

# Конспект по теме "Векторизация текста"

## Лемматизация

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 *py morphology2*, *UDPipe*, *py mystem3*.

Создадим класс для лемматизации:

```
from pymystem3 import Mystem
m = Mystem()
```

Функции *lemmatize()* передадим «лемматизируй это»:

```
m.lemmatize("лемматизируй это")
```

Функция вернула лемму каждого слова. Для функции конец строки — это тоже знак, поэтому находится и его лемма `(\n)`:

```
['лемматизировать', ' ', 'это', '\n']
```

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

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

Получаем:

```
'лемматизировать это \n'
```

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

Создадим корпус: преобразуем столбец *text* в список текстов.

Переведём тексты в стандартный для Python формат: кодировку *Unicode* (*U*).

Изменим кодировку методом *astype()*:

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

## 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.

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

```
import re
```

Познакомимся с функцией **re.sub()**. Она находит в тексте все совпадения по шаблону и заменяет их заданной строкой.

```
# pattern – шаблон
# replacement – на что заменять
# text – текст, в котором искать совпадения
re.sub(pattern, replacement, text)
```

Пусть в тексте нужно оставить только кириллические символы и пробелы. Чтобы их найти, напишем регулярное выражение.

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

```
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:

```
# диапазон букв обозначается дефисом:  
# а-я — это то же самое, что абвгдежзийклмнопрстуфхцщъыьэюя  
r'[а-яА-Я]'
```

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

```
r'[а-яА-ЯёЁ]'
```

Возьмём исходный текст. 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» (^):

```
# уже лемматизированный текст  
re.sub(r'^[а-яА-ЯёЁ ]', ' ', text)
```

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()*. Методом *split()* преобразуем этот текст в список. Если не указывать аргументы у *split()*, он делает разбиение по пробелам или группам пробелов:

```
text.split()
```

Получили список без пробелов. Методом *join()* объединим элементы в строку через пробел:

```
" ".join(text.split())
```

Пробелов теперь ровно столько, сколько нужно.

## 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:

Ананасы в шампанском! Ананасы в шампанском!  
Удивительно вкусно, искристо и остро!

Лемматизируем его:

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:

<u>ананас</u>	<u> </u>	<u>в</u>	<u> </u>	<u>шампанский</u>	<u> </u>	<u>удивительно</u>	<u> </u>	<u>вкусно</u>	<u> </u>	<u>искристый</u>	<u> </u>	<u>и</u>	<u> </u>	<u>остро</u>
2		2		2		1		1		1		1		1

Вектор этого текста:

```
[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.

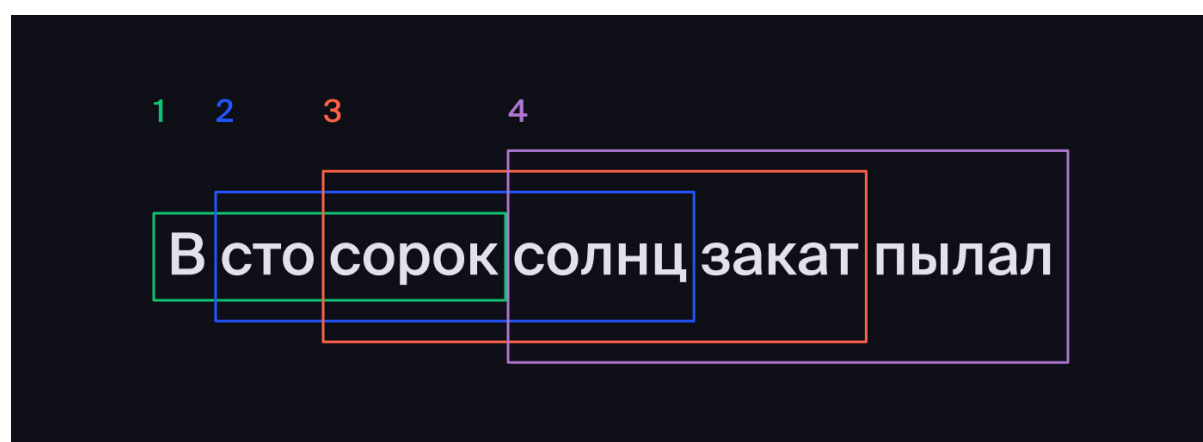
В мешке слов учитывается каждое уникальное слово. Но порядок слов и связи между ними не учитываются. Например, есть такой лемматизированный текст:

```
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.

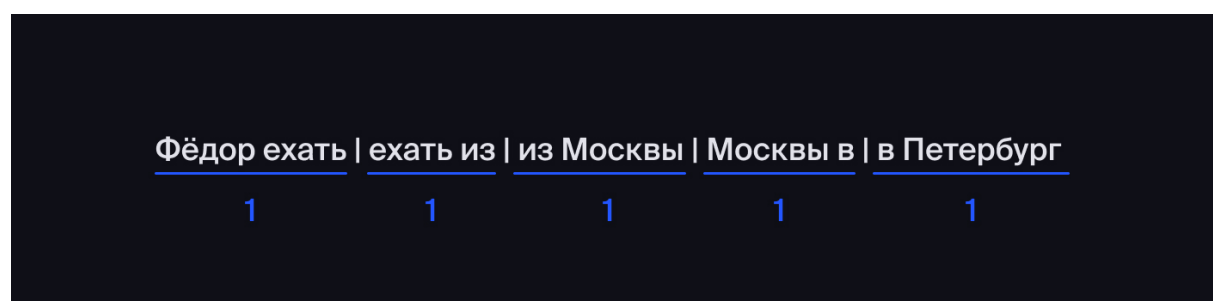
Как выглядят триграммы предложения «В сто сорок солнц закат пылал»?



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:



Фёдор ехать | ехать из | из Москвы | Москвы в | в Петербург

1 1 1 1 1

## Создание мешка слов

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

Чтобы преобразовать корпус текстов в мешок слов, обратимся к классу **CountVectorizer()**. Он находится в модуле **sklearn.feature\_extraction.text**.

Импортируем его:

```
from sklearn.feature_extraction.text import CountVectorizer
```

Создадим счётчик:

```
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.

```
# 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.

Возьмём корпус про Греку:

```
corpus = [
    'ехать Грека через река',
    'видеть Грека в река рак',
    'сунуть Грека рука в река',
    'рак за рука Грека цап'
]
```

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

```
bow.shape
```

```
(4, 10)
```

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

Представим мешок слов в виде матрицы:

```
print(bow.toarray())
```

```
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()**:

```
count_vect.get_feature_names()
```

В корпусе про Греку и рака словарь такой:

```
['рак',  
'река',  
'рука',  
'видеть',  
'грека',  
'ехать',  
'за',  
'сунуть',  
'цап',  
'через']
```

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

```
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*):

```
from nltk.corpus import stopwords
```

Чтобы пакет заработал, загрузим список стоп-слов:



```
import nltk
nltk.download('stopwords')
```

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

```
stop_words = set(stopwords.words('russian'))
```

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

```
count_vect = CountVectorizer(stop_words=stop_words)
```

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

## TF-IDF

Оценка важности слова определяется величиной **TF-IDF**. То есть *TF* отвечает за количество упоминаний слова в отдельном тексте, а *IDF* отражает частоту его употребления во всём корпусе.

Формула *TF-IDF* такая:

$$TFIDF = TF * IDF$$

*TF* рассчитывается так:

$$TF = \frac{t}{n}$$

где *t* — количество употребления слова, а *n* — общее число слов в тексте.

*IDF* нужна в формуле, чтобы уменьшить вес слов, наиболее распространённых в любом другом тексте заданного корпуса. *IDF* зависит от общего числа текстов в корпусе ( $D$ ) и количества текстов, в которых это слово встречается ( $d$ ).

$$IDF = \log_{10}\left(\frac{D}{d}\right)$$

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

Рассчитать *TF-IDF* можно и в библиотеке *sklearn*. Класс **TfidfVectorizer()** 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()* запускаяйте только на обучающей. Иначе тестирование будет нечестным:

в модели будут учтены частоты слов из тестовой выборки.

## Классификация тональности текста

Анализ тональности текста, или **сентимент-анализ**, 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.