[Машинное обучение для текстов 2](#_Toc133324169)

[Структура курса 2](#_Toc133324170)

[Ваши цели 2](#_Toc133324171)

[Векторизация текстов 3](#_Toc133324172)

[Введение 3](#_Toc133324173)

[Чему вы научитесь 3](#_Toc133324174)

[Постановка задачи 3](#_Toc133324175)

[Лемматизация 3](#_Toc133324176)

[Задача 4](#_Toc133324177)

[Регулярные выражения 5](#_Toc133324178)

[Задача 6](#_Toc133324179)

[Мешок слов и N-граммы 7](#_Toc133324180)

[Создание мешка слов 10](#_Toc133324181)

[Задача 11](#_Toc133324182)

[TF-IDF 12](#_Toc133324183)

[TF-IDF в sklearn 14](#_Toc133324184)

[Задача 14](#_Toc133324185)

[Классификация тональности текста 15](#_Toc133324186)

[Задача 15](#_Toc133324187)

[Embeddings 16](#_Toc133324188)

[Word2vec 18](#_Toc133324189)

[Embeddings для классификации 21](#_Toc133324190)

[BERT 22](#_Toc133324191)

[RuBERT и предобработка 23](#_Toc133324192)

# Машинное обучение для текстов

Вы научитесь решать задачи классификации и регрессии для текстов.

Узнаете, как делать из текстов табличные данные.

#### Структура курса

Анализ текстов помогает компаниям оценить отношение к бренду и узнать, какой контент может понравиться потребителям. Быстрее человека в разы машина определит, какой тематики и тональности текст, есть ли в нём токсичные слова.

Чтобы научить машину работать с текстами, их нужно преобразовать в понятный для неё формат. Поэтому изучение курса вы начнёте с алгоритмов извлечения признаков из текстов. Узнаете, что такое мешок слов и N-граммы. Обучите логистическую регрессию определять тональность текста.

Затем познакомитесь с новым способом создания признаков из текстов — языковыми представлениями. Освоите word2vec и BERT. Узнаете, как работать с уже обученными языковыми моделями BERT и применять их в задаче классификации.

В конце курса вас ждёт проект по классификации текстов.

💡 Английский язык пригодится не только для того, чтобы разбираться в документации. Общение на английском с коллегами и работодателями — неотъемлемая часть работы в международных компаниях. Если хотите подтянуть знание языка, обратите внимание на курс «Английский для аналитиков» от Яндекс Практикума.

Обучение на курсе построено вокруг типичных ситуаций в работе аналитика: обсуждения данных и их визуализации, стендапов, планирования спринтов, сбора требований и защиты проекта перед заказчиком, а материал закрепляется на встречах с иностранцами из IT. В результате после курса вам будет легче общаться и работать в англоязычной среде. Для студентов Практикума действует скидка 10% по промокоду STUDENTSPRACTICUM.

[**Узнать больше ➞**](https://practicum.yandex.ru/english/english_for_career/#!/tab/491633859-2?utm_source=practicum&utm_medium=students_course&utm_campaign=prct_st_crs_data_scientist&utm_content=it_english)

## Ваши цели

* Научиться вычислять величины TF-IDF для текстов.
* Понять, как находить языковые представления моделью BERT.
* Построить модель классификации для текстов.

Это сложный курс: нужно освоить много инструментов работы с текстами.

Разобраться, что такое модель BERT, вам помогут наставники на вебинаре.

## Векторизация текстов

### Введение

Научимся извлекать из текстов признаки.

#### Чему вы научитесь

* Узнаете, что такое мешок слов и N-граммы.
* Освоите метод вычисления признаков TF-IDF.
* Построите модель для классификации текстов.

### Постановка задачи

Построим модель, которая классифицирует твиты положительной и отрицательной тональности.

### Лемматизация

Прежде чем извлечь признаки из текста, упростим его.

Рассмотрим этапы предобработки текста:

1. **Токенизация** (англ. *tokenization*) — разбиение текста на токены: отдельные фразы, слова, символы.
2. **Лемматизация** (англ. *lemmatization*) — приведение слова к начальной форме — лемме.

Функция лемматизации русского текста есть в библиотеках:

* pymorphy2 (англ. *python morphology,* «морфология для Python»), \*\*
* UDPipe (англ. *universal dependencies pipeline*, «конвейер для построения общих зависимостей»*),*
* pymystem3.

Рассмотрим, как работать с библиотекой pymystem3.

Создадим класс для лемматизации:

Скопировать кодPYTHON

from pymystem3 import Mystem

m = Mystem()

Функции lemmatize() передадим «лемматизируй это»:

Скопировать кодPYTHON

m.lemmatize("лемматизируй это")

Функция вернула лемму каждого слова. Для функции конец строки — это тоже знак, поэтому находится и его лемма (\n) (от англ. *new line*, «новая строка»):

Скопировать код

['лемматизировать', ' ', 'это', '\n']

Функцией join() объединим элементы списка в строку, разделив их пробелом (но можно и без него):

Скопировать кодPYTHON

" ".join(['лемматизировать', ' ', 'это', '\n'])

Получаем:

Скопировать код

'лемматизировать это \n'

В этой теме мы будем работать с русскоязычным корпусом коротких текстов RuTweetCorp.

Перед вами уменьшенный датасет — 5000 записей. Каждая запись содержит текст поста и оценку его тональности. Если пост позитивный, то метка «1», если негативный — «0». Осторожно: твиты могут содержать обсценную (от лат. *obscenus*, «непристойный») лексику.



Импортируем данные:

Скопировать кодPYTHON

import pandas as pd

data = pd.read\_csv('tweets.csv')

Чтобы алгоритмы умели определять тематику и тональность текста, их нужно обучить на корпусе (англ. *corpus*). Это набор текстов, в котором эмоции и ключевые слова уже размечены.

Создадим корпус постов. Преобразуем столбец 'text' в список текстов. Переведём тексты в стандартный для Python формат: кодировку Unicode U.

Изменим кодировку методом astype():

Скопировать кодPYTHON

corpus = data['text'].values.astype('U')

#### Задача

Напишите функцию lemmatize(text). На вход она принимает текст из корпуса и возвращает лемматизированную строку. Возьмите первый текст из датасета tweets.csv. Код напечатает на экране исходный и лематизированный тексты.

import pandas as pd

from pymystem3 import Mystem

data = pd.read\_csv('/datasets/tweets.csv')

corpus = data['text'].values.astype('U')

def lemmatize(text):

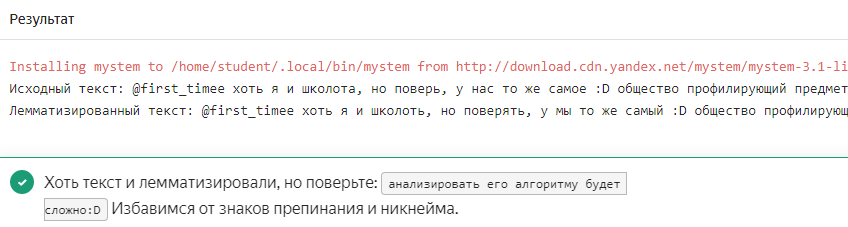
m = Mystem()

m.lemmatize(text)

return "".join(m.lemmatize(text))

print("Исходный текст:", corpus[0])

print("Лемматизированный текст:", lemmatize(corpus[0]))



### Регулярные выражения

От лишних символов текст очистят **регулярные выражения** (англ. regular expressions).

Это инструмент для поиска слова или числа по **шаблону** (англ. pattern). Он определяет, из каких частей состоит строка и какие в них символы. Например, нужно найти все даты в таком формате записи: 02.02.2020. Их шаблон — это два числа, точка, два числа, точка, четыре числа.

Шаблон для email-адреса такой: строка на латинице, символ @, строка на латинице, точка, строка на латинице.

Для работы с регулярными выражениями в Python есть встроенный модуль **re** (сокр. от regular expressions):

Скопировать кодPYTHON

import re

Познакомимся с функцией **re.sub()** (от англ. substitute, «заменять»). Она находит в тексте все совпадения по шаблону и заменяет их заданной строкой.

Скопировать кодPYTHON

*# pattern — шаблон*

*# replacement — на что заменять*

*# text — текст, в котором искать совпадения*

re.sub(pattern, replacement, text)

Вернёмся к нашей задаче. В лемматизированном тексте твита нужно оставить только кириллические символы и пробелы. Чтобы их найти, напишем регулярное выражение.

Оно начинается с символа r, заключается в кавычки и квадратные скобки:

Скопировать кодPYTHON

r'[]'

В квадратных скобках перечисляют все символы, подходящие под шаблон (в любом порядке, без пробелов). Запишем, что ищем буквы от «а» до «я». Они могут быть как в нижнем, так и верхнем регистрах, поэтому получаем:

Скопировать кодPYTHON

*# диапазон букв обозначается дефисом:*

*# а-я — это то же самое, что абвгдежзийклмнопрстуфхцчшщъыьэюя*

r'[а-яА-Я]'

Поскольку Python не знает, что буква Ё должна входить в диапазон, добавим её в нижнем и верхнем регистрах:

Скопировать кодPYTHON

r'[а-яА-ЯёЁ]'

Возьмём исходный текст твита. Под наш шаблон подходят кириллические символы и пробелы, которые как раз нужно оставить. Но если мы вызовем функцию re.sub(), их заменят пробелы. Чтобы указать, что символы под шаблон не подходят, перед набором символов поставим знак «домика» (^):

Скопировать кодPYTHON

*# уже лемматизированный текст*

text = "@first\_timee хоть я и школота, но поверь, у мы то же самый :d общество профилировать предмет типа)"

re.sub(r'[^а-яА-ЯёЁ ]', ' ', text)

Скопировать код

' хоть я и школота но поверь у мы то же самый общество профилировать предмет типа '

Так в тексте остались только кириллические символы и пробелы. После этой операции в тексте можно обнаружить лишние пробелы, для анализа они — помеха. Пробелы устраняются комбинацией функций join() и split().

Возьмём пример текста с лишними пробелами: в середине, начале и конце строки. Методом split() преобразуем его в список. Если не указывать аргументы у split(), он делает разбиение по пробелам или группам пробелов:

Скопировать кодPYTHON

text = " хоть я и школота но поверь "

text.split()

Получили список без пробелов:

Скопировать код

['хоть', 'я', 'и', 'школота', 'но', 'поверь']

Методом join() объединим элементы в строку через пробел:

Скопировать кодPYTHON

" ".join(['хоть', 'я', 'и', 'школота', 'но', 'поверь'])

Пробелов ровно столько, сколько нужно:

Скопировать кодPYTHON

'хоть я и школота но поверь'

#### Задача

Напишите функцию clear\_text(text), которая оставит в тексте только кириллические символы и пробелы. На вход она принимает текст, а возвращает очищенный текст. Дополнительно уберите лишние пробелы.

Напечатайте на экране исходный текст, а затем очищенный и лемматизированный тексты (уже в прекоде).

import pandas as pd

from pymystem3 import Mystem

import re

data = pd.read\_csv('/datasets/tweets.csv')

corpus = list(data['text'])

def lemmatize(text):

m = Mystem()

lemm\_list = m.lemmatize(text)

lemm\_text = "".join(lemm\_list)

return lemm\_text

def clear\_text(text):

template = r'[^а-яА-ЯёЁ ]'

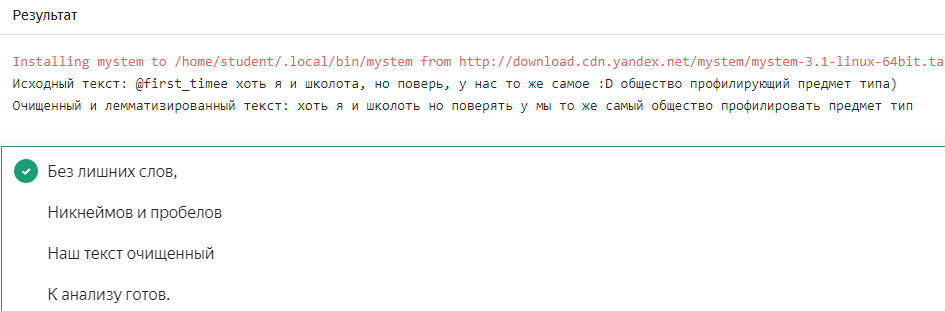
text = re.sub( r'[^а-яА-ЯёЁ ]',' ', text)

text = " ".join(text.split())

return text

print("Исходный текст:", corpus[0])

print("Очищенный и лемматизированный текст:", lemmatize(clear\_text(corpus[0])))



### Мешок слов и N-граммы

Переведём тексты в понятный для машины формат — векторный.

Преобразовать слова в векторы поможет модель **«мешок слов»** (англ. *bag* *of* *words*). Она преобразует текст в вектор, не учитывая порядок слов. Отсюда и название — «мешок».

Возьмём начало стихотворения Игоря Северянина:

Скопировать код

Ананасы в шампанском! Ананасы\*\* в шампанском!

Удивительно вкусно, искристо и остро!

Лемматизируем его:

Скопировать код

ананас в шампанский

ананас в шампанский

удивительно вкусно искристый и остро

Посчитаем количество вхождений каждого слова:

* «ананас», «в», «шампанский» — по 2 раза;
* «удивительно», «вкусно», «искристый», «и», «остро» — по 1 разу.

Получили такой результат:



Вектор этого текста:

Скопировать кодPYTHON

[2, 2, 2, 1, 1, 1, 1, 1]

Когда текстов несколько, мешок слов преобразует их в матрицу. Её строки — это тексты, а столбцы — уникальные слова из всех текстов корпуса. Числа на пересечении строк и столбцов показывают, сколько раз в тексте встречается уникальное слово.

Начало формы

Перед вами корпус. Какой вектор описывает лемматизированный текст «видеть Гpека в pека pак»:

Скопировать кодPYTHON

[

'ехать Гpека чеpез pека',

'видеть Гpека в pека pак',

'сунуть Гpека pука в pека',

'pак за pука Гpека цап'

]

Последовательность слов для вектора такая:

Скопировать кодPYTHON

['ехать', 'Гpека', 'чеpез', 'pека', 'видеть', 'в', 'pак', 'сунуть', 'pука', 'за', 'цап']



[1, 1, 1, 1, 1]



[0, 1, 0, 1, 1]



Тоже правильный ответ

[0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0]

Верно! А если составить векторы для других текстов из этой скороговорки, можно узнать, как ещё перекликаются реки, раки и руки.



Неправильный ответ

[1, 4, 1, 2, 1, 2, 2, 1, 2, 1, 1]

Каждый вектор в мешке слов соответствует одному тексту. Этот вектор описывает весь корпус.

Конец формы

В мешке слов учитывается каждое уникальное слово. Но порядок слов и связи между ними не учитываются. Например, есть такой лемматизированный текст:

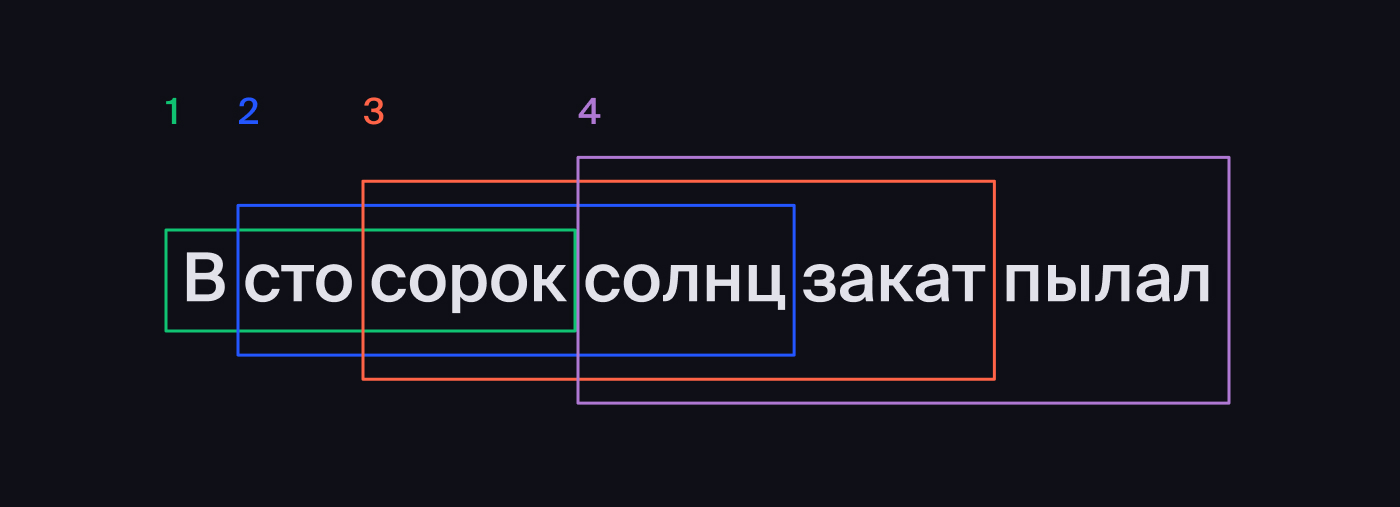
Скопировать код

Фёдор ехать из Москва в Петербург

Его набор слов: «Фёдор», «ехать», «Москва», «Петербург», «из», «в». Так куда едет Фёдор? Чтобы ответить на вопрос, посмотрим на словосочетания, или **N-граммы** (англ. *N-grams*).

*N*-грамма — это последовательность из нескольких слов. *N* указывает на количество элементов и может быть любым. Например, если *N* равно 1, получаются слова, или **униграммы** (лат. *unus*, «один»). При *N*=2 выходят словосочетания из двух слов — **биграммы** (лат. *bis*, «дважды»). Если *N*=3, то это уже **триграммы** (лат. *tres*, *\*\**«три»), т. е. из трёх слов.

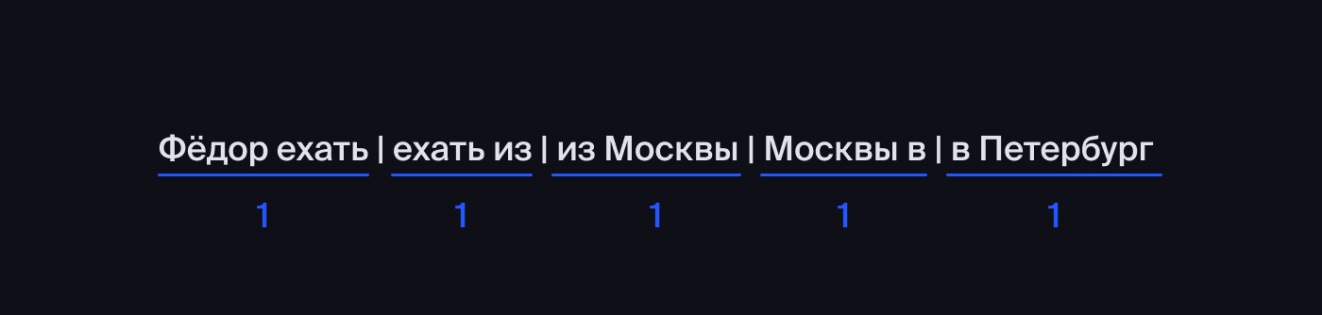
Как выглядят триграммы предложения «В сто сорок солнц закат пылал»?



Мы получили четыре триграммы: «в сто сорок», «сто сорок солнц», «сорок солнц закат», «солнц закат пылал». Слово «закат» не может быть началом следующей триграммы: после него остаётся только одно слово, а нужно два.

Вернёмся к нашему Фёдору и найдём в тексте биграмму. Получится такой набор: «Фёдор ехать», «ехать из», «из Москвы», «Москвы в», «в Петербург». Всё теперь ясно: точки А и Б найдены. Фёдор отправился из Москвы в Петербург.

Аналогично модели «мешок слов» *N*-граммы текста можно преобразовывать в векторы. Например, вектор для текста о Фёдоре выглядит так:



Начало формы

Сколько биграмм получится из цитаты Даниила Хармса. *N*-грамма не должна разбиваться точкой:

Скопировать кодPYTHON

'Предмет обезоружен. Он стручок. Вооружена только куча.'



2



3



Тоже правильный ответ

4

Слово обезоружено. Вооружены N-граммы.



Неправильный ответ

6

Насчитали больше.

### Создание мешка слов

Научимся создавать мешок слов и находить **стоп-слова** (англ. stopwords).

Чтобы преобразовать корпус текстов в мешок слов, обратимся к классу **CountVectorizer()** (англ. count vectorizer, «счётчик слов для создания векторов»). Он находится в модуле **sklearn.feature\_extraction.text** (англ. «построение признаков текста»).

Импортируем его:

Скопировать кодPYTHON

from sklearn.feature\_extraction.text import CountVectorizer

Создадим счётчик:

Скопировать кодPYTHON

count\_vect = CountVectorizer()

Передадим счётчику корпус текстов. Для этого вызовем знакомую вам функцию fit\_transform(). Счётчик выделит из корпуса уникальные слова и посчитает количество их вхождений в каждом тексте корпуса. Отдельные буквы счётчик как слова не учитывает.

Скопировать кодPYTHON

*# bow, от англ. bag of words*

bow = count\_vect.fit\_transform(corpus)

Метод вернёт матрицу, в которой одна строка — это текст, а столбец — уникальное слово из всего корпуса. Число на их пересечении покажет, сколько раз в тексте встречалось нужное слово.

Из прошлого урока возьмём корпус про Греку:

Скопировать кодPYTHON

corpus = [

'ехать Гpека чеpез pека',

'видеть Гpека в pека pак',

'сунуть Гpека pука в pека',

'pак за pука Гpека цап'

]

Создадим для него мешок слов. Чтобы получить размер матрицы, посмотрим атрибут shape:

Скопировать кодPYTHON

bow.shape

Скопировать код

(4, 10)

Всё верно, у нас 4 текста и 10 уникальных слов (предлог «в» не учитывается).

Представим мешок слов в виде матрицы:

Скопировать кодPYTHON

print(bow.toarray())

Скопировать код

array([[0, 1, 0, 0, 1, 1, 0, 0, 0, 1],

[1, 1, 0, 1, 1, 0, 0, 0, 0, 0],

[0, 1, 1, 0, 1, 0, 0, 1, 0, 0],

[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]], dtype=int64)

Список уникальных слов в мешке образует **словарь**. Он хранится в счётчике и вызывается методом **get\_feature\_names()** (англ. «получить имена признаков»):

Скопировать кодPYTHON

count\_vect.get\_feature\_names()

В корпусе про Греку и рака словарь такой:

Скопировать кодPYTHON

['pак',

'pека',

'pука',

'видеть',

'гpека',

'ехать',

'за',

'сунуть',

'цап',

'чеpез']

CountVectorizer() также нужен для расчёта N-грамм. Чтобы он считал словосочетания, укажем размер N-граммы через аргумент **ngram\_range** (англ. «диапазон N-грамм»). Например, если мы ищем словосочетания по два слова в фразе, то диапазон зададим такой:

Скопировать кодPYTHON

count\_vect = CountVectorizer(ngram\_range=(2, 2))

Со словосочетаниями счётчик работает так же, как и со словами.

У больших корпусов и мешки слов выходят большие, но часть слов в них может быть бессмысленной. Например, что можно сказать о тексте по местоимениям, союзам и предлогам? Чаще всего от них можно избавиться, причём тема текста и смысл предложения не изменятся. Когда мешок слов меньше и чище, проще найти слова, важные для классификации текстов.

Чтобы почистить мешок слов, найдём **стоп-слова**, то есть слова без смысловой нагрузки. Их много, и для каждого языка — свои. Разберём пакет stopwords, который находится в модуле nltk.corpus библиотеки **nltk** (англ. Natural Language Toolkit, «инструментарий естественного языка»):

Скопировать кодPYTHON

from nltk.corpus import stopwords

Чтобы пакет заработал, загрузим список стоп-слов:

Скопировать кодPYTHON

import nltk

nltk.download('stopwords')

Вызовем функцию **stopwords.words()**, передадим ей аргумент **'russian'**, то есть русскоязычные стоп-слова:

Скопировать кодPYTHON

stop\_words = set(stopwords.words('russian'))

При создании счётчика передадим список стоп-слов в счётчик векторов CountVectorizer():

Скопировать кодPYTHON

count\_vect = CountVectorizer(stop\_words=stop\_words)

Теперь счётчик знает, какие слова нужно исключить из мешка слов.

Конец формы

#### Задача

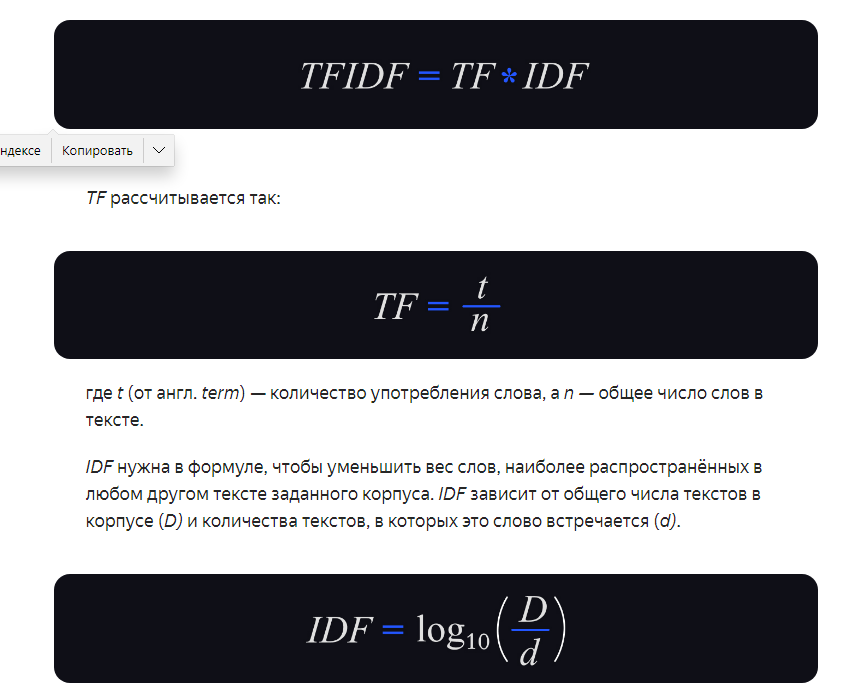
|  |  |
| --- | --- |
| 1.  В файле tweets\_lemm.csv находится датасет, в который мы добавили столбец *lemm\_text* с очищенными и лемматизированными текстами.  Создайте два мешка слов для корпуса твитов: очищенный от стоп-слов и с ними. Напечатайте их размеры на экране (уже в прекоде). | import pandas as pd  from sklearn.feature\_extraction.text import CountVectorizer  from nltk.corpus import stopwords  # < напишите код здесь >  data = pd.read\_csv("/datasets/tweets\_lemm.csv")  corpus = data['lemm\_text'].values.astype('U')  # создайте мешок слов без учёта стоп-слов  count\_vect = CountVectorizer()  bow = count\_vect.fit\_transform(corpus)  # < напишите код здесь >  print("Размер мешка без учёта стоп-слов:", bow.shape)  # создайте новый мешок слов с учётом стоп-слов  stop\_words = set(stopwords.words('russian'))  count\_vect = CountVectorizer(stop\_words=stop\_words)  bow = count\_vect.fit\_transform(corpus)  # < напишите код здесь >  print("Размер мешка с учётом стоп-слов:", bow.shape) |
| 2.  Создайте счётчик *N*-грамм для корпуса твитов. В каждой фразе по два слова. Напечатайте на экране размер *N*-грамм (уже в прекоде). | import pandas as pd  from sklearn.feature\_extraction.text import CountVectorizer  # импортируйте CountVectorizer  data = pd.read\_csv("/datasets/tweets\_lemm.csv")  corpus = list(data['lemm\_text'])  count\_vect = CountVectorizer(ngram\_range=(2, 2))  n\_gramm = count\_vect.fit\_transform(corpus)  # создайте n-грамму n\_gramm, для которой n=2  print("Размер:", n\_gramm.shape) |

### TF-IDF

Мешок слов учитывает частоту употребления слов. Посмотрим, как часто уникальное слово встречается во всём корпусе и в отдельном его тексте.

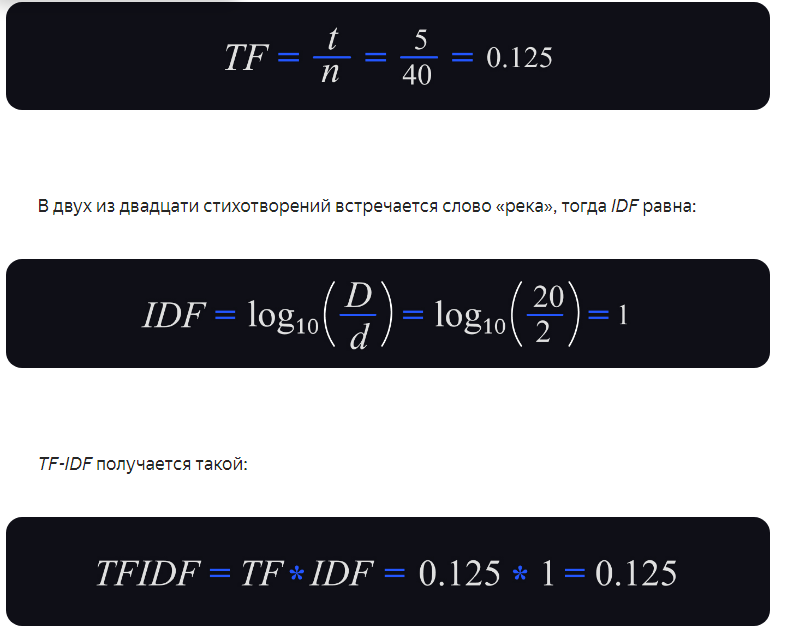
Оценка важности слова определяется величиной **TF-IDF** (от англ. term frequency, «частота терма, или слова»; inverse document frequency, «обратная частота документа, или текста»). То есть TF отвечает за количество упоминаний слова в отдельном тексте, а IDF отражает частоту его употребления во всём корпусе.

Формула TF-IDF такая:

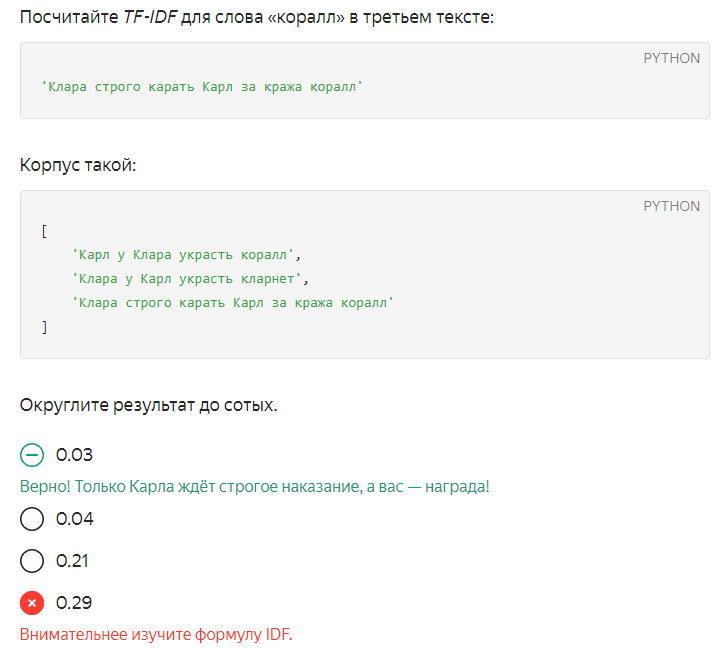


Например, рассмотрим корпус из 20-ти стихотворений. Возьмём первое: в нём 40 слов, нам интересно слово «река». В стихе оно встречается пять раз. Всего в корпусе два стихотворения с «рекой». Рассчитаем *TF-IDF* для слова «река» в первом стихотворении корпуса.

*TF* равна:



Большая величина TF-IDF говорит об уникальности слова в тексте по отношению к корпусу. Чем чаще оно встречается в конкретном тексте и реже в остальных, тем выше значение TF-IDF.



### TF-IDF в sklearn

Вычислим TF-IDF для корпуса текстов.

Рассчитать TF-IDF можно и в библиотеке sklearn. Класс **TfidfVectorizer()** (от англ. «счётчик величин TF-IDF») находится в модуле sklearn.feature\_extraction.text. Импортируем его:

from sklearn.feature\_extraction.text import TfidfVectorizer

По аналогии с CountVectorizer() создадим счётчик, указав в нём стоп-слова:

count\_tf\_idf = TfidfVectorizer(stop\_words=stopwords)

Чтобы посчитать TF-IDF для корпуса текстов, вызовем функцию fit\_transform():

tf\_idf = count\_tf\_idf.fit\_transform(corpus)

Передав TfidfVectorizer() аргумент ngram\_range, можно рассчитать N-граммы.

Если данные разделены на обучающую и тестовую выборки, функцию fit() запускайте только на обучающей. Иначе тестирование будет нечестным: в модели будут учтены частоты слов из тестовой выборки.

#### Задача

Создайте матрицу cо значениями TF-IDF по корпусу твитов. Сохраните её в переменной tf\_idf. Укажите стоп-слова. Напечатайте на экране размер матрицы (уже в прекоде).

import pandas as pd

import nltk

from nltk.corpus import stopwords as nltk\_stopwords

from sklearn.feature\_extraction.text import TfidfVectorizer

data = pd.read\_csv("/datasets/tweets\_lemm.csv")

corpus = data['lemm\_text'].values.astype('U')

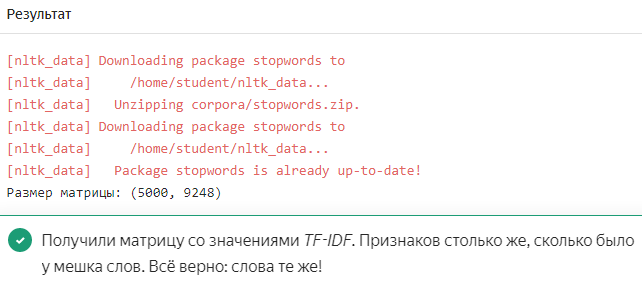
nltk.download('stopwords')

stopwords = set(nltk\_stopwords.words('russian'))

count\_tf\_idf = TfidfVectorizer(stop\_words=stopwords)

tf\_idf = count\_tf\_idf.fit\_transform(corpus)

print("Размер матрицы:", tf\_idf.shape)



### Классификация тональности текста

Для определения тональности применим величины TF-IDF как признаки.

Анализ тональности текста, или **сентимент-анализ** (от англ. sentiment, «настроение»), выявляет эмоционально окрашенные слова. Этот инструмент помогает компаниям оценивать, например, реакцию на запуск нового продукта в интернете. На разбор тысячи отзывов человек потратит несколько часов, а компьютер — пару минут.

Оценить тональность — значит отметить текст как позитивный или негативный. То есть мы решаем задачу классификации, где целевой признак равен «1» для положительного текста и «0» для отрицательного. Признаки — это слова из корпуса и их величины TF-IDF для каждого текста.

#### Задача

Обучите логистическую регрессию так, чтобы она определяла тональность текста.

Подсчитайте величину TF-IDF для текстов. Лемматизированные тексты твитов для обучения находятся в файле tweets\_lemm\_train.csv. Целевой признак вы найдёте в столбце positive.

Обученной моделью классификации определите результаты предсказания для тестовой выборки твитов, которая лежит в файле tweets\_lemm\_test.csv. В этой выборке целевого признака нет. Сохраните предсказания в столбце positive. Таблицу с результатом сохраните как csv-файл, но чтобы тренажёр принял файл, не указывайте расширение (например, назовите файл 'predictions')

Значение accuracy вашей модели должно быть не меньше 0.62.

Файлы: [tweets\_lemm\_train](https://code.s3.yandex.net/datasets/tweets_lemm_train.csv) , [tweets\_lemm\_test](https://code.s3.yandex.net/datasets/tweets_lemm_test.csv).

**Подсказка**

Для предсказания ответов повторно вычислите величину *TF-IDF* для вектора с тестовой выборкой. Примените метод *transform()* к объекту *TfidfVectorizer.*

### Embeddings

Научимся учитывать смысл, контекст и свойства слов при переводе в векторы. Мешок слов и TF-IDF этого не делали.

Чтобы машины воспринимали слова, картинки или аудио, их преобразовывают в векторный вид. Когда работают с текстом, его тоже переводят в векторный формат, или векторные представления. Частный случай этих представлений — **word** **embeddings** (англ. «слова-вложения»; «эмбеддинги»). Работают они так: сложная структура (текст) вкладывается в более простую — вектор.

Векторы-эмбеддинги содержат данные о соотношении разных слов и их свойствах. Привычное понимание свойства слова, его смысла и контекста справедливо и для машинного обучения.

Свойства — это скрытые смыслы слова. Допустим, «моряк» содержит смыслы: «мужчина», «профессия», «человек» и «морской». Смысл, или семантика слова, — это лексическое значение слова, его отличие от других слов. Но слово обычно не живёт само по себе, его окружают другие слова. Например, «лента» во фразе: «Красная лента в каштановых волосах» отличается от «ленты» из предложения: «Положите продукты на ленту перед кассой». Именно контекст и определяет смысл слов.

Эти понятия пригодятся нам в определении близости слов в их векторном представлении. Так **близкие**, или **похожие векторы**, могут отображать похожие слова. Сходство векторов рассчитывается знакомым вам евклидовым расстоянием: чем меньше расстояние, тем сильнее похожи векторы.

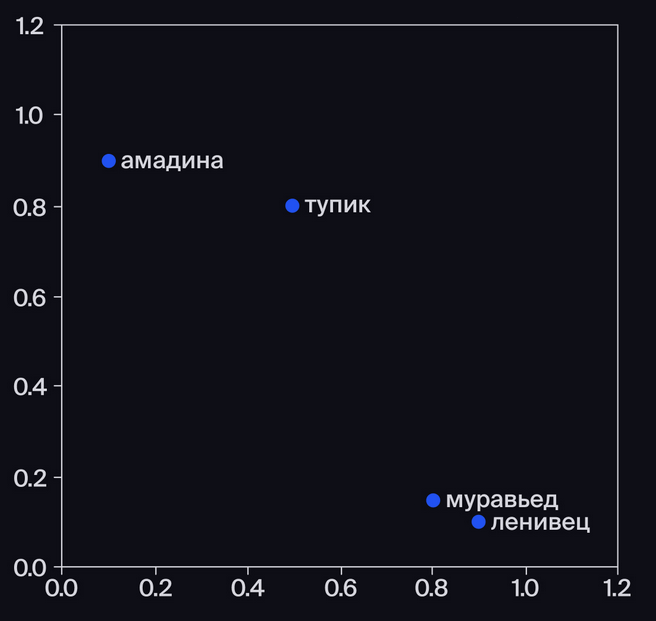
Чтобы определить близкие и дальние векторы, визуализируем векторы размерностью 2 для таких слов: «амадина», «ту́пик», «муравьед» и «ленивец».

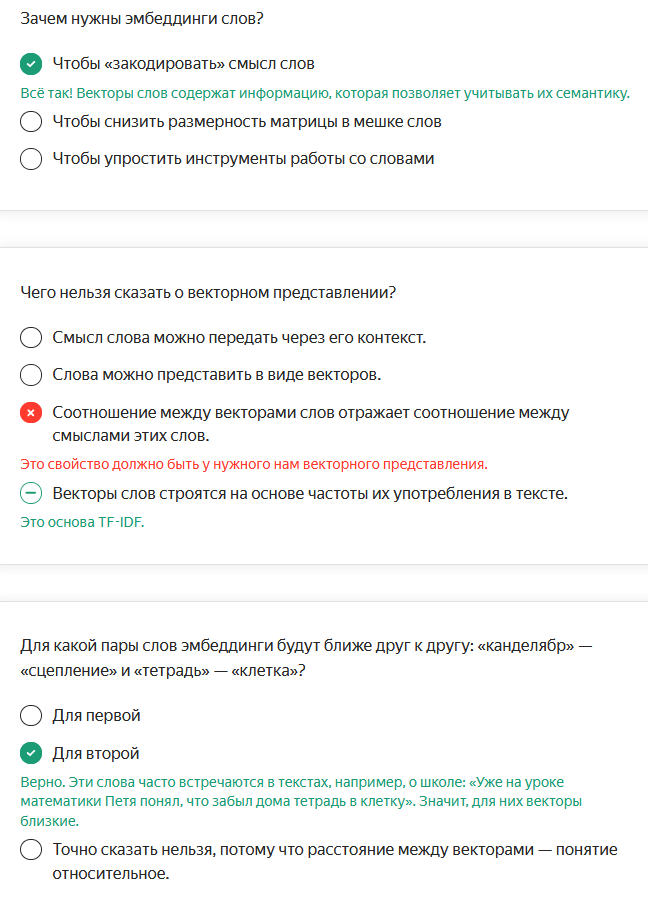
В этом уроке нам неважно, как получили эти векторы. Главное — понять, какими свойствами они обладают и как соотносятся с контекстом.

«Амадина» и «тупик» явно ближе друг к другу, чем вторая пара: «муравьед» и «ленивец». Каждое слово из первой пары можно подставить во фразу «Красный клюв \_\_\_\_ мелькнул на голубом небе». Так и есть, и «амадина», и «тупик» могут летать. А вот «муравьед» и «ленивец» — нет.

«Ленивец» и «муравьед» больше подходят для предложения: «Хуан постоянно рассказывал о нелепых животных Перу. Например, о странных \_\_\_\_ ». То есть похожесть как первой, так и второй пары векторов определяется именно их окружением. Если векторы близки, то они легко соотносятся с одним и тем же контекстом.

Можно предположить, что первая размерность векторов отвечает за способность летать, а вторая — за принадлежность к отряду неполнозубых или волосатость. Но на практике интерпретировать векторы как признаки не всегда корректно.





### Word2vec

Рассмотрим популярный метод построения языковых представлений — **word2vec** (от англ. word to vector, «слова к вектору»).

Упрощённо разберём, как работает word2vec. Из примера с амадиной и тупиком вы узнали, что сходство этих слов определяется соседством с «красным клювом». А «муравьед» и «ленивец» часто встречаются в предложениях о Перу. То есть **смысл** слов определяется их контекстом.

Тогда задача word2vec — предсказать: соседи или нет — заданные слова. Слова считаются соседями, если находятся в одном **«окне»** (максимальном расстоянии между словами). Выходит, пара слов — это признаки, а являются ли они соседями — целевой признак.

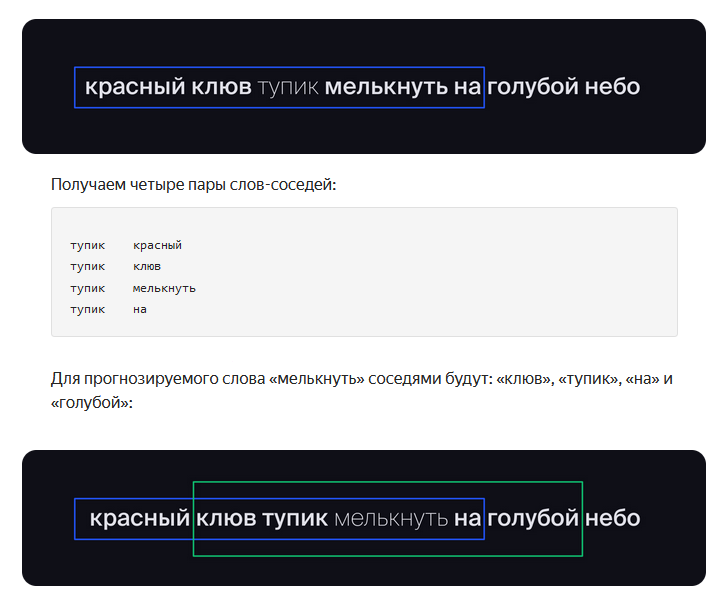
Пусть датасетом для задачи будет предложение о тупике:

Красный клюв тупика мелькнул на голубом небе

Но прежде лемматизируем текст:

красный клюв тупик мелькнуть на голубой небо

Будем считать, что прогнозируемое слово — тупик, а соседи — ближайшие от него слова с двух сторон: «красный», «клюв» и «мелькнуть», «на». Все вместе они образуют пятиграмму:



С учётом предыдущих пар получаем такие пары слов-соседей:

тупик красный

тупик клюв

тупик мелькнуть

тупик на

мелькнуть клюв

мелькнуть тупик

мелькнуть на

мелькнуть голубой

Таким же способом можно «пройтись окном» по всему тексту и составить полный список пар слов-соседей.

Следующая задача *word2vec —* научить модель отличать истинные пары соседей от случайных. А это уже походит на задачу бинарной классификации, где признаки — это слова, а целевой признак — ответ на вопрос: перед нами истинные слова-соседи или нет.

Положительные примеры мы уже собрали. Чтобы получить отрицательные, выберем случайные слова из корпуса и сделаем из них пары:

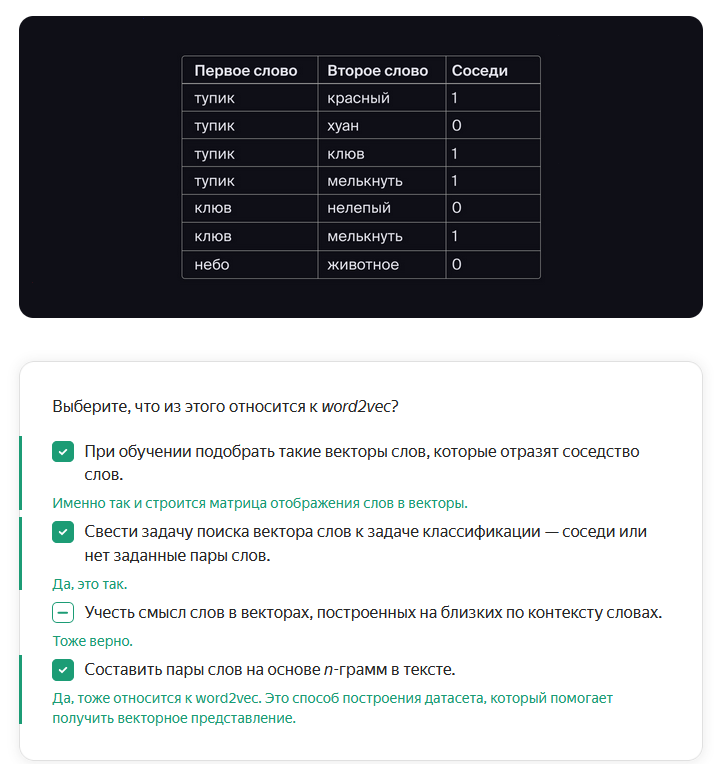
тупик хуан

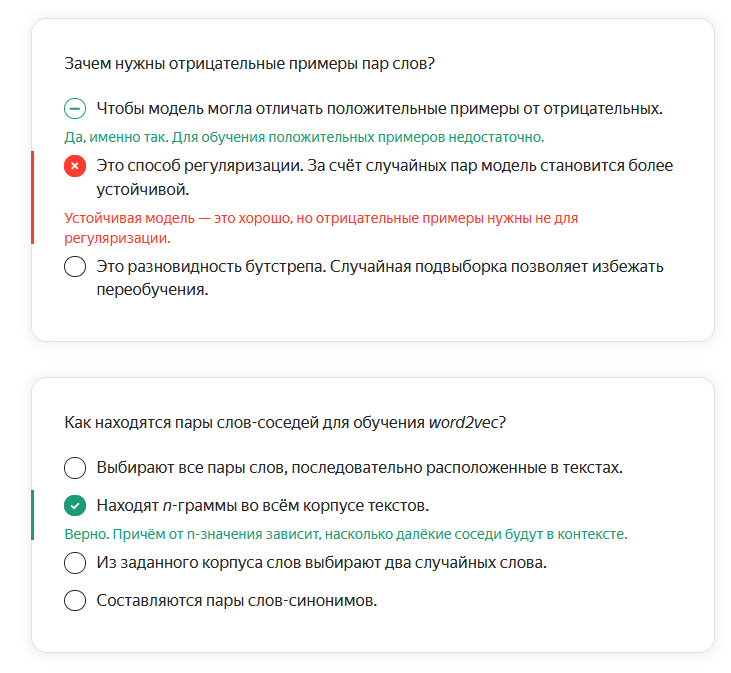
клюв нелепый

небо животное

красный рассказывать

Тогда получим такой набор данных для обучения, где метка «1» говорит, что слова-соседи — истинные, а «0» — ложные.





### Embeddings для классификации

Разберём, как векторное представление помогает решать задачи классификации и регрессии.

Применим векторное представление к задаче анализа текстов. Допустим, у нас есть корпус текстов, которые нужно классифицировать. Тогда наша модель будет состоять из двух блоков:

1. Модели перевода слов в векторное представление: исходные тексты преобразуются в векторы.
2. Модели классификации: на основе векторов текстов получаются прогнозы.



Разберём, как работает эта схема:

1. Прежде чем перейти к векторному представлению слов, проводится знакомая вам предобработка текста:
2. Выполняют токенизацию каждого текста, то есть его разбивают на слова;
3. Слова лемматизируют: приводят к начальной словарной форме (более сложные модели, например, *BERT*, этого не требуют: они сами понимают формы слов);
4. Текст очищают от стоп-слов и ненужных символов;
5. Для корректной работы алгоритма добавляют маркеры начала и конца предложения (они приравниваются к токенам).
6. На выходе у каждого исходного текста образуется свой список токенов.
7. Затем токены передают модели, которая переводит их в векторные представления. Для этого модель обращается к составленному заранее словарю токенов. На выходе для каждого текста образуются векторы заданной длины.
8. На финальном этапе модели передают признаки (векторы). И она прогнозирует эмоциональную окраску текста — 0 («отрицательная») или 1 («положительная»).

### BERT

Познакомимся с моделью BERT и узнаем, как она помогает переводить слова в векторы.

**BERT** (от англ. Bidirectional Encoder Representations from Transformers, «двунаправленная нейронная сеть-кодировщик») — нейронная сеть для создания модели языка. Её разработали в компании Google, чтобы повысить релевантность результатов поиска. Этот алгоритм понимает контекст запросов, а не просто анализирует фразы. Для машинного обучения она ценна тем, что помогает строить векторные представления. Причём в анализе текстов применяют уже предобученную на большом корпусе модель. Такие предобученные версии BERT годятся для работы с текстами на 104 языках мира, включая русский.

BERT — это результат эволюции модели word2vec. В ходе её развития были придуманы и другие модели: **FastText** (англ. «быстрый текст»), **GloVe** (англ. Global Vectors for Word Representation, «глобальные векторы для языкового представления»), **ELMO** (англ. Embeddings from Language Models, «вложения языковых моделей») и **GPT** (англ. Generative Pre-Training Transformer, «предобученный трансформер для генерации»). Сейчас самые точные — это BERT и GPT-3, которого нет в открытом доступе.

*BERT* учитывает контекст не только соседних слов, но и более дальних родственников. Работает так:

* На входе модель получает, например, такую фразу: «Красный клюв тупика *[MASK]* на голубом *[MASK]*», где **MASK** (англ. «маска») — это неизвестные слова, будто закрытые маской. Модель должна угадать эти спрятанные слова.
* Модель обучается определять, связаны ли в предложении слова между собой. У нас были скрыты такие слова: «мелькнул» и «небе». Модель должна понять, что одно слово — продолжение другого. Скажем, если вместо «мелькнул» спрятать слово «прополз», то связи модель не найдёт.

### RuBERT и предобработка

Решим задачу классификации твитов, применив векторные представления на базе BERT. Построить векторы текстов нам поможет предобученная на русских текстах модель RuBERT.

В этом уроке вы будете работать с чужим готовым кодом, менять который не нужно. Ваша задача — добиться, чтобы он работал. Ничего страшного, если вы не научитесь писать этот код самостоятельно. Для быстрого решения задачи специалисты по Data Science часто копируют чужой код.

Повторим задачу. Перед вами большой датасет с твитами. Нужно научиться определять, какие твиты негативной тональности, а какие — позитивной. Чтобы решить эту задачу, из открытого репозитория [DeepPavlov](http://docs.deeppavlov.ai/en/master/features/models/bert.html) возьмём модель RuBERT, обученную на разговорном русскоязычном корпусе.

Решим эту задачу на **PyTorch** (англ. «факел для Python»). Глубоко разбираться в средствах этой библиотеки мы не будем. Она применяется в задачах обработки естественного текста и компьютерного зрения. А нам нужна для работы с моделями типа BERT. Они находятся в библиотеке **transformers** (англ. «трансформеры»). Импортируем их:

import torch

import transformers

Прежде чем перевести тексты в векторы, подготовим их. У *RuBERT* есть собственный **токенизатор**. Это инструмент, который разбивает и преобразует исходные тексты в список токенов, которые есть, например, в словаре RuBERT. Лемматизация не требуется.

Начинаем предобработку текстов:

1. Инициализируем токенизатор как объект класса *BertTokenizer()*. Передадим ему аргумент *vocab\_file* — это файл со словарём, на котором обучалась модель. Он может быть, например, в текстовом формате (*txt*).

tokenizer = transformers.BertTokenizer(vocab\_file='/datasets/ds\_bert/vocab.txt')

1. Преобразуем текст в номера токенов из словаря методом **encode()** (англ. «закодировать»):

tokenizer.encode('Очень удобно использовать уже готовый трансформатор текста', add\_special\_tokens=True)

Получаем:

[101, 1094, 4980, 3373, 1034, 6037, 323, 73634, 10316, 102]

Для корректной работы модели мы указали аргумент add\_special\_tokens (англ. «добавить специальные токены»), равный True. Это значит, что к любому преобразуемому тексту добавляется токен начала (101) и токен конца текста (102).

1. Применим метод **padding** (англ. «отступ»), чтобы после токенизации длины исходных текстов в корпусе были равными. Только при таком условии будет работать модель *BERT*. Пусть стандартной длиной вектора *n* будет длина наибольшего во всём датасете вектора. Остальные векторы дополним нулями:f

vector = tokenizer.encode('Очень удобно использовать уже готовый трансформатор текста', add\_special\_tokens=True)

n = 280

# англ. вектор с отступами

padded = np.array(vector + [0]\*(n - len(vector)))

print(padded)

Получаем:

[101, 1094, 4980, 3373, 1034, 6037, 323, 73634, 10316, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Теперь поясним модели, что нули не несут значимой информации. Это нужно для компоненты модели, которая называется «внимание» (англ. attention). Отбросим эти токены и «создадим маску» для действительно важных токенов, то есть укажем нулевые и не нулевые значения:

attention\_mask = np.where(padded != 0, 1, 0)

print(attention\_mask.shape)

Получаем:

(280,)

### Эмбеддинги RuBERT

Почти всё готово, чтобы сформировать векторы моделью RuBERT и классифицировать твиты. В дело вступают тензоры.

Инициализируем конфигурацию **BertConfig** (англ. Bert Configuration). В качестве аргумента передадим ей JSON-файл с описанием настроек модели. **JSON** (англ. JavaScript Object Notation, «объектная запись JavaScript») — это организованный по ключам поток цифр, букв, двоеточий и фигурных скобок, который возвращает сервер при запросе.

Затем инициализируем саму модель класса **BertModel**. Передадим ей файл с предобученной моделью и конфигурацией:

Затем инициализируем саму модель класса **BertModel**. Передадим ей файл с предобученной моделью и конфигурацией:

config = transformers.BertConfig.from\_json\_file(

'/datasets/ds\_bert/bert\_config.json')

model = transformers.BertModel.from\_pretrained(

'/datasets/ds\_bert/rubert\_model.bin', config=config)

Начнём преобразование текстов в эмбеддинги. Это может занять несколько минут, поэтому подключим библиотеку **tqdm** (араб. *taqadum*, تقدّم, «прогресс»). Она нужна, чтобы наглядно показать индикатор прогресса. В *Jupyter* применим функцию *notebook()* из этой библиотеки:

from tqdm import notebook

Эмбеддинги модель *BERT* создаёт батчами. Чтобы хватило оперативной памяти, сделаем размер батча небольшим:

batch\_size = 100

Сделаем цикл по батчам. Отображать прогресс будет функция *notebook()*:

# сделаем пустой список для хранения эмбеддингов твитов

embeddings = []

for i in notebook.tqdm(range(padded.shape[0] // batch\_size)):

...

Преобразуем данные в формат **тензоров** (англ. *tensor*) — многомерных векторов в библиотеке *torch*. Тип данных *LongTensor* (англ. «длинный тензор») хранит числа в «длинном формате», то есть выделяет на каждое число 64 бита.

# преобразуем данные

batch = torch.LongTensor(padded[batch\_size\*i:batch\_size\*(i+1)])

# преобразуем маску

attention\_mask\_batch = torch.LongTensor(attention\_mask[batch\_size\*i:batch\_size\*(i+1)])

Чтобы получить эмбеддинги для батча, передадим модели данные и маску:

batch\_embeddings = model(batch, attention\_mask=attention\_mask\_batch)

Для ускорения вычисления функцией **no\_grad()** (англ. *no gradient*, «нет градиента») в библиотеке *torch* укажем, что градиенты не нужны: модель *BERT* обучать не будем.

with torch.no\_grad():

batch\_embeddings = model(batch, attention\_mask=attention\_mask\_batch)

Из полученного тензора извлечём нужные элементы и добавим в список всех эмбеддингов:

# преобразуем элементы методом numpy() к типу numpy.array

embeddings.append(batch\_embeddings[0][:,0,:].numpy())

Получаем такой цикл:

embeddings = []

for i in notebook.tqdm(range(padded.shape[0] // batch\_size)):

batch = torch.LongTensor(padded[batch\_size\*i:batch\_size\*(i+1)])

attention\_mask\_batch = torch.LongTensor(attention\_mask[batch\_size\*i:batch\_size\*(i+1)])

with torch.no\_grad():

batch\_embeddings = model(batch, attention\_mask=attention\_mask\_batch)

embeddings.append(batch\_embeddings[0][:,0,:].numpy())

Соберём все эмбеддинги в матрицу признаков вызовом функции *concatenate()*:

features = np.concatenate(embeddings)

Признаки готовы, можно обучать модель!