УНИВЕРСИТЕТ
искусственного
интеллекта

Сегментация текстов

Сегментация текстов

Базовые понятия

Сегментация текстов – сортировка фрагментов входящих текстов согласно их смыслу. На наших занятиях будем использовать сегментацию договоров на 6 смысловых групп.

Морфологический анализ – это определение характеристик слова на основе того, как это слово пишется. При морфологическом анализе не используется информация о соседних словах. Морфологический анализатор Pymorphy2 будет использован нами прежде всего для приведения слов текстов к так называемой нормальной форме, что позволяет сократить многообразие слов, экономит память, а в ряде случаев (когда информация о взаимосвязи слов в документе не так важна по сравнению со словарным составом) улучшает точность при работе нейросетей с текстом.

Word2vec embeddings (Word2vec, W2V) – предварительно обученные (предобученные) модели нейронных сетей, уже имеющие в своём составе огромный запас переведённых в векторное представление слов. Как правило, такие модели работают лучше, чем собственные рукописные эмбеддинги, так как помимо огромной базы слов, на которой они обучены, в них заложены предварительные логические или математические идеи, улучшающие анализ и обработку текста.

Сегментация текстов

Модель от GENSIM, конечно, не единственная, есть модели, которые считаются лучшими по отношению к ней. Но это хорошая классическая модель, обладающая рядом преимуществ, позволяющая понять преимущества этого класса моделей. Рекомендуем для ознакомления с более продвинутыми моделями пакета просмотреть секцию Литература.

Морфологический анализатор Pymorphy2

Морфологический анализ с помощью Pymorphy2

В pymorphy2 для морфологического анализа слов есть класс MorphAnalyzer.

```
>>> import pymorphy2
>>> morph = pymorphy2.MorphAnalyzer()
```

По умолчанию используется словарь для русского языка; чтобы вместо русского включить украинский словарь, с помощью pip установите пакет pymorphy2-dicts-uk и используйте

```
>>> morph = pymorphy2.MorphAnalyzer(lang='uk')
```

Экземпляры класса MorphAnalyzer обычно занимают порядка 15Мб оперативной памяти (т. к. загружают в память словари, данные для предсказателя и т. д.); старайтесь организовать свой код так, чтобы создавать экземпляр MorphAnalyzer заранее и работать с этим единственным экземпляром в дальнейшем.

С помощью метода MorphAnalyzer.parse() можно разобрать отдельное слово:

```
>>> morph.parse('стали')
[Parse(word='стали', tag=OpencorporaTag('VERB,perf,intr plur,past,indc'),
normal_form='стать', score=0.983766, methods_stack=((<DictionaryAnalyzer>,
'стали', 884, 4),)),
```

Морфологический анализатор Pymorphy2

Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn sing,gent'), normal_form='сталь', score=0.003246, methods_stack=((<DictionaryAnalyzer>, 'стали', 12, 1),)),

Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn sing,datv'), normal_form='сталь', score=0.003246, methods_stack=((<DictionaryAnalyzer>, 'стали', 12, 2),)),

Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn sing,loct'), normal_form='сталь', score=0.003246, methods_stack=((<DictionaryAnalyzer>, 'стали', 12, 5),)),

Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn plur,nomn'), normal_form='сталь', score=0.003246, methods_stack=((<DictionaryAnalyzer>, 'стали', 12, 6),)),

Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn plur,accs'), normal_form='сталь', score=0.003246, methods_stack=((<DictionaryAnalyzer>, 'стали', 12, 9),))]

Метод MorphAnalyzer.parse() возвращает один или несколько объектов типа Parse с информацией о том, как слово может быть разобрано.

В приведенном примере слово «стали» может быть разобрано и как глагол («они стали лучше справляться»), и как существительное («кислородно-конверторный способ получения стали»). На основе одной лишь информации о том, как слово пишется, понять, какой разбор правильный, нельзя, поэтому анализатор может возвращать несколько вариантов разбора.

У каждого разбора есть тег:

```
>>> p = morph.parse('стали')[0]
>>> p.tag
OpencorporaTag('VERB,perf,intr plur,past,indc')
```

Тег – это набор граммем, характеризующих данное слово. Например, тег 'VERB,perf,intr plur,past,indc' означает, что слово – глагол (VERB) совершенного вида (perf), непереходный (intr), множественного числа (plur), прошедшего времени (past), изъявительного наклонения (indc).

Доступные граммемы описаны тут: Обозначения для граммем (русский язык).

Морфологический анализатор Pymorphy2

Кроме того, у каждого разбора есть нормальная форма, которую можно получить, обратившись к атрибутам normal_form или normalized:

>>> p.normal_form

'стать'

>>> p.normalized

Parse(word='стать', tag=OpencorporaTag('INFN,perf,intr'), normal_form='стать', score=1.0, methods_stack=((<DictionaryAnalyzer>, 'стать', 884, 0),))

pymorphy2 умеет разбирать не только словарные слова; для не словарных слов автоматически задействуется предсказатель. Например, попробуем разобрать слово «бутявковедами» - pymorphy2 поймет, что это форма творительного падежа множественного числа существительного «бутявковед», и что «бутявковед» – одушевленный и мужского рода:

>>> morph.parse('бутявковедами')

[Parse(word='бутявковедами', tag=OpencorporaTag('NOUN,anim,masc plur,ablt'), normal_form='бутявковед', score=1.0, methods_stack=((<FakeDictionary>, 'бутявковедами', 51, 10), (<KnownSuffixAnalyzer>, 'едами')))]

Постановка слов в начальную форму

Нормальную (начальную) форму слова можно получить через атрибуты Parse.normal_form и Parse.normalized. Чтобы получить объект Parse, нужно сначаларазобрать слово и выбрать правильный вариант разбора из предложенных.

Но что считается нормальной формой? Например, возьмем слово «думающим». Иногда мы захотим нормализовать его в «думать», иногда – в «думающий», иногда – в «думающая».

Посмотрим, что сделает pymorphy2 в этом примере:

>>> morph.parse('думающему')[0].normal_form

'думать'

pymorphy2 сейчас использует алгоритм нахождения нормальной формы, который работает наиболее быстро (берется первая форма в лексеме), поэтому, например, все причастия сейчас нормализуются в инфинитивы.

Морфологический анализатор Pymorphy2

Это можно считать деталью реализации.

Если требуется нормализовывать слова иначе, можно воспользоваться методом Parse.inflect():

```
>>> morph.parse('думающему')[0].inflect({'sing', 'nomn'}).word
'думающий'
```

Gensim

Gensim — библиотека обработки естественного языка, предназначенная для «Тематического моделирования».

С его помощью можно обрабатывать тексты, работать с векторными моделями слов (такими как Word2Vec, FastText и т. д.) и создавать тематические модели текстов.

Введение

Что такое Gensim?

Тематическое моделирование — это метод извлечения основных тем, которым посвящен обрабатываемый текст. В пакете Gensim реализованы основные алгоритмы тематического моделирования LDA и LSI.

Вы можете заметить, что эти алгоритмы доступны и в других пакетах, таких как scikit, R и т. д. Но скорость обработки и качество результата и оценки тематических моделей не имеют аналогов в gensim, плюс в пакете много удобных средств для обработки текста.

Еще одно существенное преимущество gensim: он позволяет обрабатывать большие текстовые файлы, не загружая весь файл в память.

Рассмотрим

- Основные понятия в Gensim
- Что такое словарь и корпус, почему они имеют значение и как их использовать?
- Как создать и работать со словарем и корпусом?
- Как загрузить и работать с текстовыми данными из нескольких текстовых файлов в памяти?
- Как создать модель Word2Vec?

Gensim

Что такое словарь и корпус?

Чтобы работать с текстовыми документами, Gensim требует, чтобы слова (или как еще их называют в данном случае токены) были преобразованы в уникальные идентификаторы. Для этого в Gensim необходимо создать объект Dictionary, который сопоставит каждое слово с уникальным идентификатором.

Итак, как создать Словарь?

Преобразуем текст или предложения в [список слов] и передадим этот список объекту corpora.Dictionary().

Рассмотрим, зачем нужен объект словаря и как его использовать.

Объект словаря обычно используется для создания так называемого мешка слов «bag of words». Именно этот словарь и «bag of words» (Corpus) используются в качестве входных данных для тематического моделирования и других моделей, на которых специализируется Gensim.

Какие типы текстов может обрабатывать Gensim? Входной текст может быть в одной из 3 разных форм:

- Как предложения, хранящиеся в собственном объекте списка Python.
- Как один текстовый файл.
- В нескольких текстовых файлах.

Если вам необходимо обработать большой по объему тестовый файл, то Gensim может загрузить текст и обновить словарь по одной строчке за раз, без загрузки всего текстового файла в системную память. Мы покажем, как это сделать в следующих двух разделах.

Но прежде определимся с терминологией обработки естественного языка.

«Токен» обычно означает «слово». «Документ» обычно может относиться как к «предложению», так и к «абзацу», а «корпус» обычно представляет собой «собрание документов» в виде пакета слов или как его еще называют мешка слов. То есть для каждого документа корпус содержит идентификатор каждого слова и его частоту в этом документе. В результате информация о порядке слов теряется.

Далее рассмотрим, как создать словарь из списка предложений.

Gensim

Как создать словарь из списка предложений?

Вы можете создать словарь из списка предложений, из текстового файла, который содержит несколько строк текста, и из нескольких таких текстовых файлов, содержащихся в каталоге.

Начнем со «Списка предложений». Если у вас есть несколько предложений, вам нужно преобразовать каждое предложение в список слов.

```
import gensim
from gensim import corpora
from pprint import pprint
# How to create a dictionary from a list of sentences?
documents = ["The Saudis are preparing a report that will acknowledge that",
        "Saudi journalist Jamal Khashoggi's death was the result of an",
        "interrogation that went wrong, one that was intended to lead",
        "to his abduction from Turkey, according to two sources."]
documents_2 = ["One source says the report will likely conclude that",
          "the operation was carried out without clearance and",
          "transparency and that those involved will be held",
          "responsible. One of the sources acknowledged that the",
          "report is still being prepared and cautioned that",
          "things could change."]
# Tokenize(split) the sentences into words
texts = [[text for text in doc.split()] for doc in documents]
# Create dictionary
dictionary = corpora.Dictionary(texts)
# Get information about the dictionary
print(dictionary)
#> Dictionary(33 unique tokens: ['Saudis', 'The', 'a', 'acknowledge', 'are']...)
```

В итоге мы получим словарь из 34 уникальных токенов (или слов). Давайте посмотрим уникальные идентификаторы для каждого из этих токенов.

```
# Show the word to id map
print(dictionary.token2id)
#> {'Saudis': 0, 'The': 1, 'a': 2, 'acknowledge': 3, 'are': 4,
#> 'preparing': 5, 'report': 6, 'that': 7, 'will': 8, 'Jamal': 9,
#> "Khashoggi's": 10, 'Saudi': 11, 'an': 12, 'death': 13,
#> 'journalist': 14, 'of': 15, 'result': 16, 'the': 17, 'was': 18,
#> 'intended': 19, 'interrogation': 20, 'lead': 21, 'one': 22,
#> 'to': 23, 'went': 24, 'wrong,': 25, 'Turkey,': 26, 'abduction': 27,
#> 'according': 28, 'from': 29, 'his': 30, 'sources.': 31, 'two': 32}
```

Мы успешно создали объект Dictionary. Gensim будет использовать этот словарь для создания корпуса, в котором слова в документах заменяются соответствующим идентификатором, предоставленным этим словарем.

Если вы загрузите новые документы в будущем, также нужно будет обновить существующий словарь, чтобы включить новые слова.

```
documents_2 = ["The intersection graph of paths in trees",
               "Graph minors IV Widths of trees and well quasi ordering",
               "Graph minors A survey"]
texts_2 = [[text for text in doc.split()] for doc in documents_2]
dictionary.add_documents(texts_2)
# If you check now, the dictionary should have been updated with the new words (tokens).
print(dictionary)
#> Dictionary(45 unique tokens: ['Human', 'abc', 'applications', 'computer', 'for']...)
print(dictionary.token2id)
#> {'Human': 0, 'abc': 1, 'applications': 2, 'computer': 3, 'for': 4, 'interface': 5,
#>  'lab': 6, 'machine': 7, 'A': 8, 'of': 9, 'opinion': 10, 'response': 11, 'survey': 12,
#>  'system': 13, 'time': 14, 'user': 15, 'EPS': 16, 'The': 17, 'management': 18,
#>  'System': 19, 'and': 20, 'engineering': 21, 'human': 22, 'testing': 23, 'Relation': 24,
#>  'error': 25, 'measurement': 26, 'perceived': 27, 'to': 28, 'binary': 29, 'generation': 30,
#>  'random': 31, 'trees': 32, 'unordered': 33, 'graph': 34, 'in': 35, 'intersection': 36,
#>  'paths': 37, 'Graph': 38, 'IV': 39, 'Widths': 40, 'minors': 41, 'ordering': 42,
#>  'quasi': 43, 'well': 44}
```

Как создать словарь из одного или нескольких текстовых файлов?

Вы также можете создать словарь из текстового файла или из каталога, содержащего несколько текстовых файлов.

Приведенный ниже пример считывает файл построчно и использует функцию gensim simple_preprocess для обработки одной строки файла за раз.

Преимущество в том, что вы можете прочитать весь текстовый файл, не загружая файл в память сразу.

Допустим, что файл будет называться файл sample.txt, чтобы продемонстрировать это.

```
from gensim.utils import simple_preprocess
from smart_open import smart_open
import os
# Create gensim dictionary form a single tet file
dictionary = corpora.Dictionary(simple_preprocess(line, deacc=True) for line in open('sample.txt', encoding='utf-8'))
# Token to Id map
dictionary.token2id
#> {'according': 35,
#>  'and': 22,
#>  'appointment': 23,
#>  'army': 0,
#>  'as': 43,
#>  'at': 24,
#>   ...
#> }
```

Мы создали словарь из одного текстового файла.

Теперь рассмотрим, как читать по одной строке из нескольких файлов?

Gensim

Предположим, что у вас есть несколько текстовых файлов в одном каталоге, вам будет необходимо определить новый класс и добавить метод __iter__. Метод __iter__() должен перебирать все файлы в данном каталоге и возвращать обработанный список токенов слов.

Давайте определим класс с именем ReadTxtFiles, который принимает путь к каталогу, содержащему текстовые файлы.

```
class ReadTxtFiles(object):
  def __init__(self, dirname):
     self.dirname = dirname
  def __iter__(self):
     for fname in os.listdir(self.dirname):
        for line in open(os.path.join(self.dirname, fname), encoding='latin'):
           yield simple_preprocess(line)
path_to_text_directory = "lsa_sports_food_docs"
dictionary = corpora.Dictionary(ReadTxtFiles(path_to_text_directory))
# Token to Id map
dictionary.token2id
# {'across': 0,
#  'activity': 1,
#  'although': 2,
#  'and': 3,
#  'are': 4,
#  ...
# }
```

Как создать корпус слов?

Сейчас вы знаете, как создать словарь из списка и из текстового файла.

Следующий важный объект, с которым Вам нужно ознакомиться, чтобы работать в Gensim, — это корпус слов (или, как его еще называют, корпус типа «Мешок Слов»). Все объекты корпуса содержат id слова и его частоту в каждом документе.

После создания словаря все, что Вам нужно сделать, это передать токенизированный список слов в Dictionary.doc2bow()

Gensim

Давайте создадим корпус для простого списка (my_docs), содержащего 2 предложения.

```
# List with 2 sentences
my_docs = ["Who let the dogs out?",
        "Who? Who? Who? Who?"]
# Tokenize the docs
tokenized_list = [simple_preprocess(doc) for doc in my_docs]
# Create the Corpus
mydict = corpora.Dictionary()
mycorpus = [mydict.doc2bow(doc, allow_update=True) for doc in tokenized_list]
pprint(mycorpus)
#> [[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1)], [(4, 4)]]
```

Как это интерпретируется?

В строке 1 (0, 1) означает, что слово с id = 0 появляется один раз в 1-м документе. Аналогично (4, 4) во втором элементе списка означает, что слово с идентификатором 4 встречается 4 раза во втором документе. И так далее.

Если нужно будет преобразовать этот массив обратно в текст:

```
word_counts = [[(mydict[id], count) for id, count in line] for line in mycorpus]
pprint(word_counts)
#> [[('dogs', 1), ('let', 1), ('out', 1), ('the', 1), ('who', 1)], [('who', 4)]]
```

Обратите внимание, что при таком преобразовании теряется порядок слов. Сохраняется только слово и информация о его частоте.

Как создать корпус слов из текстового файла?

Использовать слова из списков в Python довольно просто, потому что весь текст уже в памяти. Однако у вас может быть большой файл, который вы, возможно, не захотите загружать целиком в память.

Вы можете импортировать такие файлы по одной строке за раз, определив класс и функцию __iter__, которая итеративно считывает файл по одной строке и выдает объект корпуса. Но как создать объект корпуса?

Gensim

__iter__ () из BoWCorpus читает строку из файла, обрабатывает ее с помощью simple_preprocess() и передает его в dictionary.doc2bow(). Чем это отличается от класса ReadTxtFiles, который мы создали ранее? В новом классе я использую функцию smart_open() из пакета smart_open, так как оно позволяет построчно открывать и читать большие файлы из различных источников таких, как S3, HDFS, WebHDFS, HTTP или локальных и сжатых файлов.

```
from gensim.utils import simple_preprocess
from smart_open import smart_open
import nltk
nltk.download('stopwords') # run once
from nltk.corpus import stopwords
stop_words = stopwords.words('english')
class BoWCorpus(object):
  def __init__(self, path, dictionary):
     self.filepath = path
     self.dictionary = dictionary
  def __iter__(self):
     global mydict # OPTIONAL, only if updating the source dictionary.
     for line in smart_open(self.filepath, encoding='latin'):
        # tokenize
         tokenized_list = simple_preprocess(line, deacc=True)
        # create bag of words
         bow = self.dictionary.doc2bow(tokenized_list, allow_update=True)
        # update the source dictionary (OPTIONAL)
         mydict.merge_with(self.dictionary)
        # lazy return the BoW
         yield bow
# Create the Dictionary
mydict = corpora.Dictionary()
# Create the Corpus
bow_corpus = BoWCorpus('sample.txt', dictionary=mydict) # memory friendly
# Print the token_id and count for each line.
```

Gensim

```
for line in bow_corpus:
  print(line)
#> [(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (10, 1), (11, 1)]
#> [(12, 1), (13, 1), (14, 1), (15, 1), (16, 1), (17, 1)]
#> ... truncated ...
```

Word2vec embeddings от GENSIM

Модель вхождения слов — это модель, которая может предоставить числовые векторы для данного слова. Используя API Gensim, вы можете загрузить предварительно созданные модели вхождения слов такие, как word2vec, fasttext, GloVe и ConceptNet. Они построены на больших корпусах часто встречающихся текстовых данных таких, как википедия, новости Google и т. д.

Однако если вы работаете в специализированной нише такой, как техническая документация, возможно, у вас не получится получить вложение слов для всех слов. Поэтому в таких случаях желательно тренировать собственную модель.

Gensim Word2Vec также позволяет обучить вашу собственную модель встраивания слов для вашего корпуса.

```
from gensim.models.word2vec import Word2Vec
from multiprocessing import cpu_count
import gensim.downloader as api
# Download dataset
dataset = api.load("text8")
data = [d for d in dataset]
# Split the data into 2 parts. Part 2 will be used later to update the model
data_part1 = data[:1000]
data_part2 = data[1000:]
```

Word2vec embeddings от GENSIM

```
# Train Word2Vec model. Defaults result vector size = 100
model = Word2Vec(data_part1, min_count = 0, workers=cpu_count())
# Get the word vector for given word
model['topic']
#> array([ 0.0512,  0.2555,  0.9393, ... ,-0.5669,  0.6737], dtype=float32)
model.most_similar('topic')
#> [('discussion', 0.7590423822402954),
#>  ('consensus', 0.7253159284591675),
#>  ('discussions', 0.7252693176269531),
#>  ('interpretation', 0.7196053266525269),
#>  ('viewpoint', 0.705356895923614 5),
#>  ('speculation', 0.7021505832672119),
#>  ('discourse', 0.7001898884773254),
#>  ('opinions', 0.6993060111999512),
#>  ('focus', 0.6959210634231567),
#>  ('scholarly', 0.6884037256240845)]
# Save and Load Model
model.save('newmodel')
model = Word2Vec.load('newmodel')
```

Мы обучили и сохранили модель Word2Vec для нашего документа. Однако, когда придет новый набор данных, нам нужно будет обновить модель, чтобы учесть новые слова.

Как обновить существующую модель Word2Vec новыми данными?

В существующей модели Word2Vec вызовите build_vocab() для нового набора данных, а затем вызовите метод train().

```
# Update the model with new data.
model.build_vocab(data_part2, update=True)
model.train(data_part2, total_examples=model.corpus_count, epochs=model.iter)
model['topic']
# array([-0.6482, -0.5468,  1.0688,  0.82  , ... , -0.8411,  0.3974], dtype=float32)
```

Word2vec embeddings от GENSIM

Как извлечь векторы слов, используя предварительно обученные модели Word2Vec и FastText?

Мы только что увидели, как получить векторы слов для модели Word2Vec, которую мы только что обучили. Тем не менее, gensim позволяет загружать самые предварительно обученные модели через API загрузчика. Давайте посмотрим, как извлечь векторы слов из пары таких моделей.

```
import gensim.downloader as api
# Download the models
fasttext_model300 = api.load('fasttext-wiki-news-subwords-300')
word2vec_model300 = api.load('word2vec-google-news-300')
glove_model300 = api.load('glove-wiki-gigaword-300')
# Get word embeddings
word2vec_model300.most_similar('support')
# [('supporting', 0.6251285076141357),
#  ...
#  ('backing', 0.6007589101791382),
#  ('supports', 0.5269277691841125),
#  ('assistance', 0.520713746547699),
#  ('supportive', 0.5110025405883789)]
```

У нас есть 3 разных модели. Вы можете оценить, какая из них работает лучше, используя соответствующую функцию evaluate_word_analogies()

```
# Word2vec_accuracy
word2vec_model300.evaluate_word_analogies(analogies="questions-words.txt")[0]
#> 0.7401448525607863
# fasttext_accuracy
fasttext_model300.evaluate_word_analogies(analogies="questions-words.txt")[0]
#> 0.8827876424099353
# GloVe accuracy
glove_model300.evaluate_word_analogies(analogies="questions-words.txt")[0]
#> 0.7195422354510931
```

Особенности разметки текстов для сегментации

Особенности разметки текстов для сегментации

При любом «обучении с учителем», как известно, необходимы входные данные, которые мы часто обозначаем как Xtrain, и правильные ответы, которые мы часто обозначаем как Ytrain. На уроке сегментации, который мы прошли ранее, класс выделенного на изображении объекта в Ytrain обозначался неким числом, которое мы чаще всего переводим в формат one hot encoding (ohe), при этом мы берём вектор из всех нулей, длина которого равна количеству классов, и записываем в него число 1 только в ту позицию, номер которой от начала соответствует классу. А если объект принадлежит сразу нескольким классам?

Именно такой случай рассматривается нами при сегментации текста. Мы сегментируем слова, причём, слово может принадлежать сразу более чем одному классу – сегменту. В этом случае наиболее простым оказывается поставить единицу не в одной позиции, соответствующей одному классу, а в нескольких, каждая из которых соответствует одному из классов, которым принадлежит слово. Таким образом, если слово, например, принадлежит 1,3 и 5 классам из 6, мы получим вектор, в котором единицы будут стоять в 1,3 и 5 позициях от начала: [1,0,1,0,1,0]

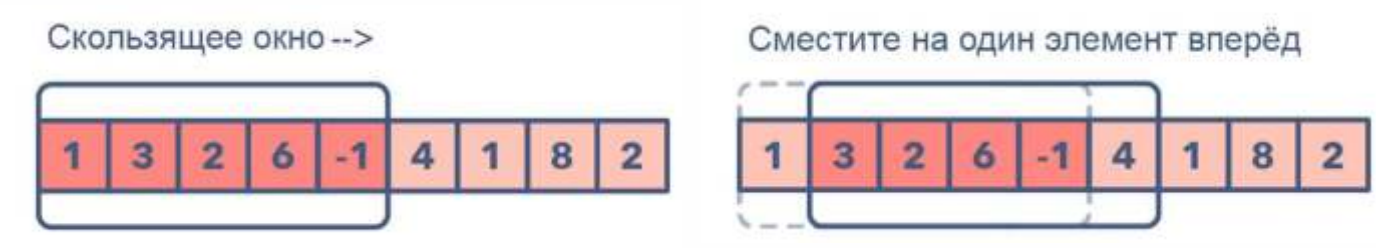
Далее важный момент. Очень неудобно и занимает очень много времени способ помечать каждое слово, как соответствующее какому-либо классу, особенно если слов в тексте очень много. Поэтому при разметке мы считаем, что классу принадлежит не одно слово, а целый фрагмент текста. Мы помечаем начало фрагмента текста специальным значком – тегом начала, а конец – тегом конца. При этом возможна ситуация, когда, например, поставлен тег начала первого типа сегмента – <S1>, через несколько слов – тег начала шестого типа сегмента – <S6>, затем опять несколько слов, тег закрытия первого типа сегмента – </S1>, через несколько слов – тег закрытия шестого типа сегмента – </S6>. То есть в данной конфигурации есть область пересечения классов, и это будет отображено в формируемых векторах меток, в области пересечения будут оба класса отражены в метках. При такой разметке, если она проведена правильно, и нет случайной встречи в тексте, аналогичной тегу набора символов, то, посчитав все теги одного типа, обнаружим, что количество тегов начала и тегов закрытия будет одинаково. Возможна также ситуация, что слова не будут отнесены ни к одному из классов.

При преобразовании текста договоров нужно добиться единого формата, в частности, чтобы каждое слово было отделено с каждой из двух сторон одним пробелом от соседнего. Это нужно для дальнейшего разбиения текста на отдельные слова и обработки его токенизатором. Также при разметке текста некоторые теги не были отделены пробелами от соседних слов, и в тексте кодом этот момент проверяется и исправляется.

Особенности разметки текстов для сегментации

При создании тренировочных и валидационных выборок текст токенизируется, затем токены тегов заносятся в массив (вектор), в котором вначале идут по порядку 6 тегов начала, затем – 6 тегов закрытия, то есть суммарное количество тегов вдвое превышает количество категорий. Далее токены тегов используются для создания меток, соответствующих токенам слов в выборках. Затем из текста убираются токены тегов, таким образом завершается создание выборок.

Поскольку тексты договоров имеют разную длину, мы можем для увеличения выборки и аугментации поступить с нарезкой фрагментов текста так, как ранее поступали с текстами писателей и с аудио, то есть нарезать фрагменты перемещающимся скользящим окном с указанным шагом. При этом шаг меньше размера окна, и фрагменты берутся с частичным перекрытием. Это движение окна для нарезки выборки схематически изображено на рисунке:



Литература

1. https://pymorphy2.readthedocs.io/en/stable/user/guide.html#
2. https://radimrehurek.com/gensim/models/word2vec.html
3. https://webdevblog.ru/gensim-rukovodstvo-dlya-nachinajushhih/