ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ЮФУ
РОСТОВ-НА-ДОНУ · ТАГАНРОГ

110 лет ЮФУ

ИНСТИТУТ
МАТЕМАТИКИ
МЕХАНИКИ
КОМПЬЮТЕРНЫХ
НАУК

Основы работы с данными для ИИ

Лекция 9
Обработка текстовых данных (NLP)

Доц
Екатерина Александровна
Преподаватель кафедры информатики и вычислительного эксперимента

Что такое Natural Language Processing?

Natural Language Processing (NLP) - Обработка естественных языков - область, включающая в себя различные методы обработки и анализа текста компьютером

NLP находится на пересечении компьютерных наук, лингвистики и искусственного интеллекта.
Не путайте область NLP с нейролингвистическим программированием.

Natural Language Processing– эта область которая включает себя любую обработку текста компьютером

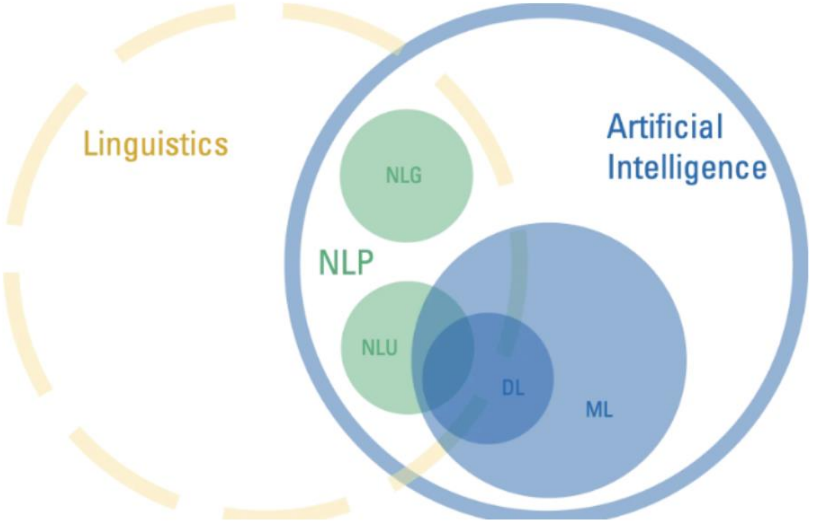
Что такое Natural Language Processing?

На картинке ниже можно заметить, что NLP находится на пересечении таких наук как лингвистика и искусственный интеллект, а также на пересечении Computer Science. NLP также подразделяется на две под области это **NLU** и **NLG:**

Natural Language Understanding (NLU) - анализ естественных языков (понимание текста)

- Синтаксический анализ - анализ структуры языков (например сказуемое и подлежащее)
- Семантический анализ - анализ значения, смысла

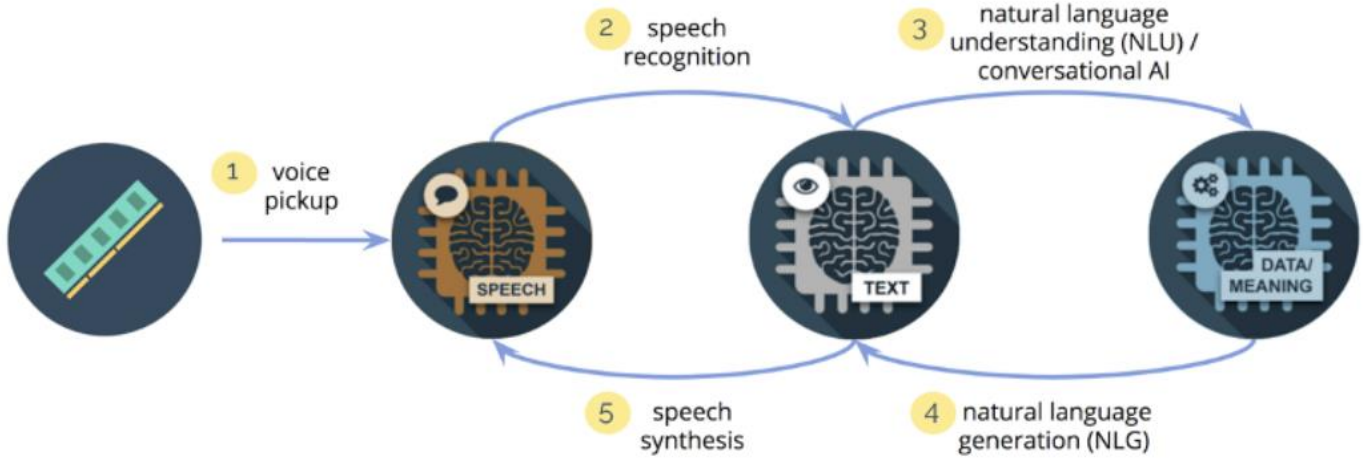
Natural Language Generation (NLG) - генерация естественных языков (генерация текстов)



Что такое Natural Language Processing?

Пример работы **голосового ассистента**.

Ниже изображена небольшая схема как может работать какой-нибудь голосовой ассистент. Первый шаг – мы получаем голосовую запись, далее происходит распознавание речи, на втором шаге звук переводится в текст, потом на этапе **NLU** преобразуется в какие-то признаки, на основе которых модель может принять какое-то решение и выдать какой-то определенный ответ. На четвертом шаге мы получаем текст ответа, и на пятом шаге мы синтезируем из этого текста голос. Потом этот голос проигрывается пользователю.



Цели NLP

- Понимание структуры и значения человеческой речи
- Создание систем, способных на
  - понимание и анализ текста и речи
  - извлечение фактов и информации из текста
  - генерацию текста и речи
- Разработка алгоритмов и моделей для текста

Наиболее значимые цели, которые можно потом подразделить – это **понимание человеческой речи** и **синтез человеческой речи**, а также, разумеется, это разработка ML-алгоритмов для этих двух таких больших целей.

Цели NLP

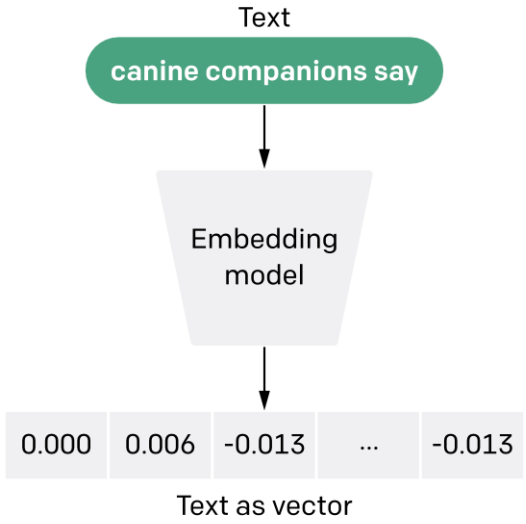
NLP позволяет:

- Обрабатывать огромные массивы текста или речи
- Автоматизировать бизнес-процессы
- Искать информацию в массиве данных
- Решение узких практических задач
- В качестве примера автоматизации бизнес-процессов можно привести пример замены 90% запросов в службу поддержку ответами чат-бота. В большей степени позволяет решать более узкие практические задачи, которых на самом деле очень много в каждой сфере то ли это медицина, то ли это юриспруденция, везде есть свои специфики, в каждом домене (области) свои слова и свои embeddings.

Embedding

Embedding – это векторное представление слова, которое кодирует его смысл при помощи **чисел** в каком-то пространстве, и в этом пространстве мы ожидаем, что слова с близким смыслом будут расположены рядом, а слова с разными смыслами далеко друг от друга

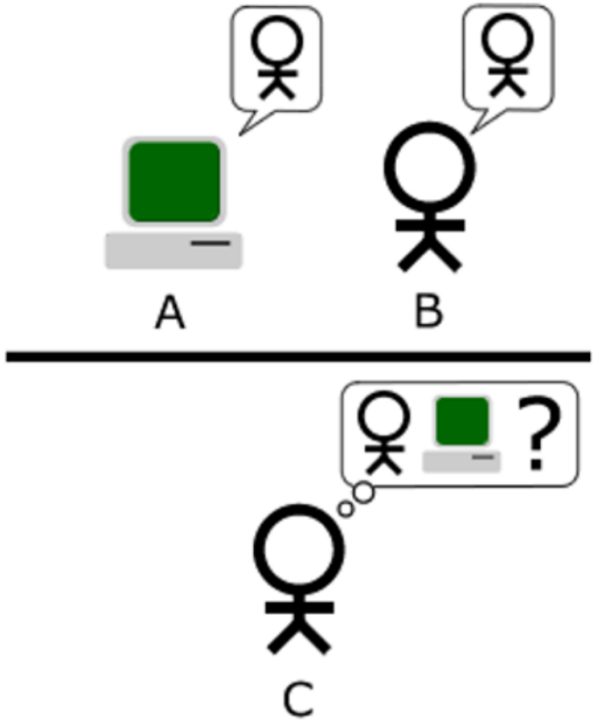
Embedding model давайте пока представим в виде "**черного ящика**", который на выходе выдает массив с числами - кодирует наше слово



Тест Тьюринга

Стандартная интерпретация этого теста звучит следующим образом:

«Человек взаимодействует с одним компьютером и одним человеком. На основании ответов на вопросы он должен определить, с кем он разговаривает: с человеком или компьютерной программой. Задача компьютерной программы — ввести человека в заблуждение, заставив сделать неверный выбор»



Тест Тьюринга

В 1950 году **Алан Тьюринг** задумался о том, а может ли машина мыслить. Он придумал такой эксперимент, у которого на самом деле задача немножко другая. Например, вы зашли на какой-то сайт, у вас в углу всплыл чат-бот. вам нужно определить на том конце с вами общается автоматизированный чат-бот или же человек. У вас есть несколько вопросов, которые вы можете задать, получить ответ. Нужно решить, является ли на том конце компьютер или же человек. В этом и есть суть данного эксперимента. Участники не видят друг друга. Если судья не может сказать определенно, кто человек, а кто робот, то машина прошла тест

Что такое токенизация, и зачем она нужна?

Допустим, у нас есть какой-то текст, и мы **не можем** его просто так взять и отправить в модель классификации - нам нужно его как-то представить, в каком-то численном виде.

Токенизация — это как раз процесс разделения текста на какие-то условные единицы, чаще всего это слова, но, могут быть и другие структурные единицы. Токенизация на предложения тоже существует, и часто ее называют **сегментацией**.

В качестве примера можно взять предложение «*трудолюбивые студенты старательно учатся обработке естественных языков*», мы хотим разбить это предложение по словам. Самый простой способ - взять и разделить по пробелам.

Что такое токенизация, и зачем она нужна?

Как мы видим, **предложение разделилось на слова**, но остались недочеты: во-первых, нужно ли нам сохранять слова, которые начинаются с заглавной буквы такими, или же перевести эту заглавную букву в срочную? Также можно заметить, что в последнем слове точка оказалась частью последнего слова. По-хорошему, этого быть не должно.

Можно **сделать токенизацию немного сложнее**: мы можем убрать заглавную букву, переведя весь текст в нижний регистр, также можем окружить знаки препинания, такие, как точка, запятая, восклицательные и вопросительные знаки пробелами, чтобы после вызова метода split() они стали отдельными токенами

"Трудолюбивые студенты старательно учатся обработке естественных языков."

-> ["Трудолюбивые", "студенты", "старательно", "учатся", "обработке", "естественных", "языков"]

Что такое токенизация, и зачем она нужна?

Как мы видим, **предложение разделилось на слова**, но остались недочеты: во-первых, нужно ли нам сохранять слова, которые начинаются с заглавной буквы такими, или же перевести эту заглавную букву в срочную? Также можно заметить, что в последнем слове точка оказалась частью последнего слова. По-хорошему, этого быть не должно.

Можно **сделать токенизацию немного сложнее**: мы можем убрать заглавную букву, переведя весь текст в нижний регистр, также можем окружить знаки препинания, такие, как точка, запятая, восклицательные и вопросительные знаки пробелами, чтобы после вызова метода split() они стали отдельными токенами

```
"Трудолюбивые студенты старательно учатся обработке естественных языков."
-> ["Трудолюбивые", "студенты", "старательно", "учатся", "обработке", "естественных", "языков"]
```

Токенизация

```
import re s = "Трудолюбивые студенты старательно учатся
обработке естественных языков."
s = s.lower()
s = re.sub("([.,!?])", r" \1 ", s) # окружить данные символы
пробелами
s = re.sub(r" {2,}", " ", s) # убрать дублирующие пробелы
tokens = s.split()
print(s)
```

Векторизация

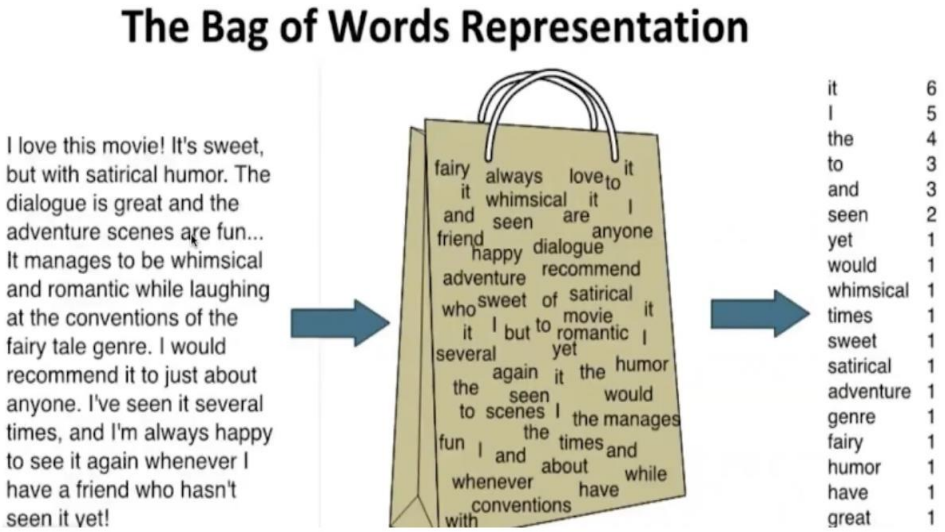
Векторизация — это перевод текста (слов/предложений) в **числа**: вектор или матрицу признаков, с которыми уже может работать модель

Основные способы векторизации

1) Bag-of-Words (CountVectorizer)

Bag-of-Words (BoW) или мешок-слов -- неупорядоченный набор слов, входящих в обрабатываемый текст. В этой модели текст (одно предложение или весь документ) представляется в виде "мешка" его слов без какого-либо учета грамматики и порядка слов, но с сохранением информации об их количестве.

- Игнорирует порядок
- Переводит человеческий язык слов в понятный для компьютера язык цифр



Основные способы векторизации

Здесь еще один пример, дано слева 5 предложений сверху мы можем заметить наш словарь: все слова, которые есть в нашем примере. Как можно заметить, в предложениях есть слово Матрица, которого нет в словаре, и получилось, что это слово – **out-of-vocabulary**. Это отдельная проблема в NLP, которую можно решать разными методами. Мы представили каждое предложение в виде численного представления, и уже на этой матрице можно учить нашу логистическую регрессию.

	it	is	puppy	cat	pen	a	this
it is a puppy	1	1	1	0	0	1	0
it is a kitten	1	1	0	0	0	1	0
it is a cat	1	1	0	1	0	1	0
that is a dog and this is a pen	0	2	0	0	1	2	1
it is a matrix	1	1	0	0	0	1	0

Основные способы векторизации

```
from sklearn.feature_extraction.text import CountVectorizer
texts = [
"Мне понравился фильм!",
"Совсем не рекомендую.",
"Актёры играли отлично."
]
X = CountVectorizer(ngram_range=(1,2), max_features=20000).fit_transform(texts)
X.toarray()
```

Вывод:

```
array([[0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1],
       [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0],
       [1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]])
```

NLTK

Теперь перейдем к уже существующим инструментам: известная в NLP библиотека – nltk (naturallanguage toolkit), которая позволяет токенизировать, стеммировать, предобрабатывать, тегировать тексты и предоставляет удобный интерфейс ко множеству различных датасетов.

Сейчас же мы только посмотрим как работает метод word_tokenize(). Для того, чтобы запустить этот метод, нам нужно импортировать библиотеку и установить пакет punkt.

```
import nltk nltk.download('punkt’)


nltk.word_tokenize("Студенты хотят получить зачет по предмету")
```

```
['Трудолюбивые', 'студенты', 'старательно', 'учатся', 'обработке',
'естественных', 'языков', '.']
```

NLTK

Как мы видим, предложение токенизировалось, точка выделилась в отдельный токен, и никаких дополнительных действий делать не нужно. Также nltk позволяет делить текст подлиннее на предложения:

```
s = "В машинном обучении используются специальные обработчики исходного текста, которые позволяют разбить его на блоки, пригодные для подачи в модель. Такие обработчики называются токенизаторами (англ. Tokenize). Токенизация — самый первый шаг при обработке текста, результатом которого является набор (список) так называемых токенов (подстрок)."
nltk.sent_tokenize(s)
```

['В машинном обучении используются специальные обработчики исходного текста,
которые позволяют разбить его на блоки, пригодные для подачи в модель.',
'Такие обработчики называются токенизаторами (англ.',
'Tokenize).',
……)

NLTK

Можно заметить, что такая токенизация работает не идеально: первое предложение отделилось хорошо, а вот место, где внутри текста есть сокращение, модель посчитала концом предложения, что, конечно, неправильно.

nltk - многоязычная библиотека, она работает на множестве разных языков, но есть и более специфические библиотеки, например, rusenttokenize, которая токенизирует предложения именно **на русском языке**.

NLTK

```
!pip install rusenttokenize # установить библиотеку rusenttokenize


from IPython.display import clear_output
clear_output() # удалить вывод ячейки


from rusenttokenize import ru_sent_tokenize
s = "В машинном обучении используются специальные обработчики исходного текста,
которые позволяют разбить его на блоки, пригодные для подачи в модель. Такие
обработчики называются токенизаторами (англ. Tokenize). Токенизация — самый
первый шаг при обработке текста, результатом которого является набор (список) так
называемых токенов (подстрок)."
ru_sent_tokenize(s)
```

NLTK

Результат

['В машинном обучении используются специальные обработчики исходного текста, которые позволяют разбить его на блоки, пригодные для подачи в модель.',
'Такие обработчики называются токенизаторами (англ. Tokenize).',
'Токенизация — самый первый шаг при обработке текста, результатом которого является набор (список) так называемых токенов (подстрок).']

Лемматизация

Разобрав токенизацию, можно перейти к **лемматизации**. Допустим, у нас есть слово «*трудолюбивые*». В русском языке слова имеют **разные формы**, то есть, в зависимости от падежа, у слова изменяется окончание. Нужно ли модели считать что эти слова разные, или же это одни и те же слова? Разумеется, мы бы хотели, чтобы **формы слова учитывались как одно слово**, потому что они в сущности одинаковые, и отличается только **окончание**, поэтому мы бы хотели, чтобы слово, которое встретилось в какой-то нестандартной форме, оказалось в своей нормальной форме, и потом поступило бы в модель в этой нормальной форме. В случае слова «*трудолюбивые*» это будет «*трудолюбивый*», в случае какого-то глагола это будет его инфинитивная форма.

Лемматизация

Лемматизация — это вид предобработки, в котором мы каждое слово заменяем его нормальной формой.

Если у нас есть предложение «*трудолюбивые студенты старательно учатся обработке естественных языков*», то после запуска модели у нас получится «*трудолюбивый студент старательно учиться обработка естественный язык*».

Лемматизация

```
import spacy


nlp = spacy.load("ru_core_news_sm")
lemmatizer = nlp.get_pipe("lemmatizer")


s2 = 'трудолюбивые студенты старательно учатся обработке естественных языков'
print(lemmatizer.mode)  # какой лемматизатор используется
print([token.lemma_ for token in nlp(s2)])
```

```
pymorphy3
['трудолюбивый', 'студент', 'старательно', 'учиться', 'обработка',
'естественный', 'язык']
```

Лемматизация

Как можно заметить, мы использовали модель ***Spacy***, ту же самую для русского языка.

Если вывести у этого лемматизатора атрибут mode, то видно, что модель, которая используется внутри Space — это ***pymorphy3***.

pymorphy3 - это отдельная библиотека для русского языка, которая позволяет производить много разных манипуляций, в том числе и лемматизацию.

```
import pymorphy3
import nltk


morph = pymorphy3.MorphAnalyzer()


s2 = 'трудолюбивые студенты старательно учатся обработке естественных языков'
print([morph.parse(w)[0].normal_form for w in nltk.word_tokenize(s2)])
```

Лемматизация

Результат

['трудолюбивый', 'студент', 'старательно', 'учиться', 'обработка', 'естественный', 'язык']

Стемминг

Лемматизация не самая простая вещь, которую можно попробовать, если у нас есть какая-то проблема с формами слов. У разных слов разные окончания, и можно привести эти разные слова в одну и ту же форму, просто обрезав эти окончания.

Стемминг как раз преследует эту идею, превращая слово «*учатся*» в «*учат*», «*старательно*» в «*старательно*» и так далее. Чтобы сделать это в коде можно использовать пакет stem из библиотеки nltk, в котором есть ***snowball stemmer***, который также поддерживает русский.

```
import nltk
from nltk.stem.snowball import SnowballStemmer


stemmer = SnowballStemmer(language="russian")
s2 = 'трудолюбивые студенты старательно учатся обработке естественных языков'
for token in nltk.word_tokenize(s2):
    print(token + ' --> ' + stemmer.stem(token))
```

Стемминг

Результат

трудолюбивые --> трудолюбив

студенты --> студент

старательно --> старательн

учатся --> учат

обработке --> обработк

естественных --> естествен

языков --> язык

Стоп-слова

Допустим, мы хотим найти в Яндексе какие-то статьи по запросу «*коврик для мыши*», и получаем в выдаче 3 статьи:

первая содержит все три слова по паре раз

вторая статья содержит только слово «*коврик*» несколько раз

третья содержит только предлог «*для*» 1000 раз.

Нужно ли эту третью статью оценивать выше чем предыдущие, потому что там одно из слов встречается чаще? Разумеется, нет. Мы этого не хотим, потому что предлог «*для*» **не несет большой информации,** это **стоп-слово** не имеющее особого смысла.

Стоп-слова

В nltk есть пакет, который позволяет для многих языков получить список стоп-слов. В нем, в основном, союзы, предлоги и некоторые наречия.

```
from nltk.corpus import stopwords
nltk.download('stopwords’)


for i in range(len(stopwords.words("russian")) // 10 + 1):
      print(stopwords.words("russian")[10 * i:10 *(i + 1)])
```

['и', 'в', 'во', 'не', 'что', 'он', 'на', 'я', 'с', 'со']
['как', 'а', 'то', 'все', 'она', 'так', 'его', 'но', 'да', 'ты']
['к', 'у', 'же', 'вы', 'за', 'бы', 'по', 'только', 'ее', 'мне']
['было', 'вот', 'от', 'меня', 'еще', 'нет', 'о', 'из', 'ему', 'теперь']
['когда', 'даже', 'ну', 'вдруг', 'ли', 'если', 'уже', 'или', 'ни', 'быть']
…

Стоп-слова

Попробуем убрать стоп-слова

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


s3 = "которые позволяют разбить его на блоки, пригодные для подачи в модель"
[w for w in nltk.word_tokenize(s3) if w not in stopwords.words("russian")]
```

['которые',
 'позволяют',
 'разбить',
 'блоки',
 ',',
 'пригодные',
 'подачи',
 'модель']

Инструменты

nltk также содержит в себе множество разных **датасетов**, в свое время они были наиболее популярными данными на которых проводили все эксперименты. Это размеченные датасеты ***Brown*** и ***Reuters***. ***Reuters*** - это новостное агентство, которое выкладывало огромное количество новостей по жанрам.

В ***Brown*** можно получить слова из какой-то конкретной категории, например, из новостей. Также можно итерироваться по отдельным файлам - взять информацию из отдельного файла или из сразу нескольких категорий. Таким образом, nltk позволяет нам протестировать наш код на каких-то данных не загружая их дополнительно из интернета, и это достаточно качественные данные, на которых уже проводилось очень много экспериментов.

Инструменты

```
import nltk
from nltk.corpus import brown, reuters


nltk.download("brown")
brown.raw()[:100]
```

```
'\n\n\tThe/at Fulton/np-tl County/nn-tl Grand/jj-tl Jury/nn-tl said/vbd Friday/nr
an/at investigation/nn'
```

Инструменты

Pymorphy2 - это библиотека для русского языка, которая позволяет делать множество разных манипуляций.

```
import pymorphy2
morph = pymorphy2.MorphAnalyzer()
morph.parse('стали')
```

[Parse(word='стали', tag=OpencorporaTag('VERB,perf,intr plur,past,indc'), normal_form='стать', score=0.975342, methods_stack=((DictionaryAnalyzer(), 'стали', 945, 4),)),
Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn sing,gent'), normal_form='сталь', score=0.010958, methods_stack=((DictionaryAnalyzer(), 'стали', 13, 1),)),
Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn plur,nomn'), normal_form='сталь', score=0.005479, methods_stack=((DictionaryAnalyzer(), 'стали', 13, 6),)),
Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn sing,datv'), normal_form='сталь', score=0.002739, methods_stack=((DictionaryAnalyzer(), 'стали', 13, 2),)),
Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn sing,loct'), normal_form='сталь', score=0.002739, methods_stack=((DictionaryAnalyzer(), 'стали', 13, 5),)),
Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn plur,accs'), normal_form='сталь', score=0.002739, methods_stack=((DictionaryAnalyzer(), 'стали', 13, 9),))]

Инструменты

Аналогично можно запустить и при помощи pymorphy3

Здесь можно заметить, что мы парсим слово «*стали*» и первый вариант это **глагол** «*стали*», а второй уже **существительное** – обработанное железо. Также мы получаем **скоры** – оценки того, что модель считает глагол наиболее вероятной частью речи, но все было бы не так интересно, если бы мы только могли получать части речи.

Также можно получить более конкретную информацию о форме слова:

```
p = morph.parse('стали')[0]
p.tag
```

```
OpencorporaTag('VERB,perf,intr plur,past,indc')
```

Сложности в предобработке

Мы рассмотрели токенизацию текста, но также иногда полезно перед этим **предообработать текст**:

- перевести в нижний регистр
- убрать лишние пробелы и переводы строк и табуляцию
- заменить все символы, которые не входят перечень допустимых на пробелы.

Ведь данные бывают грязными, и в них могут быть различные символы, даже иероглифы, которые нужно как-то отфильтровать, и вот это можно сделать регулярным выражением. Затем опять убираем дублирующие пробелы и пробелы в начале и конце строки.

Сложности в предобработке

Это все зависит от конкретной задачи, которую вы решаете. В некоторых задачах лемматизация может хорошо отработать, а в некоторых она может убрать нужную информацию, потому что во время лемматизации мы теряем падежные формы, и можем не понять, к чему относятся конкретные слова в тексте, поэтому не во всех задачах лемматизация применима.

Данные могут быть очень грязными и довольно разнообразными: у нас в мире **7000 языков**, однако наибольшее количество всех текстовых данных написано на паре десятков языков.

Также у нас в языках есть **сокращения**, **упрощения**, когда людям лень написать «*спасибо*», или даже не столько лень, сколько тратить время на написание лишних букв, и появляются новые слова в языке «*asap*» — это английское сокращение «*как можно быстрее*».

Пример простой предобработки

```
import re
import unicodedata


def preprocess_sentence(s):
    # убрать пробелы в начале и в конце строки
    s = s.strip()
    # перевести все символы в нижний регистр
    s = s.lower()
    # отделить пробелами символы ".", ",", "!", "?"
    s = re.sub(r"([.,!?])", r" \1 ", s)
    # заменить на пробелы все символы, кроме а-я, А-Я, ".", ",", "!", "?"
    s = re.sub(r"[^а-яА-Я.,!?]+", " ", s)
    # убрать дублирующие пробелы
    s = re.sub(r"\s{2,}", " ", s)
    # убрать пробелы в начале и в конце строки
    s = s.strip()
    return s
```

Byte-pair-encoding

Теперь перейдем к последней, но, наверное, самой важной части — это **byte-pair-encoding**. Она важнее, чем другие части, потому что используется в самых современных моделях, таких как трансформеры **BERT** и **GPT**.

В **лемматизации** мы **теряем** часть информации, если после лемматизации мы получили «*проблема исправлять обработка*», то какой был изначальный вариант? Сложно сказать: «*проблему исправила обработка*» или «*проблема исправит обработку*» по смыслу можем понять, что первый вариант более вероятен, чем второй, но синтаксически это не очевидно.

Byte-pair-encoding

И **другой момент**: у нас есть составные слова, например, «*трудолюбие*», оно состоит из двух корней это «*труд*» и «*люб*». Или слово «*морфология*», которые тоже состоит из двух частей. И если мы будем иметь в словаре отдельные упоминания, отдельные индексы для слов «*трудолюбие*» «*труд*» и «*люб*», то **модель** никак не будет знать что они **похожи между собой.**

"Трудолюбие" = "труд" + "о" + "люб" + "и" + "е"

"Морфология" = "морф" + "о" + "лог" + "и" + "я"

Как раз эту проблему решает **byte-pair-encoding**, кстати, она также решает проблему **out-of-vocabulary** слов, которых нет в словаре. Как она это делает?

Byte-pair-encoding

Byte Pair Encoding (BPE) -- токенизация, опирающая на сухую статистику и делящая текст на морфемы, делит текст не на слова, а на подслова (subword - "подслова", или же n-gram - N-граммы).

- кодирование слов через подслова
- уменьшение размера словаря
- лучшее качество на редких словах

Алгоритм Byte-pair Encoding

Например, весь наш текст это «*зеленая зелень зеленит зеленую зелень*».

Первоначальный словарь - это просто множество всех отдельных букв, которые есть в изначальном тексте без пробела.

Хотя иногда, конечно, пробел бывает полезным, например, для мультиязычных моделей, потому что в китайском языке пробелы редко ставят, слова могут быть не отделены пробелом, и тогда для других языков нужно оставлять пробел.

Текст: `"зелёная зелень зеленит зелёную зелень"`

Первоначальный словарь: `{'а', 'е', 'з', 'и', 'л', 'н', 'т', 'у', 'ь', 'ю', 'я', 'ё'}`

Алгоритм Byte-pair Encoding

```
text = "зелёная зелень зеленит зелёную зелень"
vocab = set(text) - {' '}
vocab
```

{'а', 'е', 'з', 'и', 'л', 'н', 'т', 'у', 'ь', 'ю', 'я', 'ё'}

Алгоритм Byte-pair Encoding

Здесь же мы разделили все на слова, но слова — это не то что мы хотим, мы хотим получить подслова, самые популярные, самые наиболее встречаемые в этом тексте. Тут самое заметно это «*зел*». Как это можно сделать?

Алгоритм не такой сложный - для всех букв, которые у нас есть в словаре, **считаем пары букв**, которые наиболее **часто встречаются**. Наши кандидаты — это сумма двух букв в словаре.

Алгоритм Byte-pair Encoding

```
from collections import Counter
def count(text):
    candidates = {w1 + w2 for w1 in vocab for w2 in vocab} - vocab
    max_len = max(len(w) for w in candidates)
    occurences = []
    for i in range(len(text)):
        for j in range(1, min(max_len, len(text) - i) + 1):
            if text[i:i + j] in candidates:
                occurences.append(text[i:i + j])
    return Counter(occurences)
text = "зелёная зелень зеленит зелёную зелень"
vocab = set(text) - {' '}
count(text)
```

Алгоритм Byte-pair Encoding

```
Counter({'зе': 5,
         'ел': 5,
         'лё': 2,
         'ён': 2,
         'на': 1,
         'ая': 1,
         'ле': 3,
         'ен': 3,
         'нь': 2,
         'ни': 1,
         'ит': 1,
         'ну': 1,
         'ую': 1})
```

Алгоритм Byte-pair Encoding

Среди всех вариантов мы отбираем те, **которые встречаются наиболее часто** в нашем тексте, то есть в предложении «*зеленая зелень зеленит зеленую зелень*». В итоге мы получили словарь, в который мы оперативно добавляем **токены-подслова,** которые наиболее часто появляются в нашем тексте.

Алгоритм **BPE** позволяет без зависимости от конкретного языка находить отдельные корни и выделять их в слова словаря.

Представленный алгоритм – это псевдокод. Нужно учитывать больше нюансов. Например, в случае когда какая-то морфема употребляется **внутри слова**, а не в начале.

Если у нас есть слово «*овал*» и слово «*бордовый*», «*ов*» **в начале слова имеет другой смысл**, чем «*ов*», к**оторый находится в середине** какого-то слова. Поэтому при делении на токены к нему будут добавлены две решетки, чтобы обозначить что «*ов*» это не новое слово, а **часть старого**.

Инструменты

Есть библиотека tokenizers, которая позволяет использовать токенизатор буквально в несколько строк кода.

Мы просто объявляем токенизатор, учим BPETrainer, записываем наш текст в файл и потом запускаем.

Инструменты

```
!pip install tokenizers


from tokenizers import Tokenizer
from tokenizers.models import BPE
from tokenizers.trainers import BpeTrainer
from tokenizers.pre_tokenizers import Whitespace
tokenizer = Tokenizer(BPE(unk_token="[UNK]"))
trainer = BpeTrainer(vocab_size=20, special_tokens=["[UNK]", "[CLS]", "[SEP]", "[PAD]",
"[MASK]"])
tokenizer.pre_tokenizer = Whitespace()
text = "зелёная зелень зеленит зелёную зелень"
with open("text.txt", "w", encoding="utf-8") as f:
    f.write(text)
tokenizer.train(["text.txt"], trainer)
tokenizer.get_vocab()
```