

Применение методов машинного обучения в области обработки естественного языка в облачной среде Google Colaboratory

Эрдман Александр Алексеевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье рассмотрены метод обработки естественного языка и способ анализа содержания текста. Реализация данных технологий осуществляется с помощью языка программирования Python и библиотеки NLP. Результатом исследования является применение технологий на практическом примере.

Ключевые слова: Python, NLP, машинное обучение

Application of machine learning methods in the field of natural language processing in the Google cloud environment Colaboratory

Erdman Alexander Alekseevich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article discusses the method of natural language processing and the method of text content analysis. The implementation of these technologies is carried out using the Python programming language and the NLP library. The result of the research is the application of technology on a practical example.

Keywords: Python, NLP, machine learning

1 Введение

1.1 Актуальность

Актуальность темы "Обработка естественного языка" (Natural Language Processing, NLP) в настоящее время крайне высока. Это объясняется тем, что человеческий язык является наиболее естественным и удобным способом коммуникации для человека, и, следовательно, большая часть информации в мире представлена именно в виде текстовых данных. В связи с этим, возникает необходимость в автоматизированном анализе и обработке текстовых данных. Это позволяет решать различные задачи, такие как поиск информации, машинный перевод, анализ мнений, выделение ключевых слов и фраз, автоматическое составление сводки новостей и многие другие.

Таким образом, актуальность темы "обработка естественного языка" объясняется необходимостью в автоматизированном анализе и обработке текстовых данных, взрывным ростом объема информации в Интернете, а также сложностью и многозначностью текстовых данных, которые делают их

перспективной областью применения машинного обучения и глубокого обучения.

1.2 Обзор исследований

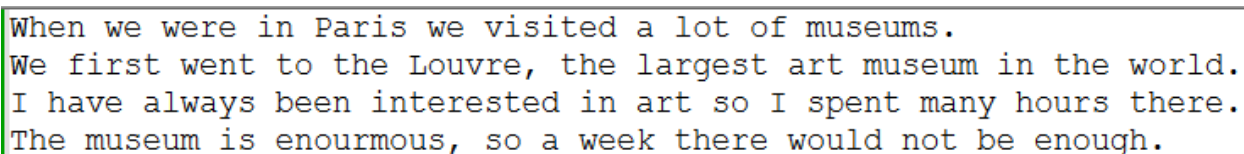
В.А. Негодин в своём труде рассмотрел различные библиотеки, позволяющие работать с машинным обучением на языке программирования Python [1]. А.В. Давыдов, А.К. Жусупова и О.С. Салыкова в своей статье проанализировали и сравнили между собой различные языки программирования в машинном обучении [2]. А.В. Киримова описала преимущества пакетов языка программирования Python для работы с машинным и глубоким обучением [3]. Р.К. Акжолов и А.В. Верига рассмотрели методы предварительной обработки текста для дальнейшей работы с алгоритмами машинного обучения [4]. Р.И. Каримов в своей работе описал лингвистический метод обработки текста, который состоит из четырех этапов: графематический анализ, морфологический анализ, синтаксический анализ и семантический анализ [5].

2 Материалы и методы

В качестве среды разработки выступает система Google Colaboratory, которая предоставляет множество библиотек, связанных с машинным обучением. В качестве языка программирования используется Python.

3 Результаты и обсуждения

Распространённой задачей данной темы является определение темы текста. Если человек с данной задачей справляется довольно быстро и правильно, то у компьютера возникают трудности. В качестве корпуса (совокупность рассматриваемых текстов) берётся следующий текст (рис. 1).



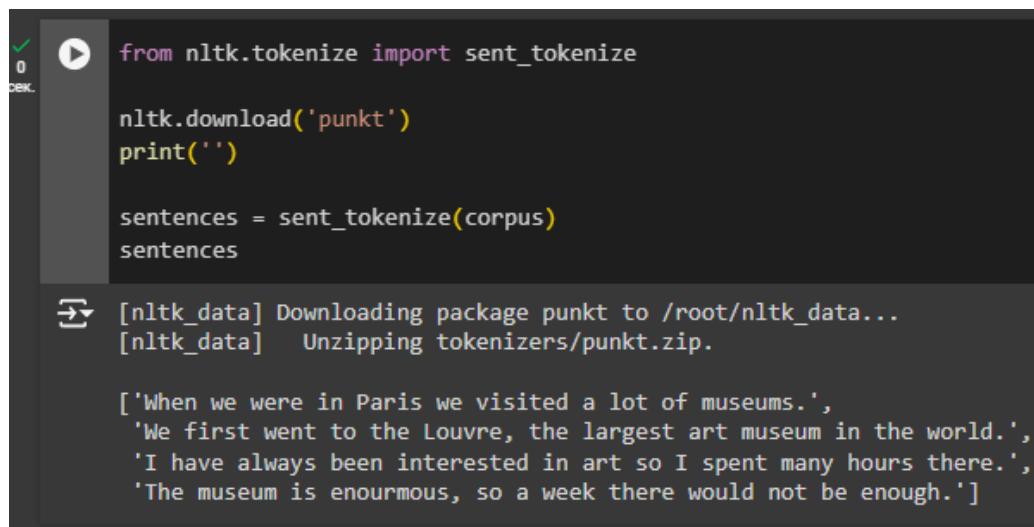
```
When we were in Paris we visited a lot of museums.  
We first went to the Louvre, the largest art museum in the world.  
I have always been interested in art so I spent many hours there.  
The museum is enormous, so a week there would not be enough.
```

Рисунок 1. Корпус

Данный корпус помещается в одноимённую переменную. После определения с текстом, происходит импорт необходимых библиотек для работы, в частности NLP – главного инструмента для работы обработки естественного языка, а также Pandas и Numpy. Pandas предоставляет структуры данных и функции для обработки табличных данных, подобных данным в Excel или базах данных SQL. Numpy, в свою очередь, предоставляет структуры данных и функции для работы с многомерными массивами и матрицами.

Перед непосредственным анализом следует совершить предварительную обработку данных. Для решения поставленной задачи используются стандартный метод библиотеки NLTK `sent_tokenize`. Данная

функция позволяет разбивать текст на предложения (рис. 2). Данный метод использует обученную модель для иностранного языка.



```
from nltk.tokenize import sent_tokenize

nltk.download('punkt')
print('')

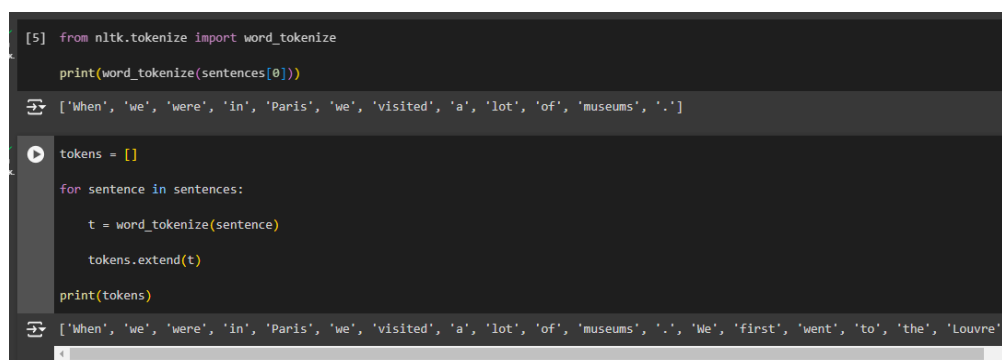
sentences = sent_tokenize(corpus)
sentences

[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.

['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 enormous, so a week there would not be enough.']
```

Рисунок 2. Применение функции `sent_tokenize` для разбиения корпуса на предложения

Следующим шагом является токенизация слов (рис. 3). Токенизация слов - это процесс разбиения текста на отдельные слова или токены. Этот процесс является важной частью обработки текстов в машинном обучении NLP, поскольку алгоритмы машинного обучения работают с численными данными, а не с текстом. Токенизация, в конкретной задачи нужна для дальнейшего удаления стоп-слов - позволяет удалить стоп-слова, которые являются общими словами, такими как "the", "is", "and", которые не несут существенной информации для задачи. Это может уменьшить размерность данных и улучшить производительность модели. А также выполняет функцию нормализации текста - приведение всех слов к нижнему регистру, удаление знаков препинания и т.д. Это может помочь уменьшить разнообразие слов в данных и улучшить производительность модели.



```
[5] from nltk.tokenize import word_tokenize

print(word_tokenize(sentences[0]))

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

tokens = []

for sentence in sentences:
    t = word_tokenize(sentence)
    tokens.extend(t)

print(tokens)

['When', 'we', 'were', 'in', 'Paris', 'we', 'visited', 'a', 'lot', 'of', 'museums', '.', 'We', 'first', 'went', 'to', 'the', 'Louvre',
```

Рисунок 3. Токенизация слов всех предложений

После токенизации следует ряд функция для перевода слов в нижний регистр, удаления стоп-слов и знаков препинания (рис. 4).

```
[6] from nltk.corpus import stopwords
nltk.download('stopwords')

unique_stops = set(stopwords.words('english'))
no_stops = []

for token in tokens:
    token = token.lower()

    if token not in unique_stops and token.isalpha():
        no_stops.append(token)

print(no_stops)
```

```
[1] ['paris', 'visited', 'lot', 'museums', 'first', 'went', 'louvre', 'largest', 'art', 'museum', 'world', 'always', 'interested', 'art', 'spent', 'many', 'hours', 'museum', 'enormous', 'week', 'would', 'enough']
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data] Whipping corpus/stopwords.zip
```

Рисунок 4. Преобразование в нижний регистр, удаление стоп-слов и знаков пунктуации

В данном блоке кода происходит импорт модуля «stopwords». Следующим шагом создаётся множество «unique_stops», который содержит все стоп-слова используемого языка. Пустой список «no_stops» в дальнейшем будет содержать в себе токены без стоп-слов. С помощью цикла for осуществляется перебор слов из списка «tokens». Внутри цикла происходит переход каждого слова в нижний регистр с помощью метода «lower()», после чего происходит проверка преобразованного токена на наличие стоп-слова и одновременно осуществляется удаление любых токенов, которые являются знаком препинания или цифрой. Цикл завершается добавлением токена в список «no_stops». На выходе метода получается список слов, переведённый в нижний регистр без стоп-слов и знаков препинания.

Следующим шагом обработки слов является лемматизация. Данный процесс осуществляет преобразования слова в его основную форму, называемую леммой. Лемма - это базовая форма слова, которая используется в словаре. Например, леммой слова "running" является "run", а леммой слова "cars" является "car". Лемматизация является важной частью обработки текстов в NLP (Natural Language Processing), поскольку она позволяет уменьшить разнообразие слов в тексте и улучшить качество анализа текста. С помощью данного процесса можно добиться уменьшения размерности данных, улучшения качества анализа текста, улучшение визуализации данных.

Лемматизация списка токенов осуществляется с помощью библиотеки NLTK и её компонента WordNet (рис. 5).

```
[51] from nltk.stem import WordNetLemmatizer
nltk.download('wordnet')

lemmatizer = WordNetLemmatizer()
lemmatized = []

for token in no_stops:
    token = lemmatizer.lemmatize(token)
    lemmatized.append(token)

print(lemmatized)
```

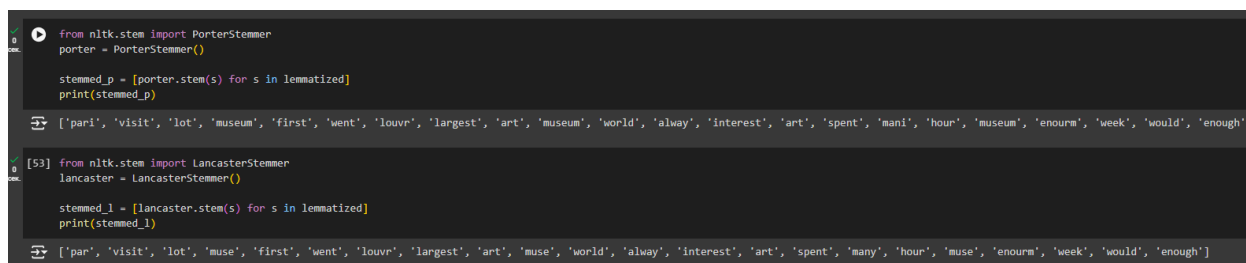
```
[1] ['paris', 'visited', 'lot', 'museum', 'first', 'went', 'louvre', 'largest', 'art', 'museum', 'world', 'always', 'interested', 'art', 'spent', 'many', 'hour', 'museum', 'enormous', 'week', 'would', 'enough']
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Package wordnet is already up-to-date!
```

Рисунок 5. Процесс лемматизации

После лемматизации следует процесс стемминг, который дополняет предыдущий шаг. Стемминг - это процесс уменьшения слов до их основного корня, называемого "стемом". Стем является частью слова, которая

сохраняется неизменной при склонении или спряжении. Например, стемом слова "running" является "run", а стемом слова "cars" является "car". Стемминг является полезной техникой в обработке текстов, поскольку он позволяет уменьшить разнообразие слов в тексте и улучшить качество анализа текста.

Для реализации этого процесса используются компоненты из вышеописанной библиотеки PorterStemmer и LancasterStemmer (рис. 6).

The image is a screenshot of a Jupyter Notebook interface. It contains two code blocks. The first block imports PorterStemmer from nltk.stem, creates a PorterStemmer object, and applies it to a list of lemmatized words. The second block imports LancasterStemmer from nltk.stem, creates a LancasterStemmer object, and applies it to the same list of lemmatized words. The output of both blocks shows the stemmed versions of the words.

```
from nltk.stem import PorterStemmer
porter = PorterStemmer()

stemmed_p = [porter.stem(s) for s in lemmatized]
print(stemmed_p)

['pari', 'visit', 'lot', 'museum', 'first', 'went', 'louvre', 'largest', 'art', 'museum', 'world', 'always', 'interest', 'art', 'spent', 'mani', 'hour', 'museum', 'enough', 'week', 'would', 'enough']

[53] from nltk.stem import LancasterStemmer
lancaster = LancasterStemmer()

stemmed_l = [lancaster.stem(s) for s in lemmatized]
print(stemmed_l)

['par', 'visit', 'lot', 'muse', 'first', 'went', 'louvre', 'largest', 'art', 'muse', 'world', 'always', 'interest', 'art', 'spent', 'many', 'hour', 'muse', 'enough', 'week', 'would', 'enough']
```

Рисунок 6. Стемминг слов с помощью методов PorterStemmer и LancasterStemmer

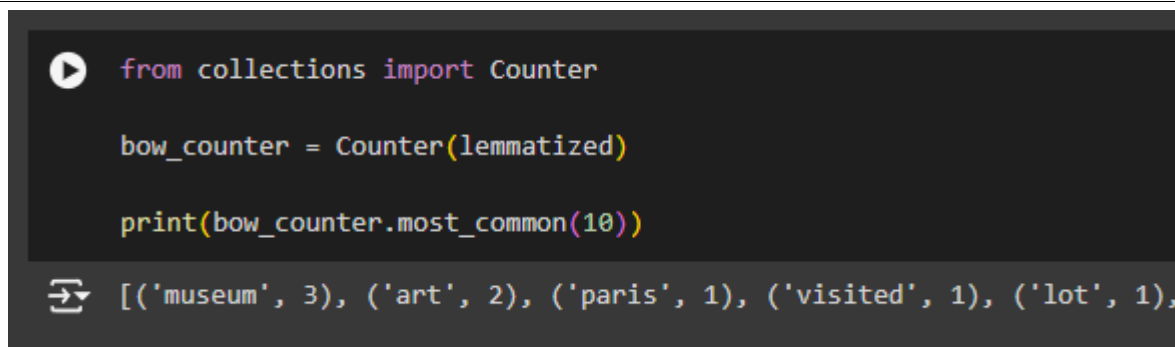
В первом блоке кода происходит создание экземпляра класса, который будет использоваться для стемминга слов. Списочное включение «stemmed_p» преобразует каждый лемматизированный токен (слово) из списка «lemmatized» в его стем с помощью метода «stem()» экземпляра класса «PorterStemmer». Список «stemmed_p» содержит все стеммингованные токены (слова) из списка «lemmatized». В завершении работы алгоритма на выходе получается список со словами, у которых удалены приставки и суффиксы слова.

Во втором блоке кода используется схожий алгоритм LancasterStemmer, который является обновлённой версией вышеописанного метода и позволяет не только преобразовывать слова в корень слова, но также восстанавливать удалённые суффиксы и приставки в предыдущем методе.

В обоих результатах алгоритмы сильно обрезают слова, что может привести к дальнейшим неточностям работы. По этой причине, данные, полученные после стемминга, использоваться не будут из-за возможности искажений. Вместо них будут использоваться лемматизированные данные.

Первичная обработка данных завершена и начинается непосредственно анализ содержания текста. Первым методом выступает «мешок слов» (bag of words). В методе "мешок слов" каждое уникальное слово в наборе текстовых данных рассматривается как отдельный элемент вектора, а частота появления слова в тексте соответствует значению элемента вектора. Таким образом, каждый текст представляется как вектор, содержащий частоты появления всех уникальных слов в наборе данных.

Для создания списка с ключом и значением, где ключом является слово, а значение – это количество повторений в тексте (рис. 7).



```
from collections import Counter

bow_counter = Counter(lemmatized)

print(bow_counter.most_common(10))
```

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

Рисунок 7. Анализ частоты появления слова в тексте

Для непосредственного преобразования текстовых данных в численные вектора используется метод `CountVectorizer` (рис. 8). После выполнения метода на выходе получается матрица формата `csc`. Для просмотра матрицы в привычном формате следует ещё одно преобразование с помощью метода `toarray`. В результате строки представляют из себя предложения, а столбцы – слова. В дальнейшем с помощью данного массива можно увидеть, какие токены используются. Для этого реализуется метод `«vocabulary»`. Далее осуществляется преобразование матрицы в `dataframe` (рис. 9). С помощью цикла осуществляется проход по элементам матрицы, обозначенные соответствующим именем. При этом применяется функция `«enumerate»`, которая возвращает индекс каждого элемента. С помощью метода `«DataFrame»` происходит непосредственное преобразование.

```

[54] from sklearn.feature_extraction.text import CountVectorizer

      vectorizer = CountVectorizer(analyzer = "word",
                                  lowercase = True,
                                  tokenizer = None,
                                  preprocessor = None,
                                  stop_words = 'english',
                                  max_features = 5000)

[55]
      bow_cv = vectorizer.fit_transform(sentences)

      print(type(bow_cv))

<class 'scipy.sparse._csr.csr_matrix'>

[56]
      print(bow_cv.toarray())

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

[14]
      bow_cv.shape

(4, 15)

```

Рисунок 8. Преобразование текста в вектора и вывод результат в виде матрицы

```

index_list = []

for i, _ in enumerate(bow_cv):
    index_list.append(f'Sentence_{i}')

bow_cv_df = pd.DataFrame(data = bow_cv.toarray(),
                        index = index_list,
                        columns = tokens)

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

Рисунок 9. Преобразование и вывод dataframe

Для более детального анализа далее применяется метод TF-IDF. Данный метод из себя представляет численную меру, которая используется для оценки важности каждого слова (термина) в наборе текстовых данных.

Для реализации метода необходимо импортировать библиотеку «sklearn», в которой находится нужный метод `TfidfTransformer`. Для расчёта IDF используется матрица, полученная на предыдущих этапах анализа.

В начале создаётся экземпляр класса `TfidfTransformer`, который будет использоваться для вычисления TF-IDF весов. Аргументами являются «smooth_idf = True», указывающий на использование сглаживания IDF для предотвращения деления на ноль, а также аргумент «use_idf = True», устанавливающий использование IDF для вычисления TF-IDF весов. Для непосредственного расчёта IDF слов используется метод «fit». В конце результат вычисления записывается в dataframe (рис. 10).

```
[19]
from sklearn.feature_extraction.text import TfidfTransformer

tfidf_trans = TfidfTransformer(smooth_idf = True, use_idf = True)

tfidf_trans.fit(bow_cv)

df_idf = pd.DataFrame(tfidf_trans.idf_, index = tokens, columns = ["idf_weights"])
```

Рисунок 10. Расчёт IDF слов.

Для получения меры TF-IDF, которая получается перемножением TF и IDF, которая является более точной оценкой важности каждого слова (термина) в наборе текстовых данных, чем просто частота появления слова (термина) в тексте (TF). Достигается результат за счёт метода «transform()».

После этого можно увидеть результат TF-IDF для конкретного слова в документе. Для этого создаётся dataframe для удобной визуализации (рис. 11).

```
[20]
tf_idf_vector = tfidf_trans.transform(bow_cv)
tf_idf_vector
```

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

```
[21]
df_tfidf = pd.DataFrame(tf_idf_vector.toarray(), columns = vectorizer.get_feature_names_out())

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

```
df_tfidf.T.shape
```

(15, 4)

Рисунок 11. Получение TF-IDF и визуализация результата

В результате исследования были рассмотрены методы для предварительной обработки текстовой информации, а также на практике применены методы для анализа текста – «мешок слов» и TF-IDF. Метод TF-IDF позволяет представлять текст в числа и векторизовать.

Библиографический список

1. Негодин В.А. Машинное обучение в языке программирования Python // Форум молодых ученых. 2019. № 8 (36). С. 201-203.
2. Давыдов А.В., Жусупова А.К., Салыкова О.С. Сравнение различных языков программирования, применяемых в машинном обучении // Вестник науки. 2023. Т. 3. № 2 (59). С. 155-165.
3. Киримова А.В. Библиотеки для машинного обучения языка программирования python // Modern Science. 2021. № 7. С. 293-296.
4. Акжолов Р.К., Верига А.В. Предобработка текста для решения задач NLP // Вестник науки. 2020. Т. 1. № 3 (24). С. 66-68.
5. Каримов Р.И. NLP - обработка естественного языка. лингвистический метод обработки текста // В сборнике: МАВЛЮТОВСКИЕ ЧТЕНИЯ. материалы XV Всероссийской молодежной научной конференции: в 7 томах . Уфа, 2021. С. 250-254.