors(phrase_vectors_2d[:, 0], phrase_vectors_2d[:, 1],  
              phrase=[phrase[:50] for phrase in chosen_phrases],  
              radius=20,)
```



Проверим, насколько наше представление верное. Возьмем какую-то фразу и найдем 10 ближайших к ней фраз:

```
# compute vector embedding for all lines in data
data_vectors = np.vstack([get_phrase_embedding(l) for l in data])
```

```
norms = np.linalg.norm(data_vectors, axis=1)
```

```
printable_set = set(string.printable)
```

```
data_subset = [x for x in data if set(x).issubset(printable_set)]
```

```
def find_nearest(query, k=10):
```

```
    """
```

```
given text line (query), return k most similar lines from
data, sorted from most to least similar
```

```
similarity should be measured as cosine between query and
line embedding vectors
```

```
hint: it's okay to use global variables: data and
data_vectors. see also: np.argpartition, np.argsort
```

```
"""
# YOUR CODE

query_vector = get_phrase_embedding(query)

dists = data_vectors.dot(query_vector[:, None])[:, 0] /
((norms+1e-16)*np.linalg.norm(query_vector))

nearest_elements = dists.argsort(axis=0)[-k:][:-1]

out = [data[i] for i in nearest_elements]

return out# <YOUR CODE: top-k lines starting from most
similar>
```

```
results = find_nearest(query="How do i enter the matrix?", k=10)
results
```

```
>>> ['How do I get to the dark web?\n',
      'What universal remote do I need and how do I set it up to a
      Blaupunkt TV?\n',
      'How do I connect the ASUS_T00Q to my PC?\n',
      'How do you print the gridlines in Excel 2010?\n',
      'How do you print the gridlines in Excel 2007?\n',
      'How do you print the gridlines in Excel 2003?\n',
```

```
'I would like to create a new website. What do I have to do?\n',  
  
'How do I get the new Neko Atsume wallpapers? How do they  
work?\n',  
  
'I want to experience the 4G network. Do I need to change my SIM  
card from 3G to 4G?\n',  
  
'What do I have to do to sell my photography?\n']
```

```
print(''.join(results))  
  
assert len(results) == 10 and isinstance(results[0], str)  
assert results[0] == 'How do I get to the dark web?\n'  
# assert results[3] == 'What can I do to save the world?\n'
```

```
>>> How do I get to the dark web?
```

```
What universal remote do I need and how do I set it up to a  
Blaupunkt TV?
```

```
How do I connect the ASUS_T00Q to my PC?
```

```
How do you print the gridlines in Excel 2010?
```

```
How do you print the gridlines in Excel 2007?
```

```
How do you print the gridlines in Excel 2003?
```

```
I would like to create a new website. What do I have to do?
```

```
How do I get the new Neko Atsume wallpapers? How do they work?
```

```
I want to experience the 4G network. Do I need to change my SIM  
card from 3G to 4G?
```

```
What do I have to do to sell my photography?
```

```
find_nearest(query="How does Trump?", k=10)
```

```
>>> ['What does Donald Trump think about Israel?\n',  
     'Who or what is Donald Trump, really?\n',  
     'Donald Trump: Why are there are so many questions about Donald  
Trump on Quora?\n',  
     'Does anyone like Trump and Clinton?\n',  
     'What does Cortana mean?\n',  
     'Did Bill Gates outcompete and outsmart IBM? Why? How?\n',  
     'Why and how is Bill Gates so rich?\n',  
     'What does Donald Trump think about Pakistan?\n',  
     'What do you think about Trump and Obama?\n',  
     'Who and what is Quora?\n']
```

```
find_nearest(query="Why don't i ask a question myself?", k=10)
```

```
>>> ["Why don't my parents listen to me?\n",  
     "Why don't people appreciate me?\n",  
     "Why she don't interact with me?\n",  
     "Why don't I get a date?\n",  
     "Why don't I get a girlfriend?\n",  
     "Why don't I have a girlfriend?\n",  
     "Why don't I have a boyfriend?\n",  
     "Why don't I like people touching me?\n",  
     "Why can't I ask a question anonymously?\n",  
     "Why don't you use Facebook much?\n"]
```

Задание для самостоятельной работы. Решите, релевантные рекомендации в приведенных примерах или нет?

Поместим все наши данные в некоторый алгоритм кластеризации. Возьмем алгоритм KMeans:

```
from sklearn.cluster import DBSCAN, KMeans
```

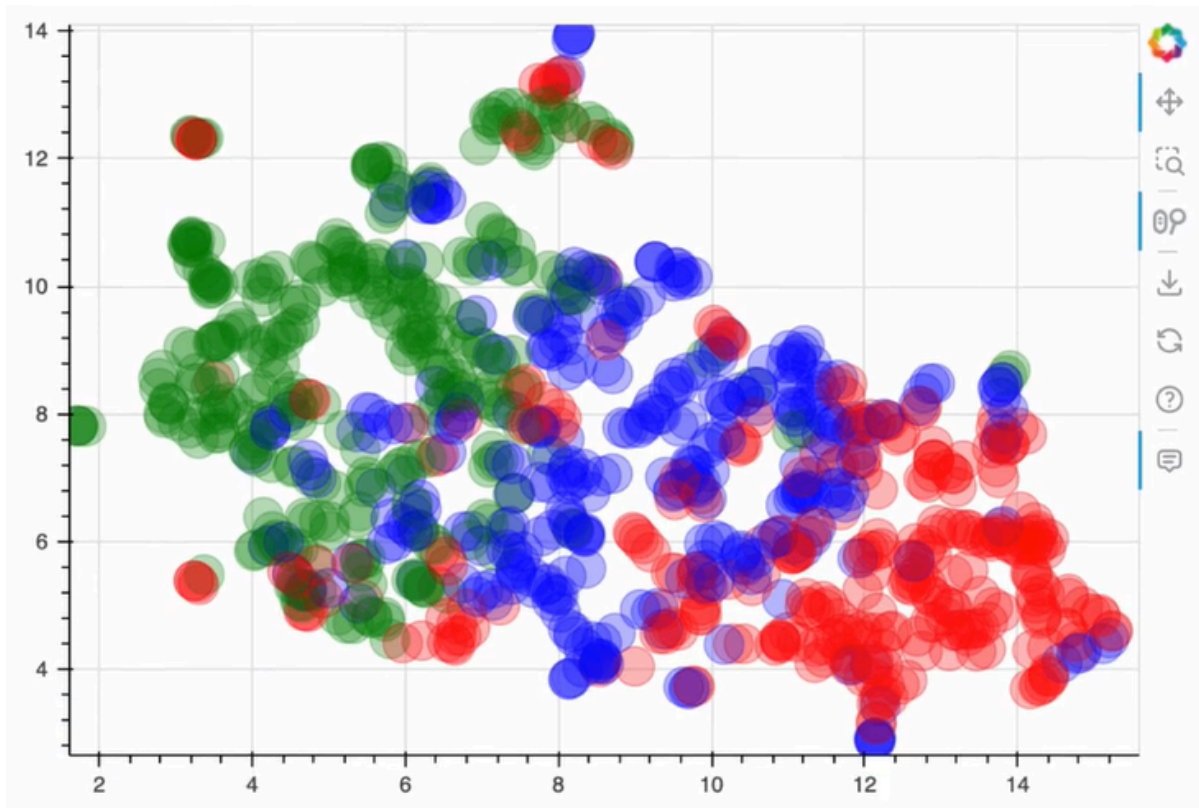
На вход KMeans будет подаваться количество кластеров. KMeans на каждом шаге будет считать центр кластера, расстояние между векторами до этого центра, смотреть, какой ближайший. И менять центр кластера, основываясь на информации предыдущего шага.

```
kmeans = KMeans(3) #3 кластера
```

```
labels = kmeans.fit_predict(np.asarray(phrase_vectors))
```

```
_colors = ['red', 'green', 'blue']
```

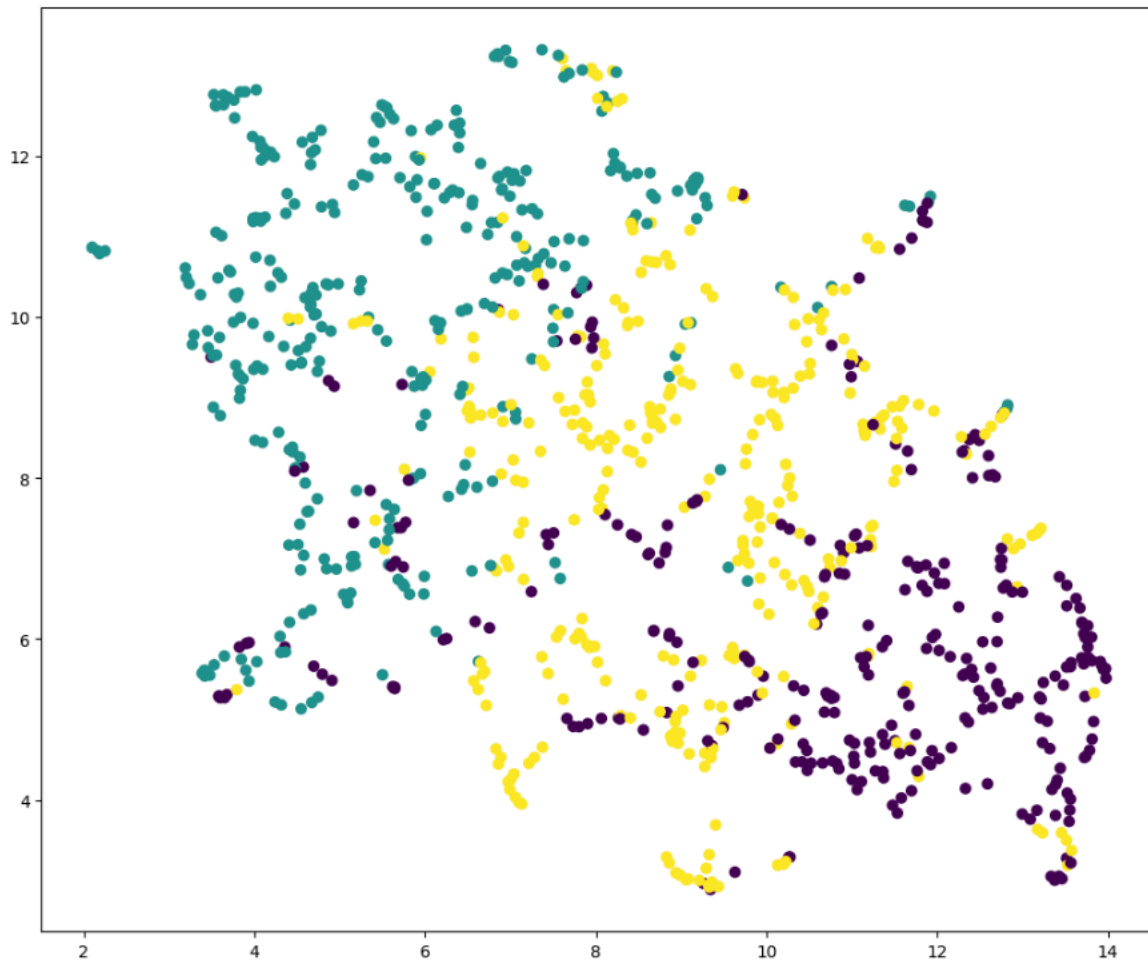
```
draw_vectors(phrase_vectors_2d[:, 0], phrase_vectors_2d[:, 1],
              color=[_colors[l] for l in labels],
              phrase=[phrase[:50] for phrase in chosen_phrases],
              radius=20,)
```



Нарисуем немного по-другому:

```
plt.figure(figsize=(12, 10))  
  
plt.scatter(phrase_vectors_2d[:,0], phrase_vectors_2d[:, 1],  
c=labels.astype(float))
```

```
<matplotlib.collections.PathCollection at 0x7f7bdf2090d0>
```



Кластеры разделяются не очень хорошо, очень много выбросов.

Теперь попробуем написать наш собственный Word2vec в виде нейронной сети, не используя трюки «из-под капота».

Проверим вместимость видеокарты:

```
!nvidia-smi
```

Wed May 17 14:17:19 2023

NVIDIA-SMI 450.102.04 Driver Version: 450.102.04 CUDA Version: 12.1									
GPU	Name	Persistence-M		Bus-Id	Disp.A	Volatile	Uncorr.	ECC	
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage		GPU-Util	Compute	M.	
0	Tesla	V100-SXM2...	On	00000000:06:00.0	Off			0	
N/A	37C	P0	59W / 300W	13159MiB / 16160MiB		0%		Default	N/A
1	Tesla	V100-SXM2...	On	00000000:07:00.0	Off			0	
N/A	38C	P0	61W / 300W	3138MiB / 16160MiB		0%		Default	N/A
2	Tesla	V100-SXM2...	On	00000000:0A:00.0	Off			0	
N/A	42C	P0	69W / 300W	5MiB / 16160MiB		0%		Default	N/A
3	Tesla	V100-SXM2...	On	00000000:0B:00.0	Off			0	
N/A	35C	P0	59W / 300W	423MiB / 16160MiB		0%		Default	N/A
4	Tesla	V100-SXM2...	On	00000000:85:00.0	Off			0	
N/A	34C	P0	43W / 300W	9MiB / 16160MiB		0%		Default	N/A
5	Tesla	V100-SXM2...	On	00000000:86:00.0	Off			0	
N/A	38C	P0	59W / 300W	12457MiB / 16160MiB		0%		Default	N/A
6	Tesla	V100-SXM2...	On	00000000:89:00.0	Off			0	
N/A	59C	P0	236W / 300W	16022MiB / 16160MiB		72%		Default	N/A
7	Tesla	V100-SXM2...	On	00000000:8A:00.0	Off			0	
N/A	36C	P0	43W / 300W	3MiB / 16160MiB		0%		Default	N/A
Processes:									
GPU	GI	CI	PID	Type	Process name	GPU Memory			
	ID	ID				Usage			

>>>

Будем использовать метод Skip-gram, который по слову передает его контекст.
Построим нейронную сеть:

```
import os

os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"

os.environ["CUDA_VISIBLE_DEVICES"]="4"
```

```
import torch

import torch.nn as nn

import torch.autograd as autograd

import torch.optim as optim

import torch.nn.functional as F

from torch.optim.lr_scheduler import StepLR, ReduceLROnPlateau
```

Построим два словаря. Один по слову выдает уникальный индекс, второй, наоборот, по индексу выдает уникальное слово:

```
vocabulary = set(itertools.chain.from_iterable(data_tok))

word_to_index = {word: index for index, word in
enumerate(vocabulary)}

index_to_word = {index: word for word, index in
word_to_index.items()}

word_counter = {word: 0 for word in word_to_index.keys() }
```

Создаем пары контекста:

```
context_tuple_list = []

w = 4

for text in data_tok:
    for i, word in enumerate(text):
        first_context_word_index = max(0, i-w)
        last_context_word_index = min(i+w, len(text))

        for j in range(first_context_word_index,
last_context_word_index):
            if i!=j:
                context_tuple_list.append((word_to_index[word],
word_to_index[text[j]]))

                word_counter[word] += 1.

print("There are {} pairs of target and context
words".format(len(context_tuple_list)))
```

```
context_tuple_list
```

```
(60577, 30463),  
(60577, 52925),  
(60577, 81630),  
(60577, 27168),  
(60577, 17094),  
(81630, 69151),  
(81630, 30463),  
(81630, 52925),  
(81630, 60577),  
(81630, 27168),  
(81630, 17094),  
(81630, 35710),  
...]  
>>>
```

В данном листе у нас слева располагается индекс таргетного слова, справа — индекс контекста. Такие пары будем подавать на вход нашей модели. Всего таких пар у нас 41323344:

```
data_torch =  
torch.tensor(context_tuple_list).type(torch.LongTensor)  
  
X_torch = data_torch[:, 0]  
  
y_torch = data_torch[:, 1]  
  
del data_torch
```

```
len(X_torch)
```

```
>>> 41323344
```

```
X_torch
```

```
>>> tensor([75321, 75321, 75321, ..., 63595, 63595, 63595])
```

```
y_torch
```

```
>>> tensor([ 6564,  8290, 66801, ..., 63314, 21277,  3592])
```

Теперь приступим к самой модели Word2vec.

```
class Word2VecModel(nn.Module):

    def __init__(self, embedding_size, vocab_size):

        super().__init__()

        # YOUR CODE HERE

        self.word2emb = nn.Embedding(vocab_size, embedding_size) #
слои в Torch. Принимает на вход 2 параметра: количество предложений и
embedding_size — варьируемый параметр.

        self.emb2ctxt = nn.Linear(embedding_size,
vocab_size) #передаем вектор на линейный слой и предсказываем наш контекст

    def forward(self, word):

        # YOUR CODE HERE

        emb = self.word2emb(word)

        return self.emb2ctxt(emb)
```

Слой `self.word2emb` переводит наши слова в эмбединги.

Word2vec — это что-то типа автоэнкодера.

Переносим все наши вычисления на GPU:

```
device = torch.device('cuda:0') if torch.cuda.is_available() else
torch.device('cpu')
```

```
len(word_to_index)
```

```
>>> 87819
```

Создадим модель:

```
model = Word2VecModel(15, len(word_to_index)).to(device)
```

15 — размер эмбединга.

Объявляем модель, оптимизатор, функцию потерь и прочее:

```
loss_func = nn.CrossEntropyLoss()
opt = torch.optim.Adam(model.parameters(), lr=0.01)
# To reduce learning rate on plateau of the loss functions
lr_scheduler = ReduceLROnPlateau(opt, patience=35)
```

```
loss_func(model(X_torch[:5].to(device)), y_torch[:5].to(device))
```

```
>>> tensor(11.4281, device='cuda:0', grad_fn=<NllLossBackward0>)
```

Обучим модель:

```
batch_size = 1024
n_iterations = 1000
local_train_loss_history = []
```

Функция для отрисовки функции потерь:

```
def plot_train_process(train_loss):  
    fig, axes = plt.subplots(1, 1, figsize=(15, 5))  
  
    axes.set_title('Loss')  
    axes.plot(train_loss, label='train')  
    axes.legend()  
    plt.show()
```

```
import matplotlib.pyplot as plt  
from IPython.display import clear_output  
  
batch_size = 1024  
n_steps = 1000  
loss_history = []  
for i in range(n_steps):  
    ix = np.random.randint(0, len(context_tuple_list),  
batch_size)  
  
    x_batch = X_torch[ix].to(device)  
    y_batch = y_torch[ix].to(device)  
  
    # YOUR CODE HERE  
  
    # predict logits
```

```
y_pred = model(x_batch)

# YOUR CODE HERE

# compute loss

loss = loss_func(y_pred, y_batch)

# YOUR CODE HERE

# clear gradients

opt.zero_grad()

# YOUR CODE HERE

# compute gradients

loss.backward()

# YOUR CODE HERE

# optimizer step

opt.step()

loss_history.append(loss.item())

lr_scheduler.step(loss_history[-1])

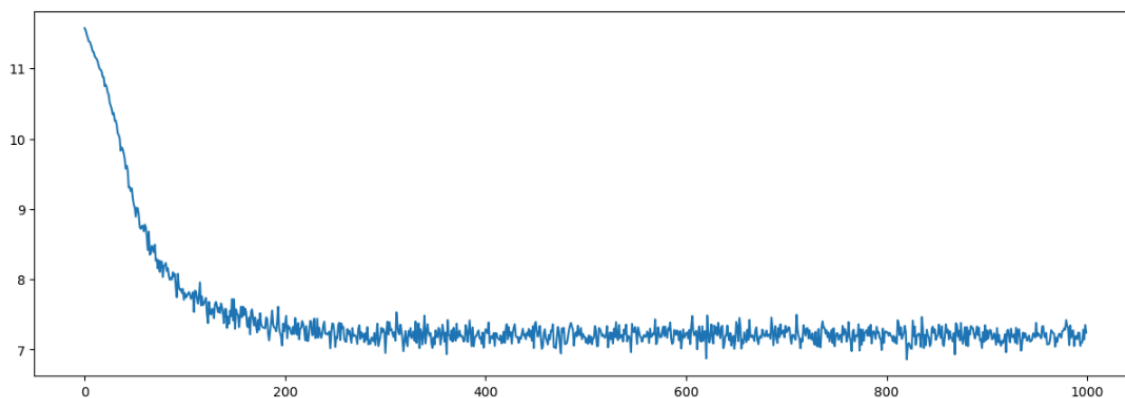
if (i + 1) % 100 == 0:

    clear_output(wait=True)
```



```
plt.figure(figsize=(15, 5))  
  
plt.plot(loss_history)  
  
plt.show()
```

>>>



Видим, что функция потерь падает, наша модель обучается.

Посмотрим на результат. Матрица эмбедингов задается следующим образом:

```
matrix = next(model.word2emb.parameters()).detach().cpu()
```

```
def get_closest(word, top_n):  
  
    global matrix, word_to_index, index_to_word  
  
    y = matrix[word_to_index[word]][None, :]  
  
    dist = F.cosine_similarity(matrix, y)  
  
    index_sorted = torch.argsort(dist)
```

```
top_n = index_sorted[-top_n:]  
  
return [index_to_word[x] for x in top_n.numpy()]
```

```
get_closest('cat', 8)
```

```
>>>['relocation',  
    'fisherface',  
    'prettier',  
    'ecom',  
    'josephine',  
    'allbestlist',  
    'sooryavansham',  
    'cat']
```

Дополнительные материалы для самостоятельного изучения

1. [WordNet](#)
2. [Natural Language Toolkit](#)
3. [Beautiful Soup](#)
4. [Морфологический анализатор pymorphy2](#)
5. [Word2Vec Tutorial – The Skip-Gram Model](#)