Лекция 2.3.2. Векторизация текста. Метод TF-IDF, косинус сходства. Распознавание именованных сущностей.

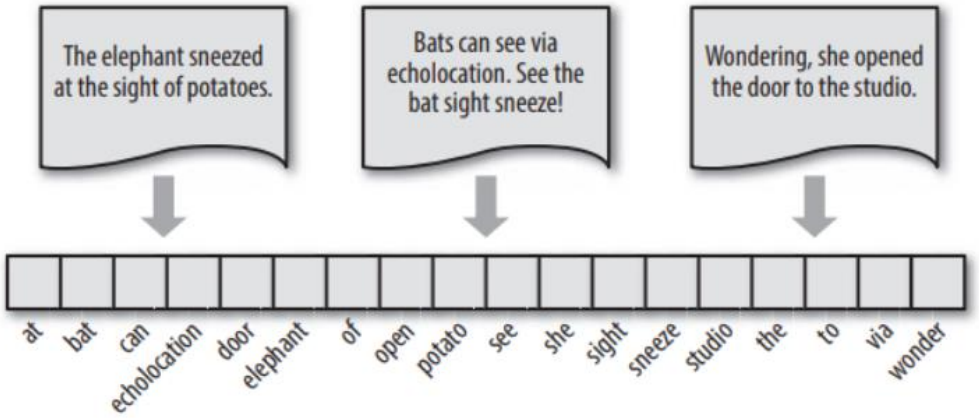
Как только текст превратился в очищенную нормализованную последовательность слов, запускается процесс их векторизации.

Векторизация — процесс преобразования нечисловых данных (например, текста, изображений и т. д.) в векторное представление, к которому можно применить методы машинного обучения.

Преобразование документов в численный вид дает возможность осуществлять их анализ и создавать экземпляры, с которыми смогут работать алгоритмы машинного обучения. В анализе текста экземплярами являются целые документы или высказывания, которые могут иметь самые разные размеры, от коротких цитат или твитов до целых книг. Но сами векторы всегда имеют одинаковую длину. Каждое свойство в векторном представлении — это признак. В случае с текстом признаки представляют атрибуты и свойства документов — включая их содержимое и метаатрибуты, такие как длина документа, имя автора, источник и дата публикации. Вместе все признаки документа описывают многомерное пространство признаков, к которому могут применяться методы машинного обучения.

Мешок слов (Bag-of-words)

Основным методом векторизации текста является мешок слов (bag-of-words), идея которого состоит в том, что смысл и сходство слов кодируются в виде словаря. Мешок слов – детальная репрезентативная модель для упрощения обработки текстового содержания. Она не учитывает грамматику или порядок слов и нужна, главным образом, для определения количества вхождений отдельных слов в анализируемый текст. На практике bag-of-words реализуется следующим образом: создается вектор длиной в словарь, для каждого слова считается количество вхождений в текст и это число подставляется на соответствующую позицию в векторе. Однако, при этом теряется порядок слов в тексте, а значит, после векторизации предложения, к примеру, “У меня нет кошки” и “Нет, у меня кошки” будут идентичны, но противоположны по смыслу.



Реализовать подобный метод векторизации на Python можно через библиотеку sklearn, которая предоставляет множество алгоритмов, необходимых для машинного обучения. Как раз в ней существует функция `CountVectorizer()`, автоматически формирующая мешок слов:

```
#Подключение библиотек
import nltk
import string
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

nltk.download('punkt') #Загрузка расширения NLTK (Выполнить один
 раз и закоментировать!)

text = 'Погода класс! Погода супер!' #Пример текст

remove_punct_dict = dict((ord(punct), None) for punct in string.
punctuation) #Убираем знаки пунктуации из предложения
word_token = nltk.word_tokenize(text.lower().translate(remove_pu
nct_dict)) #Разбиваем "очищенное" предложение на токены

bow = CountVectorizer() #Инициализируем объект CountVectorizer
BOW = bow.fit_transform(word_token) #Через функцию fit_transform
 образуем мешок слов
bagOFwords = pd.DataFrame(BOW.toarray()) #Преобразуем в DataFram
e для вывода
bagOFwords.columns = bow.get_feature_names() #Добавим подписи к
столбцам

print(bagOFwords.head())
```

```
Out:          класс  погода  супер
          0      0       1      0
          1      1       0      0
          2      0       1      0
          3      0       0      1
```

Как видно из этого примера, в предложении «Погода класс! Погода супер!» слово «погода» встречается два раза и, если мы обратим внимание на вывод, а именно на столбец соответствующего слова, то можно заметить две «1». Это говорит о том, что слово «погода» встречается в тексте два раза, а все остальные один раз.

Метод TF-IDF

Представление «мешок слов», о котором рассказывалось ранее, описывает документы без учета контекста корпуса. Более удачным методом и более распространённым является метод TF-IDF (TF — частота слова, term frequency, IDF — обратная частота документа, inverse document frequency).

TF-IDF - статистическая мера, используемая для оценки важности слова в контексте документа. Вес некоторого слова пропорционален частоте употребления этого слова в документе и обратно пропорционален частоте употребления слова во всех документах коллекции. Размер вектора в данной модели растет пропорционально размеру словаря, однако его можно ограничить, используя только самые частотные слова.

TF-IDF состоит из двух компонентов: Term Frequency (частота слова в документе) и Inverse Document Frequency (обратная частота документа).

$$TF_{token_i} = \frac{n_i}{N_i} \qquad IDF_{token} = log\frac{p}{P}$$

где n_i - сколько раз встречается токен в i-ом документе, N_i - общее количество токенов в i-ом документе, p – общее количеств документов, P – количество документов, в которых встречается токен.

В конечном счете, TF-IDF – это произведение TF на IDF:

$$TF\,IDF = TF \times IDF$$

Поскольку отношение логарифмической функции IDF больше или равно 1, величина TF–IDF всегда больше или равна нулю. Соответственно, согласно нашим представлениям, чем ближе к 1 оценка TF–IDF слова, тем более информативным является это слово для данного документа. И наоборот, чем ближе эта оценка к нулю, тем менее информативно слово.

В качестве примера рассмотрим текст, содержащий 100 слов, в котором слово «телефон» появляется 5 раз. Параметр TF для слова «телефон» равен (5/100) = 0,05. Теперь предположим, что у нас 10 миллионов документов, и

слово телефон появляется в тысяче из них. Коэффициент вычисляется как log(10 000 000/1000) = 4. Таким образом, TD-IDF равен 0,05 * 4 = 0,20.

В библиотеке Scikit-Learn есть преобразователь TfidfVectorizer, в модуле feature_extraction.text, предназначенный для векторизации документов с оценками TF–IDF. Внутренне TfidfVectorizer вызывает преобразователь CountVectorizer, который мы использовали для превращения мешка слов в количества вхождений лексем, а затем TfidfTransformer, нормализующий эти количества обратной частотой документа.

На входе TfidfVectorizer принимает последовательность имен файлов, объектов файлов или строк с коллекцией исходных документов, подобно преобразователю CountVectorizer. В результате применяются методы по умолчанию лексемизации и предварительной обработки, если не указаны другие функции.

В Python подобных подход векторизации будет выглядеть следующим образом:

```
from sklearn.feature_extraction.text import TfidfVectorizer #Под
ключаем преобразователь

text = 'У нас дома есть две собаки и одна кошка. Кошка очень умн
ая, а собаки игривые!' #Пример текста
sent_tokens = nltk.sent_tokenize(text) #Разбиваем предложение на
 токены

tfidf = TfidfVectorizer() #Инициализация преобразователя
res_tfidf = tfidf.fit_transform(sent_tokens) #Через функцию fit_
transform образуем матрицу TF-IDF

import pandas as pd #Подключим библиотеку pandas для вывода резу
льтатов в виде таблицы
pd.DataFrame(data = res_tfidf.toarray(),columns = tfidf.get_feat
ure_names())
```

Out:

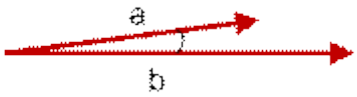
	две	дома	есть	игривые	кошка	нас	одна	очень	собаки	умная
0	0.407824	0.407824	0.407824	0.000000	0.29017	0.407824	0.407824	0.000000	0.29017	0.000000
1	0.000000	0.000000	0.000000	0.499221	0.35520	0.000000	0.000000	0.499221	0.35520	0.499221

Результатом работы преобразователя является разреженная матрица, где каждый ключ является парой «документ/лексема», а значением служит оценка TF–IDF.

Оценка косинус сходства

Теперь, когда у нас есть эта матрица, мы можем легко вычислить оценку сходства. Есть несколько вариантов сделать это; такие как Евклидово, Пирсоновское и косинусное сходства. Опять же, нет правильного ответа, какой счёт является лучшим, но предпочтение отдадим последнему, так как он хорошо работает в сочетании с методом TF-IDF.

Косинусное расстояние, также известное как косинусное сходство, является мерой величины разности между двумя индивидуумами, используя косинусное значение угла между двумя векторами в векторном пространстве. Чем ближе значение косинуса к 1, это указывает, что включенный угол ближе к 0 градусам, тем два вектора более похожи. На рисунке ниже угол между двумя векторами a и b очень мал. Можно сказать, что вектор a и вектор b имеют высокое сходство.



Таким образом, можно сказать, что Косинусное сходство – это мера сходства между двумя векторами, которая используется для измерения косинуса угла между ними. В математическом виде, оценка косинус сходства выглядит так:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

Где A и B – векторы.

В библиотеке Scikit-Learn есть так же функция, позволяющая получить оценку косинусного сходства. Называется она `cosine_similarity()`. В качестве своих параметром, эта функция принимает два вектора, а их мы получили как раз векторизатором TF-IDF:

```
from sklearn.feature_extraction.text import TfidfVectorizer #Под
ключаем преобразователь

text = 'У нас дома есть две собаки и одна кошка. Кошка очень умн
ая, а собаки игривые!' #Пример текста
```

```
sent_tokens = nltk.sent_tokenize(text) #Разбиваем предложение на
 токены

tfidf = TfidfVectorizer() #Инициализация преобразователя
res_tfidf = tfidf.fit_transform(sent_tokens) #Через функцию fit_
transform образуем матрицу TF-IDF

from sklearn.metrics.pairwise import cosine_similarity #Из библи
отеки sklearn подключим функцию, высчитывающую косинус сходство
cosine_similarity(res_tfidf[-1], res_tfidf)

    Out: array([[0.20613697]])
```

В итоге получим, что наши два предложения, не сильно и похожи друг на друга, т.к. результирующее значение далеко от единицы.

Распознавание именованных сущностей (NER)

Теперь, когда мы освоили приемы преобразования текста в числа и научились оценивать подобия векторов, стоит сделать шаг назад и задуматься о такой проблеме как именованные сущности.

Когда человек читает книгу, он без труда понимает, что какие-то слова в тексте – это имя героя, а какие-то – название местности, даже если он впервые столкнулся с таким именем или названием. Для компьютера работа по распознаванию имен людей, названий организаций, топонимов и т.п. оказалась довольно сложной, но решаемой.

Что такое NER? Имена людей, названия организаций, книг, городов, и другие имена собственные называют «именованные сущности» (named entities), а саму задачу — «распознавание именованных сущностей», что на английском означает «Named Entity Recognition» или коротко NER; это сокращение регулярно используется и в русскоязычных текстах.

За одной задачей NER, на самом деле, стоит две:

1) обнаружить, что какая-то последовательность слов — это именованная сущность;

2) понять, к какому классу (имя человека, название организации, город и т.п.) эта именованная сущность относится.

На первый взгляд кажется, что с именованными сущностями не должно возникнуть особых проблем, большинство из них – имена собственные, они пишутся с большой буквы, так что найти их в тексте не сложно, однако не тут-то было. Во-первых, во многих языках вообще нет больших букв (например, в китайском или арабском), а во-вторых, в европейских языках приходится сталкиваться с некоторыми трудностями:

1) Именованные сущности редко состоят из одного слова. Например, из предложения «Звонил доктор Владимир Бомгард» нужно извлечь, как минимум, имя и фамилию — «Владимир Бомгард», а для многих задач полезно уметь находить полное «наименование» человека, о котором говорится: «доктор Владимир Бомгард».

2) Не всегда очевидны границы, например, словосочетание «Иван Васильевич и партнеры» может быть названием некоторой организации, а может подразумевать некоторого Ивана Васильевича и отдельно его партнеров.

3) С большой буквы пишутся не только имена собственные. Это верно и для русского, и еще больше для английского, но особенно хорошо видно в немецком, где с большой буквы пишутся вообще все существительные.

Хотя большинство именованных сущностей состоит из нескольких слов, при решении этой задачи обычно рассматривают отдельные слова и решают, является ли это слово частью именованной сущности или нет. При этом различают начало, середину и конец именованной сущности. При разметке именованных сущностей обычно принято использовать префикс «B» (beginning) для обозначения первого слова, «E» — для последнего слова и «I» (intermediate) — для всех слов между. Иногда также используется префикс «S» (single) для обозначения именованной сущности, состоящей из одного слова.

Таким образом, задача сводится к пословной классификации. Сейчас для построения классификатора обычно тренируют нейросетевую модель на большом количестве текстов с разметкой именованных сущностей. Хотя хорошие результаты дают и классические классификаторы, работающие с пред заданным множеством признаков. Большие буквы или нестандартное использование больших и маленьких букв (iPhone), а также специфических символов (H&M) внутри слова — все это полезные признаки для выявления именованных сущностей.

Мы же будем использовать известные нам библиотеки и модули, такие как NLTK, SpaCy, Natasha и другие. При этом, поиск именованных сущностей в библиотеках NLTK и SpaCy мы уже проводили, когда извлекали POS-теги слов, но напомню, что специализируются они больше на англоязычные тексты.

Для поиска русских именованных сущностей и для обработки русского текста в целом, в первую очередь, стоит обратить свое внимание на библиотеку Natasha, которая и была разработана для этих целей.

Рассмотрим небольшой отрывок из знаменитого произведения Федора Достоевского «Преступление и наказание»:

Действительно, я у Разумихина недавно еще хотел было работы просить, чтоб он мне или уроки достал, или что-нибудь... — додумывался Раскольников, — но чем теперь-то он мне может помочь? Положим, уроки достанет, положим, даже последнею копейкой поделится, если есть у него копейка, так что можно даже и сапоги купить, и костюм поправить, чтобы на уроки ходить... гм... Ну, а дальше? На пятаки-то что ж я сделаю? Мне разве того теперь надобно? Право, смешно, что я пошел к Разумихину...

В этом отрывке есть как минимум 2 уникальные именованные сущности – это Разумихин и Раскольников. Давайте попробуем извлечь их методами и функциями библиотеки Natasha:

```
import natasha as nt


#Инициализируем вспомогательные объекты библиотеки
segmenter = nt.Segmenter()
morph_vocab = nt.MorphVocab()

emb = nt.NewsEmbedding()
morph_tagger = nt.NewsMorphTagger(emb)
ner_tagger = nt.NewsNERTagger(emb)

#Задаем функцию, для извлечения именнованных сущностей
names_extractor = nt.NamesExtractor(morph_vocab)

#Пример текста
text = '''Действительно, я у Разумихина недавно еще хотел было р
аботы просить, чтоб он мне или уроки достал, или что-
нибудь... — додумывался Раскольников, — но чем теперь-
то он мне может помочь?
Положим, уроки достанет, положим, даже последнею копейкой подели
тся, если есть у него копейка, так что можно даже и сапоги купит
ь,
и костюм поправить, чтобы на уроки ходить... гм... Ну, а дальше?
 На пятаки-то что ж я сделаю? Мне разве того теперь надобно?
Право, смешно, что я пошел к Разумихину...'''

#Передадим наш текст в объект Doc из библиотеки и совершим над н
им несколько операций
doc = nt.Doc(text)
doc.segment(segmenter) #Токенизация
doc.tag_morph(morph_tagger) #Определим морфологические POS-теги
doc.tag_ner(ner_tagger) #Определим POS-теги NER

#Нормализуем наш текст
for span in doc.spans:
    span.normalize(morph_vocab)
```

```
#Извлечем именнованные сущности
for span in doc.spans:
    if span.type == nt.PER:
        span.extract_fact(names_extractor)

#Выведем сущности в виде словаря
names_dict = {_.normal: _.fact.as_dict for _ in doc.spans if _.f
act}
print(names_dict)

Out: {'Разумихин': {'last': 'Разумихин'}, 'Раскольников':
{'last': 'Раскольников'}}
```

В результате выполнения этой программы получили наши именованные сущности, а именно Разумихин и Раскольников. При этом программа определила, что это у нас в тексте присутствуют не просто имена, а фамилии персонажей и более того привела их в форму именительного падежа.