

Обработка естественного языка

[Материалы](#) > [Вводный курс](#)

Сегодня мы рассмотрим тему **обработки естественного языка** (Natural Language Processing, NLP), области на стыке математики, информатики и лингвистики, занимающейся пониманием (анализом) и созданием (синтезом) текстов с помощью компьютера.

В частности мы разберем тему (1) предварительной обработки языка (language pre-processing), (2) а также изучим два несложных способа **анализа содержания текста** (topic identification).

Постановка задачи

В школьных учебниках можно встретить такое задание: «Прочитайте текст. Определите его тему». У человека, как правило, это не вызывает никаких сложностей. Попробуем привить этот навык компьютеру.

В качестве примера, возьмем следующий текст на английском языке:

When we were in Paris we visited a lot of museums. We first went to the Louvre, the largest art museum in the world. I have always been interested in art so I spent many hours there. The museum is enourmous, so a week there would not be enough.

Содержание этого текста можно описать словами «музей» (museum), «Лувр» (Louvre), «Париж» (Paris), «искусство» (art). Посмотрим, что скажет компьютер.

Откроем блокнот к этому занятию📓

```
1 # Возьмём исходный текст для анализа
2 corpus = 'When we were in Paris we visited a lot of museums. We first went to the Louvre'
```

В лингвистике совокупность рассматриваемых текстов принято называть **корпусом** (corpus, мн.ч. ко́рпусы, corpога).

Импортируем пакет библиотек для обработки естественного языка (Natural Language Toolkit или NLTK) и другие уже известные нам библиотеки.

```
1 import nltk
2 import pandas as pd
```

```
3 | import numpy as np
```

Предварительная обработка текста

Шаг 1. Разделение на предложения

Вначале текст логично разделить на предложения или токенизировать. Хотя задача кажется тривиальной (достаточно разделить текст по точкам, восклицательным и вопросительным знакам), есть несколько сложностей. Например, фраза «Мороженое стоит 100 руб. 20 коп.» может быть ошибочно разбита на два предложения. Помимо сокращений есть и другие сложности. Например, из-за опечатки предложение может заканчиваться пробелом, а не знаком препинания.

Для решения этой задачи можно использовать функцию библиотеки NLTK `sent_tokenize()`.

```
1 | # импортируем функцию sent_tokenize()
2 | from nltk.tokenize import sent_tokenize
3 |
4 | # скачиваем модель, которая будет делить текст на предложения
5 | nltk.download('punkt')
6 |
7 | # и применяем функцию к тексту
8 | sentences = sent_tokenize(corpus)
9 | sentences
```

```
1 | ['When we were in Paris we visited a lot of museums.',
2 |  'We first went to the Louvre, the largest art museum in the world.',
3 |  'I have always been interested in art so I spent many hours there.',
4 |  'The museum is enourmous, so a week there would not be enough.']
```

Функция использует уже обученную модель, в нашем случае, для английского языка:

`nltk_data/tokenizers/punkt/english.pickle`. Также можно использовать модели для других языков или обучить собственную модель.

Шаг 2. Разделение на слова

Разделение на слова или токенизация слов — следующий шаг в обработке текста. В целом используется аналогичный подход и функция `word_tokenize()`. Для примера, разобьем на слова первое предложение.

```
1 | # импортируем функцию word_tokenize()
2 | from nltk.tokenize import word_tokenize
3 |
4 | # и разобьем на слова первое предложение
5 | print(word_tokenize(sentences[0]))
```

```
1 | ['When', 'we', 'were', 'in', 'Paris', 'we', 'visited', 'a', 'lot', 'of', 'museums', '.']
```

Сделаем то же самое для всего текста.

```
1 | # для этого создадим пустой список
2 | tokens = []
3 |
4 | # в цикле for пройдемся по каждому предложению
5 | for sentence in sentences:
6 |
7 |     # создадим списки из токенов
```

```
8     t = word_tokenize(sentence)
9
10    # и присоединим списки друг к другу
11    tokens.extend(t)
12
13    print(tokens)
```

```
1    ['When', 'we', 'were', 'in', 'Paris', 'we', 'visited', 'a', 'lot', 'of', 'museums', '.']
```

Шаг 3. Перевод в нижний регистр, удаление стоп-слов и знаков пунктуации

Выясняется, что для анализа текста далеко не все слова являются релевантными. Если мы попробуем применить к тексту статистические методы, то многие важные с точки зрения человеческой грамматики слова будут только мешать. В частности, это предлоги, союзы и артикли. Назовем их «стоп-словами» и попробуем опустить.

Перед этим обязательно преобразуем все заглавные буквы в строчные.

```
1    # импортируем модуль стоп-слов
2    from nltk.corpus import stopwords
3
4    # скачаем словарь стоп-слов
5    nltk.download('stopwords')
6
7    # мы используем set, чтобы оставить только уникальные значения
8    unique_stops = set(stopwords.words('english'))
9
10   # создаём пустой список без стоп-слов
11   no_stops = []
12
13   # проходимся по всем токенам
14   for token in tokens:
15
16       # переводим все слова в нижний регистр
17       token = token.lower()
18
19       # если токен не в списке стоп-слов и не является знаком пунктуации,
20       if token not in unique_stops and token.isalpha():
21
22           # добавляем его в список
23           no_stops.append(token)
24
25   print(no_stops)
```

```
1    ['paris', 'visited', 'lot', 'museums', 'first', 'went', 'louvre', 'largest', 'art', 'mu
```

Как вы вероятно заметили, мы дополнительно удалили пунктуацию с помощью метода `.isalpha()`.

Шаг 4. Лемматизация

Кроме того, с точки зрения анализа содержания слова «музеи» и «музей» означают одно и то же, но компьютер посчитает их разными словами. Значит, слова нужно привести к начальной (словарной) форме или лемме.

```
1    # импортируем класс для лемматизации
2    from nltk.stem import WordNetLemmatizer
```

```
3 |
4 | # импортируем словарь
5 | nltk.download('wordnet')
6 |
7 | # создаём объект этого класса
8 | lemmatizer = WordNetLemmatizer()
9 |
10 | # и пустой список для слов после лемматизации
11 | lemmatized = []
12 |
13 | # проходимся по всем токенам
14 | for token in no_stops:
15 |
16 |     # применяем лемматизацию
17 |     token = lemmatizer.lemmatize(token)
18 |
19 |     # добавляем слово после лемматизации в список
20 |     lemmatized.append(token)
21 |
22 | print(lemmatized)
```

```
1 | ['paris', 'visited', 'lot', 'museum', 'first', 'went', 'louvre', 'largest', 'art', 'mus
```

WordNet Lemmatizer библиотеки NLTK использует базу данных [WordNet](#) для поиска словарной формы слова.

Шаг 5. Стемминг

Стемминг (stemming), в отличие от лемматизации, ориентирован на поиск основы слова (stem). Попробуем применить и его.

Для начала используем стеммер (т.е. инструмент стемминга) Портера. Он достаточно консервативен, т.е. не так активно обрезает слова в поисках основы.

```
1 | # импортируем класс стеммера Porter и создаём объект этого класса
2 | from nltk.stem import PorterStemmer
3 | porter = PorterStemmer()
4 |
5 | # используем list comprehension вместо цикла for для стемминга и создания нового списка
6 | # такая запись намного короче
7 | stemmed_p = [porter.stem(s) for s in lemmatized]
8 | print(stemmed_p)
```

```
1 | ['pari', 'visit', 'lot', 'museum', 'first', 'went', 'louv', 'largest', 'art', 'museum'
```

Как вы видите, list comprehension позволяет в одну строчку создавать список. Подробнее этот прием мы рассмотрим на [следующем курсе](#).

Теперь применим более агрессивный стеммер Ланкастера.

```
1 | # аналогично импортируем класс Lancaster и создаём объект этого класса
2 | from nltk.stem import LancasterStemmer
3 | lancaster = LancasterStemmer()
4 |
5 | # также используем list_comprehension
6 | stemmed_l = [lancaster.stem(s) for s in lemmatized]
7 | print(stemmed_l)
```

```
1 | ['par', 'visit', 'lot', 'muse', 'first', 'went', 'louv', 'largest', 'art', 'muse', 'wc
```

К сожалению, ни тот, ни другой не показали хороших результатов. Стемминг к тексту мы применять не будем.

Мешок слов

Принцип метода **мешка слов** (Bag of Words, BoW) чрезвычайно прост. Мы считаем, как часто встречается каждое слово в тексте.

Несмотря на простоту, при правильной предобработке текста (в первую очередь удалении стоп-слов, которые и будут наиболее частотными) этот метод показывает неплохие результаты. Применим его к тексту с помощью класса **Counter** модуля **Collections**.

```
1 # из модуля collections импортируем класс Counter
2 from collections import Counter
3
4 # применяем класс Counter к словам после лемматизации
5 # на выходе возвращается словарь { слово : его частота в тексте }
6 bow_counter = Counter(lemmatized)
7 # print(bow_counter)
8
9 # функция most_common() упорядочивает словарь по значению
10 # посмотрим на первые 10 наиболее частотных слов
11 print(bow_counter.most_common(10))
```

```
1 [('museum', 3), ('art', 2), ('paris', 1), ('visited', 1), ('lot', 1), ('first', 1), ('v
```

Этот же метод можно реализовать с помощью класса **CountVectorizer** библиотеки **Scikit-learn**.

```
1 # импортируем класс CountVectorizer из библиотеки Scikit-learn
2 from sklearn.feature_extraction.text import CountVectorizer
3
4 # создаём объект этого класса и
5 # указываем, что хотим перевести слова в нижний регистр, а также
6 # отфильтровать стоп-слова через stop_words = 'english'
7 vectorizer = CountVectorizer(analyzer = "word",
8                               lowercase = True,
9                               tokenizer = None,
10                              preprocessor = None,
11                              stop_words = 'english',
12                              max_features = 5000)
```

```
1 # применяем этот объект к предложениям (ещё говорят документам)
2 bow_cv = vectorizer.fit_transform(sentences)
3
4 # на выходе получается матрица csr
5 print(type(bow_cv))
```

```
1 <class 'scipy.sparse.csr.csr_matrix'>
```

Преобразуем матрицу csr в привычный формат массива Numpy.

```
1 # для этого можно использовать .toarray()
2 print(bow_cv.toarray())
```

```
1 [[0 0 0 0 0 1 0 0 1 1 0 1 0 0 0]
2  [1 0 0 0 1 0 1 1 0 0 0 0 0 1 1]
3  [1 0 1 1 0 0 0 0 0 0 1 0 0 0 0]
4  [0 1 0 0 0 0 0 1 0 0 0 0 1 0 0]]
```

Строки представляют собой предложения (документы), столбцы — слова (токены).

```
1 bow_cv.shape
```

```
1 (4, 15)
```

Есть два способа посмотреть на используемые токены. С помощью атрибута `vocabulary_`.

```
1 vocab = vectorizer.vocabulary_  
2 print(vocab)
```

```
1 {'paris': 9, 'visited': 11, 'lot': 5, 'museums': 8, 'went': 13, 'louvre': 6, 'largest':
```

Здесь числа это не частотность слов, а просто их порядковый номер или индекс. Также токены можно вывести с помощью метода `.get_feature_names_out()`.

```
1 tokens = vectorizer.get_feature_names_out()
```

```
1 ['art' 'enourmous' 'hours' 'interested' 'largest' 'lot' 'louvre' 'museum'  
2 'museums' 'paris' 'spent' 'visited' 'week' 'went' 'world']
```

Результат для удобства также можно преобразовать в датафрейм.

```
1 # вначале создадим индекс предложений (документов)  
2 index_list = []  
3  
4 # в цикле пройдемся по элементам матрицы, обозначим их через '_'  
5 # функция enumerate задаст каждому элементу индекс, начиная с 0  
6 for i, _ in enumerate(bow_cv):  
7  
8     # прибавим индекс к слову Sentence  
9     index_list.append(f'Sentence_{i}')  
10  
11 # теперь можно использовать pd.DataFrame()  
12 bow_cv_df = pd.DataFrame(data = bow_cv.toarray(),  
13                           index = index_list,  
14                           columns = tokens)  
15 bow_cv_df
```

	art	enourmous	hours	interested	largest	lot	louvre	museum	museums	paris	spent	visited	week	went	world
Sentence_0	0	0	0	0	0	1	0	0	1	1	0	1	0	0	0
Sentence_1	1	0	0	0	1	0	1	1	0	0	0	0	0	1	1
Sentence_2	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0
Sentence_3	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0

Итог работы `CountVectorizer` может показаться менее наглядным, чем метод мешка слов (и кроме того у него нет встроенной возможности лемматизации и стемминга), однако этот класс позволяет выразить слова с помощью **числовых векторов**.

Это очень важный результат, о котором мы поговорим в конце занятия.

Метод TF-IDF

Чуть более сложный и продвинутый метод определения значимости слов в тексте называется TF-IDF (term frequency — inverse document frequency).

Основная идея

Если слово часто встречается во всех документах (это в первую очередь касается предлогов,


союзов и других стоп-слов), то вряд ли эти слова имеют большое значение. И наоборот, если слово встречается только в одном документе, вероятно оно в большей степени определяет его содержание.

Другими словами, определяется не только значимость слова в тексте, но и значимость слова с учётом всех текстов.

Простой пример и формула

Предположим, что у нас есть два текста, и мы посчитали частотность слов в каждом из них (т.е. создали мешок слов).

Документ 1		Документ 2	
Токен	Частотность	Токен	Частотность
this	1	this	1
is	1	is	1
a	2	another	2
sample	1	example	3

Пример взят из [Википедии](#) 

Расчет tf-idf для слова this

Поставим задачу рассчитать TF-IDF для слова *this* в каждом из документов. На первом этапе вычисляем **частоту слова** (term frequency или TF) относительно всех слов в документе.

$$tf(this, d_1) = \frac{\text{частотность слова}}{\text{всего слов}} = \frac{1}{5} = 0.2$$

$$tf(this, d_2) = \frac{\text{частотность слова}}{\text{всего слов}} = \frac{1}{7} \approx 0.14$$

Такой подход гарантирует, что мы учитываем, как часто встречается слово относительно длины документа.

На втором этапе рассчитаем **IDF** слова *this*. То есть мы делим общее количество документов в корпусе на количество документов, в которых встречается искомый токен, и берем логарифм (в данном случае, десятичный) частного.

$$idf(this, D) = \log_{10} \left(\frac{\text{всего документов}}{\text{документов с токеном}} \right) = \log_{10} \left(\frac{2}{2} \right) = 0$$

Остаётся перемножить TF и IDF.

$$tf - idf(this, d_1, D) = 0.2 \times 0 = 0$$

$$tf - idf(this, d_2, D) = 0.14 \times 0 = 0$$

Этот показатель равен нулю, что отражает низкую значимость слова *this* для обоих документов. Теперь сделаем аналогичный расчет для слова *example*.

Расчет tf-idf для слова *example*

$$tf(example, d_1) = \frac{0}{5} = 0$$

$$tf(example, d_2) = \frac{3}{7} \approx 0.429$$

$$idf(example, D) = \log_{10} \left(\frac{2}{1} \right) \approx 0.301$$

$$tf - idf(example, d_1, D) = 0 \times 0.301 = 0$$

$$tf - idf(example, d_2, D) = 0.429 \times 0.301 \approx 0.129$$

В данном случае слово *example* имеет заметно большее значение для второго документа. Для первого документа его значимость по-прежнему равна нулю (такого слова там нет).

Логика формулы следующая, чем выше частота слова в документе (tf) и чем реже оно встречается в целом в документах (idf), тем выше общий показатель (tf-idf).

Также замечу, что на практике эти формулы часто модифицируют. Например, к знаменателю формулы расчета idf добавляют единицу, чтобы избежать деления на ноль, если документов с таким токеном не найдется.

$$idf = \log \left(\frac{\text{всего документов}}{\text{документов с токеном} + 1} \right)$$

TF-IDF с помощью библиотеки Scikit-learn

Применим этот метод к исходному тексту про Париж и музеи. Вначале последовательно используем классы `CountVectorizer` и `TfidfTransformer`.

Способ 1. `CountVectorizer` + `TfidfTransformer`

TF или **частоту слов** можно взять из предыдущего раздела.

```
1 | bow_cv
```

```
1 | <4x15 sparse matrix of type '<class 'numpy.int64''>'
2 |   with 17 stored elements in Compressed Sparse Row format>
```

Теперь нужно рассчитать **IDF**.

```
1 | # импортируем TfidfTransformer (CountVectorizer уже импортирован)
2 | from sklearn.feature_extraction.text import TfidfTransformer
3 |
4 | # создадим объект класса TfidfTransformer
5 | tfidf_trans = TfidfTransformer(smooth_idf = True, use_idf = True)
```



```

6 |
7 | # и рассчитаем IDF слов
8 | tfidf_trans.fit(bow_cv)
9 |
10 | # поместим результат в датафрейм
11 | df_idf = pd.DataFrame(tfidf_trans.idf_, index = tokens, columns = ["idf_weights"])

```

Остается **TF x IDF**.

```

1 | # рассчитаем TF-IDF (по сути умножаем TF на IDF)
2 | tf_idf_vector = tfidf_trans.transform(bow_cv)
3 | tf_idf_vector

```

```

1 | <4x15 sparse matrix of type '<class 'numpy.float64'>'
2 |     with 17 stored elements in Compressed Sparse Row format>

```

Теперь мы можем посмотреть на показатель TF-IDF для конкретного слова в конкретном документе.

```

1 | # для этого переведем матрицу csr в обычный массив Numpy
2 | df_tfidf = pd.DataFrame(tf_idf_vector.toarray(), columns = vectorizer.get_feature_names())
3 |
4 | # и транспонируем его (запишем столбцы в виде строк)
5 | print(df_tfidf.T)

```

	0	1	2	3
art	0.0	0.344315	0.414289	0.000000
enourmous	0.0	0.000000	0.000000	0.617614
hours	0.0	0.000000	0.525473	0.000000
interested	0.0	0.000000	0.525473	0.000000
largest	0.0	0.436719	0.000000	0.000000
lot	0.5	0.000000	0.000000	0.000000
louvre	0.0	0.436719	0.000000	0.000000
museum	0.0	0.344315	0.000000	0.486934
museums	0.5	0.000000	0.000000	0.000000
paris	0.5	0.000000	0.000000	0.000000
spent	0.0	0.000000	0.525473	0.000000
visited	0.5	0.000000	0.000000	0.000000
week	0.0	0.000000	0.000000	0.617614
went	0.0	0.436719	0.000000	0.000000
world	0.0	0.436719	0.000000	0.000000

Всего после обработки метод оставил 15 слов.

```
1 | df_tfidf.T.shape
```

```
1 | (15, 4)
```

Такого же результата можно добиться, применив метод `TfidfVectorizer`.

Способ 2. `TfidfVectorizer`

```

1 | # импортируем класс TfidfVectorizer
2 | from sklearn.feature_extraction.text import TfidfVectorizer
3 |
4 | # создаем объект класса TfidfVectorizer
5 | tfidf_vectorizer = TfidfVectorizer(use_idf = True, stop_words= 'english')
6 |
7 | # сразу рассчитываем TF-IDF слов
8 | tfidf = tfidf_vectorizer.fit_transform(sentences)

```

Мы можем посмотреть, какие слова остались после фильтрации.

```
1 | print(tfidf_vectorizer.get_feature_names_out())
```

```
1 ['art' 'enourmous' 'hours' 'interested' 'largest' 'lot' 'louvre' 'museum'
2  'museums' 'paris' 'spent' 'visited' 'week' 'went' 'world']
```

В частности мы видим, что метод `TfidfVectorizer` оставил те же слова, что и `CountVectorizer` и `TfidfTransformer`. Посмотрим IDF слов.

```
1 tfidfvectorizer.idf_
```

```
1 array([1.51082562, 1.91629073, 1.91629073, 1.91629073, 1.91629073,
2        1.91629073, 1.91629073, 1.51082562, 1.91629073, 1.91629073,
3        1.91629073, 1.91629073, 1.91629073, 1.91629073, 1.91629073])
```

Посмотрим на количество документов и количество токенов (слов).

```
1 tfidf.shape
```

```
1 (4, 15)
```

Рассчитаем значение **TF-IDF** для каждого слова по каждому тексту.

```
1 # посмотрим на значение TF-IDF для конкретного слова в конкретном документе
2 # чем оно уникальнее для конкретного документа, тем выше показатель
3 df_tfidf = pd.DataFrame(tfidf.toarray(), columns = tfidfvectorizer.get_feature_names_out())
4 print(df_tfidf.T)
```

	0	1	2	3
art	0.0	0.344315	0.414289	0.000000
enourmous	0.0	0.000000	0.000000	0.617614
hours	0.0	0.000000	0.525473	0.000000
interested	0.0	0.000000	0.525473	0.000000
largest	0.0	0.436719	0.000000	0.000000
lot	0.5	0.000000	0.000000	0.000000
louvre	0.0	0.436719	0.000000	0.000000
museum	0.0	0.344315	0.000000	0.486934
museums	0.5	0.000000	0.000000	0.000000
paris	0.5	0.000000	0.000000	0.000000
spent	0.0	0.000000	0.525473	0.000000
visited	0.5	0.000000	0.000000	0.000000
week	0.0	0.000000	0.000000	0.617614
went	0.0	0.436719	0.000000	0.000000
world	0.0	0.436719	0.000000	0.000000

В целом упражнение можно завершить. Однако напомним, что нашей целью было определить тему текста. Для этого мы также можем рассчитать **среднее значение TF-IDF для каждого слова по всем текстам**.

Вычислим среднее арифметическое по строкам матрицы, приведенной выше (подробное объяснение кода вы найдете в [блокноте](#)).

```
1 mean_weights = np.asarray(tfidf.mean(axis = 0)).ravel().tolist()
2 mean_weights
```

```
1 [0.18965081782108964,
2  0.15440359274390048,
3  0.13136818731601646,
4  0.13136818731601646,
5  0.10917982746877804,
6  0.125,
7  0.10917982746877804,
8  0.2078121960479979,
9  0.125,
10 0.125,
11 0.13136818731601646,
12 0.125,
13 0.15440359274390048,
```

```
14 | 0.10917982746877804,  
15 | 0.10917982746877804]
```

И создадим датафрейм, отсортировав слова по убыванию средних весов.

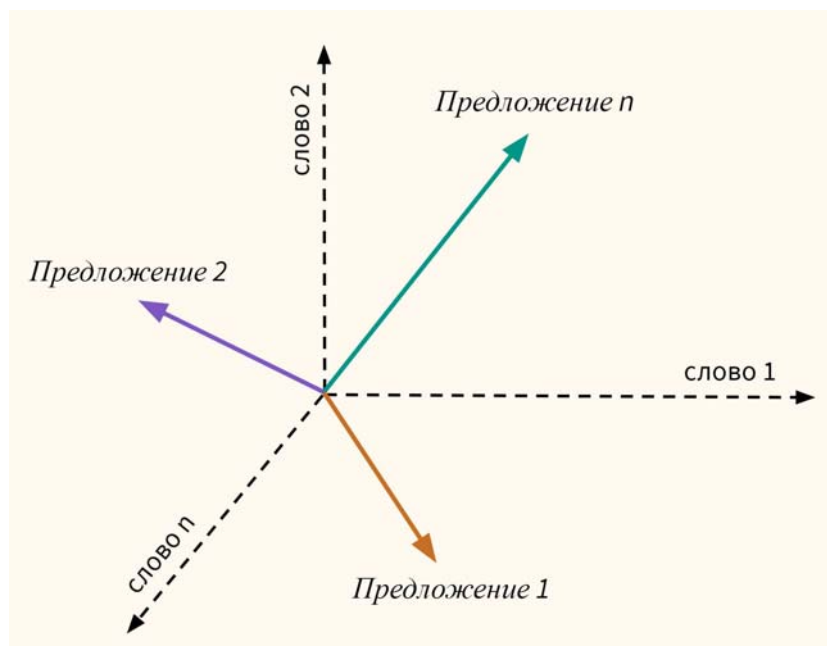
```
1 | # создаём датафрейм из словаря  
2 | mean_weights_df = pd.DataFrame({'term': tfIdfVectorizer.get_feature_names_out(), 'mean_  
3 |  
4 | # сортируем по убыванию 10 слов с максимальным средним TF-IDF  
5 | mean_weights_df.sort_values(by = 'mean_weights', ascending = False).reset_index(drop =
```

	term	mean_weights
0	museum	0.207812
1	art	0.189651
2	enourmous	0.154404
3	week	0.154404
4	hours	0.131368
5	interested	0.131368
6	spent	0.131368
7	lot	0.125000
8	museums	0.125000
9	paris	0.125000

Результаты

Как мы видим, ни один из алгоритмов не справился со своей задачей на отлично. Отчасти это связано с ограничениями самих методов, отчасти с тем, что мы взяли очень небольшой текст для анализа.

При этом несмотря на неидеальный результат, мы добились, как уже было сказано, **векторизации текста**, то есть описания его с помощью чисел.



Дополнительные примеры

Прежде чем завершить, приведем два примера применения текстовых векторов.

Косинусное расстояние между текстовыми векторами

Возьмем два предложения и объединим их в корпус.

```
1 text1 = 'all the world's a stage, and all the men and women merely players'
2 text2 = 'you must be the change you wish to see in the world'
3
4 corpus = [text1, text2]
```

Создадим объект класса `TfidfVectorizer` и рассчитаем веса `tf-idf`.

```
1 # создадим объект класса TfidfVectorizer
2 tfidf_vectorizer = TfidfVectorizer(use_idf = True, stop_words = 'english')
3
4 # на выходе получаем два вектора, где каждое значение - это вес (показатель tf-idf) слов
5 X = tfidf_vectorizer.fit_transform(corpus)
6
7 # преобразуем данные в массив Numpy
8 print(X.toarray())
```

```
1 [[0.         0.4261596  0.4261596  0.4261596  0.4261596  0.
2      0.4261596  0.30321606]
3  [0.6316672  0.         0.         0.         0.         0.6316672
4      0.         0.44943642]]
```

Для удобства, мы можем преобразовать данные в датафрейм.

```
1 # данными станут веса ti-idf, индексом - список векторов, столбцами - токены
2 vectors_df = pd.DataFrame(data = X.toarray(),
3                           index = ['vector1', 'vector2'],
4                           columns = tfidf_vectorizer.get_feature_names_out())
5 vectors_df
```

	change	men	merely	players	stage	wish	women	world
vector1	0.000000	0.42616	0.42616	0.42616	0.42616	0.000000	0.42616	0.303216
vector2	0.631667	0.00000	0.00000	0.00000	0.00000	0.631667	0.00000	0.449436

Вектора готовы. Напомню формулу расчета косинусного расстояния.

$$\cos \theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

Теперь поместим каждый вектор в отдельную переменную.

```
1 vector1 = X.toarray()[0]
2 vector2 = X.toarray()[1]
```

Выполним операции в числителе формулы.

```
1 # рассчитаем скалярное произведение векторов
2 numerator = np.dot(vector1, vector2)
```

Займемся знаменателем.

```
1 # (1) рассчитаем длины (по большому счету, это теорема Пифагора)
2 vector1Len = np.linalg.norm(vector1)
3 vector2Len = np.linalg.norm(vector2)
4
5 # (2) перемножим их
6 denominator = vector1Len * vector2Len
```

Остается рассчитать косинус угла.

```
1 # посмотрим, чему равен косинус угла между векторами
2 cosine = numerator/denominator
3 cosine
```

```
1 0.13627634143908643
```

И после этого перевести его в радианы, а затем в градусы.

```
1 # найдем угол в градусах по его косинусу
2 # для этого вначале вычислим угол в радианах
3 angle_radians = np.arccos(cosine)
4
5 # затем в градусах
6 angle_degrees = angle_radians * 360/2/np.pi
7 round(angle_degrees, 2)
```

```
1 82.17
```

Кластерный анализ текста

Теперь давайте посмотрим, как можно разделить предложения в тексте на темы (кластеры).

Возьмем текст с предложениями из Википедии, посвященными науке о данных и Большому театру.

```
1 text = '''
2 Data science is an interdisciplinary field that uses scientific methods, processes, al
3 It applies knowledge and actionable insights from data across a broad range of applica
4 Data science is related to data mining, machine learning and big data.
5 The Bolshoi Theatre is a historic theatre in Moscow, Russia.
6 It was originally designed by architect Joseph Bové, which holds ballet and opera perf
7 Before the October Revolution it was a part of the Imperial Theatres of the Russian Em
8 Data science is a concept to unify statistics, data analysis, informatics, and their r
9 However, data science is different from computer science and information science.
10 The main building of the theatre, rebuilt and renovated several times during its histo
11 On 28 October 2011, the Bolshoi re-opened after an extensive six-year renovation.
12 '''
```

Разобьем текст на предложения и переведем в нижний регистр.

```
1 # создадим список из предложений
2 corpus = []
3
4 # для этого в цикле for пройдемся по тексту, разделив его по символу новой строки \n
5 for line in text.split('\n'):
6
7     # если строка не пустая (т.е. True)
8     if line:
9
10         # переводим ее в нижний регистр
11         line = line.lower()
12         # и добавляем в список
13         corpus.append(line)
14
15 corpus
```

```
1 ['data science is an interdisciplinary field that uses scientific methods, processes,  
2 'it applies knowledge and actionable insights from data across a broad range of appli  
3 'data science is related to data mining, machine learning and big data.',  
4 'the bolshoi theatre is a historic theatre in moscow, russia.',  
5 'it was originally designed by architect joseph bové, which holds ballet and opera pe  
6 'before the october revolution it was a part of the imperial theatres of the russian  
7 'data science is a concept to unify statistics, data analysis, informatics, and their  
8 'however, data science is different from computer science and information science.',  
9 'the main building of the theatre, rebuilt and renovated several times during its his  
10 'on 28 october 2011, the bolshoi re-opened after an extensive six-year renovation.']
```

Создадим векторы каждого из предложений.

```
1 # применим TfidfVectorizer  
2 tfidfVectorizer = TfidfVectorizer(use_idf = True, stop_words= 'english')  
3  
4 # на выходе получаем векторы предложений  
5 X = tfidfVectorizer.fit_transform(corpus)
```

Применим алгоритм k-средних и разделим предложения на кластеры.

```
1 # импортируем алгоритм k-средних из библиотеки sklearn  
2 from sklearn.cluster import KMeans  
3  
4 # так как мы знаем, что темы две, используем гиперпараметр k = 2  
5 kmeans = KMeans(n_clusters = 2, n_init = 10, random_state = 42).fit(X)
```

Как результат, мы создали две модели:

- модель векторизации через tfidfVectorizer;
- модель кластеризации.

Возьмем новые предложения и с помощью обученных моделей разобьем их на кластеры.

```
1 # первое предложение из области Data Science, два других - про Большой театр  
2 prediction = ['Many statisticians, including Nate Silver, have argued that data science  
3 'Urusov set up the theatre in collaboration with English tightrope walker  
4 'Until the mid-1990s, most foreign operas were sung in Russian, but Itali  
5  
6 # применим две модели, сначала создадим векторы новых предложений (tfidfVectorizer.trans  
7 # затем отнесем их к одному из кластеров (kmeans.predict)  
8 kmeans.predict(tfidfVectorizer.transform(prediction))  
  
1 array([0, 1, 1], dtype=int32)
```

Как мы видим, первое предложение отнесено к одному кластеру, второе и третье — к другому.

Подведем итог

Сегодня мы впервые поработали с текстами. В частности, научились предварительно обрабатывать их и анализировать содержание. Для этого мы использовали метод мешка слов и метод TF-IDF. Второй метод позволил превратить предложения в числа или векторизовать их.

Благодаря векторизации предложений, мы смогли рассчитать косинусное сходство между двумя документами и провести кластерный анализ.

Ответы на вопросы

Вопрос. Имеет ли значение какое основание логарифма использовать в формуле TF-IDF?

Ответ. Нет, не имеет, основание может быть любым. В формуле выше используется десятичный логарифм, sklearn использует натуральный.

Вопрос. Во втором дополнительном примере, откуда мы знаем, что кластеров должно быть два (в алгоритме k-means)?

Ответ. В данном случае, мы либо заранее знаем, что темы две, и нужно научить алгоритм разделять тексты на эти два кластера, либо попробовать метод локтя, как мы это делали на занятии по кластеризации.

Вопрос. Зачем мы применили `.ravel()` к массиву при работе с `TfidfVectorizer` (способ 2)? То есть зачем убрали второе измерение?

Ответ. Мы это сделали, чтобы затем корректно сработала функция `tolist()`. Если не использовать `.ravel()`, то применив только `tolist()` мы получим вложенные списки `[[some_list]]`, а нам нужен обычный список `[some_list]`.

Попробуйте в качестве эксперимента обойтись без `.ravel()`.
