



Deep Learning School

Физтех-Школа Прикладной математики и информатики (ФПМИ) МФТИ

Some parts of the notebook are almost the copy of [mmta-team course](#). Special thanks to mmta-team for making them publicly available. [Original notebook](#).

Прочитайте семинар, пожалуйста, для успешного выполнения домашнего задания. В конце ноутка напишите свой вывод. Работа без вывода оценивается ниже.

Задача поиска схожих по смыслу предложений

Мы будем ранжировать вопросы [StackOverflow](#) на основе семантического векторного представления

До этого в курсе не было речи про задачу ранжирования, поэтому введем математическую формулировку

Задача ранжирования(Learning to Rank)

- X - множество объектов
- $X^l = \{x_1, x_2, \dots, x_l\}$ - обучающая выборка

На обучающей выборке задан порядок между некоторыми элементами, то есть нам известно, что некий объект выборки более релевантный для нас, чем другой:

- $i \prec j$ - порядок пары индексов объектов на выборке X^l с индексами i и j ###

Задача: построить ранжирующую функцию $a : X \rightarrow R$ такую, что

$$i \prec j \Rightarrow a(x_i) < a(x_j)$$

Ranking



Embeddings

Будем использовать предобученные векторные представления слов на постах Stack Overflow.

[A word2vec model trained on Stack Overflow posts](#)

```
In [1]: !wget https://zenodo.org/record/1199620/files/SO_vectors_200.bin?download=1

--2022-10-28 09:45:40-- https://zenodo.org/record/1199620/files/SO_vectors_200.bin?down
load=1
Resolving zenodo.org (zenodo.org)... 188.184.117.155
Connecting to zenodo.org (zenodo.org)|188.184.117.155|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1453905423 (1.4G) [application/octet-stream]
Saving to: 'SO_vectors_200.bin?download=1'

SO_vectors_200.bin? 100%[=====] 1.35G 28.7MB/s in 49s

2022-10-28 09:46:31 (28.1 MB/s) - 'SO_vectors_200.bin?download=1' saved [1453905423/1453
905423]
```

```
In [2]: from gensim.models.keyedvectors import KeyedVectors
wv_embeddings = KeyedVectors.load_word2vec_format("SO_vectors_200.bin?download=1", bina
```

Как пользоваться этими векторами?

Посмотрим на примере одного слова, что из себя представляет embedding

```
In [3]: word = 'dog'
if word in wv_embeddings:
    print(wv_embeddings[word].dtype, wv_embeddings[word].shape)
```

```
float32 (200,)
```

```
In [4]: print(f"Num of words: {len(wv_embeddings.index2word)}")
```

```
Num of words: 1787145
```

Найдем наиболее близкие слова к слову `dog` :

Вопрос 1:

- Входит ли слов `cat` топ-5 близких слов к слову `dog` ? Какое место? Ответ: Нет, не входит. Находится на месте 26

```
In [5]: # method most_similar
for i in range(30):
    print(f"#{i + 1}: {wv_embeddings.most_similar(positive='dog', topn=30)[i]}")
```

```
#1: ('animal', 0.8564180135726929)
#2: ('dogs', 0.7880867123603821)
#3: ('mammal', 0.7623804807662964)
#4: ('cats', 0.7621253728866577)
#5: ('animals', 0.760793924331665)
#6: ('feline', 0.7392398118972778)
#7: ('bird', 0.7315489053726196)
#8: ('animal1', 0.7219215631484985)
#9: ('doggy', 0.7213349938392639)
#10: ('labrador', 0.7209131717681885)
#11: ('canine', 0.7209056615829468)
#12: ('meow', 0.7185295820236206)
#13: ('cow', 0.7080444693565369)
#14: ('dog2', 0.7057910561561584)
#15: ('woof', 0.7050611972808838)
#16: ('dog1', 0.7038840055465698)
#17: ('dog3', 0.701882004737854)
#18: ('penguin', 0.6970292329788208)
#19: ('bulldog', 0.6940488815307617)
#20: ('mammals', 0.6931389570236206)
#21: ('bark', 0.6913799047470093)
#22: ('fruit', 0.6892251968383789)
#23: ('reptile', 0.6891210079193115)
#24: ('furry', 0.6863498687744141)
#25: ('carnivore', 0.6862949728965759)
#26: ('cat', 0.6852341294288635)
#27: ('horse', 0.6833381056785583)
#28: ('kitten', 0.6820152997970581)
#29: ('sheep', 0.6802570223808289)
#30: ('chihuahua', 0.6791757941246033)
```

Векторные представления текста

Перейдем от векторных представлений отдельных слов к векторным представлениям вопросов, как к среднему векторов всех слов в вопросе. Если для какого-то слова нет предобученного вектора, то его нужно пропустить. Если вопрос не содержит ни одного известного слова, то нужно вернуть нулевой вектор.

```
In [6]: import numpy as np
import re
# you can use your tokenizer
# for example, from nltk.tokenize import WordPunctTokenizer
class MyTokenizer:
    def __init__(self):
        pass
    def tokenize(self, text):
        return re.findall('\w+', text)
tokenizer = MyTokenizer()
```

```
In [7]: def question_to_vec(question, embeddings, tokenizer, dim=200):
    """
        question: строка
        embeddings: наше векторное представление
        dim: размер любого вектора в нашем представлении

        return: векторное представление для вопроса
    """
    '''your code'''

    tokens = tokenizer(question)
    q_vec = np.zeros(dim)
    count = 0
    for token in tokens:
        if token in embeddings:
            q_vec += embeddings[token]
            count += 1
    if count != 0:
        q_vec = q_vec / count

    return q_vec
```

Теперь у нас есть метод для создания векторного представления любого предложения.

Вопрос 2:

- Какая третья(с индексом 2) компонента вектора предложения `I love neural networks` (округлите до 2 знаков после запятой)? Ответ: -1.29

```
In [8]: '''your code'''
np.round(question_to_vec('I love neural networks', wv_embeddings, tokenizer.tokenize)[2
```

Out[8]: -1.29

Оценка близости текстов

Представим, что мы используем идеальные векторные представления слов. Тогда косинусное расстояние между дублирующими предложениями должно быть меньше, чем между случайно взятыми предложениями.

Сгенерируем для каждого из N вопросов R случайных отрицательных примеров и примешаем к ним также настоящие дубликаты. Для каждого вопроса будем

ранжировать с помощью нашей модели $R + 1$ примеров и смотреть на позицию дубликата. Мы хотим, чтобы дубликат был первым в ранжированном списке.

Hits@K

Первой простой метрикой будет количество корректных попаданий для какого-то K :

$$\text{Hits@K} = \frac{1}{N} \sum_{i=1}^N [\text{rank}_{q'_i} \leq K],$$

- $[x < 0] \equiv \begin{cases} 1, & x < 0 \\ 0, & x \geq 0 \end{cases}$ - индикаторная функция
- q_i - i -ый вопрос
- q'_i - его дубликат
- $\text{rank}_{q'_i}$ - позиция дубликата в ранжированном списке ближайших предложений для вопроса q_i .

DCG@K

Второй метрикой будет упрощенная DCG метрика, учитывающая порядок элементов в списке путем домножения релевантности элемента на вес равный обратному логарифму номера позиции::

$$\text{DCG@K} = \frac{1}{N} \sum_{i=1}^N \frac{1}{\log_2(1 + \text{rank}_{q'_i})} \cdot [\text{rank}_{q'_i} \leq K],$$

С такой метрикой модель штрафует за большой ранк корректного ответа

Вопрос 3:

- Максимум Hits@47 - DCG@1 ? Ответ: 1. Например если в каждом вопросе дубликат будет стоять на любом месте из отрезка $[2, 47]$, то Hits@47 будет равен 1(а больше 1 получить нельзя), а DCG@1 будет равен 0(а меньше 0 получить нельзя)



Пример оценок

Вычислим описанные выше метрики для игрушечного примера. Пусть

- $N = 1, R = 3$
- "Что такое python?" - вопрос q_1
- "Что такое язык python?" - его дубликат q'_i

Пусть модель выдала следующий ранжированный список кандидатов:

1. "Как изучить с++?"
2. "Что такое язык python?"
3. "Хочу учить Java"
4. "Не понимаю Tensorflow"

$$\Rightarrow rank_{q'_i} = 2$$

Вычислим метрику $Hits@K$ для $K = 1, 4$:

- $[K = 1] Hits@1 = [rank_{q'_i} \leq 1] = 0$
- $[K = 4] Hits@4 = [rank_{q'_i} \leq 4] = 1$

Вычислим метрику $DCG@K$ для $K = 1, 4$:

- $[K = 1] DCG@1 = \frac{1}{\log_2(1+2)} \cdot [2 \leq 1] = 0$
- $[K = 4] DCG@4 = \frac{1}{\log_2(1+2)} \cdot [2 \leq 4] = \frac{1}{\log_2 3}$

Вопрос 4:

- Вычислите $DCG@10$, если $rank_{q_i}' = 9$ (округлите до одного знака после запятой).
Ответ: 0.3

```
In [9]: np.round(1 / np.log2(1 + 9), 1)
```

```
Out[9]: 0.3
```

HITS_COUNT и DCG_SCORE

Каждая функция имеет два аргумента: *dup_ranks* и *k*. *dup_ranks* является списком, который содержит рейтинги дубликатов (их позиции в ранжированном списке).

Например, *dup_ranks* = [2] для примера, описанного выше.

```
In [10]: def hits_count(dup_ranks, k):
        """
        dup_ranks: list индексов дубликатов
        result: вернуть Hits@k
        """
        '''your code'''
        hits_value = 0
        for rank in dup_ranks:
            if rank <= k:
                hits_value += 1

        if len(dup_ranks) != 0:
            hits_value /= len(dup_ranks)

        return hits_value
```

```
In [11]: def dcg_score(dup_ranks, k):
        """
        dup_ranks: list индексов дубликатов
        result: вернуть DCG@k
        """
        '''your code'''
        dcg_value = 0
        for rank in dup_ranks:
            if rank <= k:
                dcg_value += 1 / np.log2(1 + rank)

        if len(dup_ranks) != 0:
            dcg_value /= len(dup_ranks)

        return dcg_value
```

Протестируем функции. Пусть $N = 1$, то есть один эксперимент. Будем искать копию вопроса и оценивать метрики.

```
In [12]: import pandas as pd
```

```
In [13]: copy_answers = ["How does the catch keyword determine the type of exception that was th
```

```
# наши кандидаты
candidates_ranking = ["How Can I Make These Links Rotate in PHP",
                      "How does the catch keyword determine the type of exception that",
                      "NSLog array description not memory address",
                      "PECL_HTTP not recognised php ubuntu"],]

# dup_ranks – позиции наших копий, так как эксперимент один, то этот массив длины 1
dup_ranks = [2]

# вычисляем метрику для разных k
print('Ваш ответ HIT:', [hits_count(dup_ranks, k) for k in range(1, 5)])
print('Ваш ответ DCG:', [round(dcg_score(dup_ranks, k), 5) for k in range(1, 5)])
```

Ваш ответ HIT: [0.0, 1.0, 1.0, 1.0]
 Ваш ответ DCG: [0.0, 0.63093, 0.63093, 0.63093]

У вас должно получиться

```
In [14]: # correct_answers - метрика для разных k
correct_answers = pd.DataFrame([[0, 1, 1, 1], [0, 1 / (np.log2(3)), 1 / (np.log2(3)), 1
                                index=['HITS', 'DCG'], columns=range(1,5))
correct_answers
```

```
Out[14]:
```

	1	2	3	4
HITS	0	1.00000	1.00000	1.00000
DCG	0	0.63093	0.63093	0.63093

Данные

[arxiv link](#)

`train.tsv` - выборка для обучения.

В каждой строке через табуляцию записаны: <вопрос>, <похожий вопрос>

`validation.tsv` - тестовая выборка.

В каждой строке через табуляцию записаны: <вопрос>, <похожий вопрос>, <отрицательный пример 1>, <отрицательный пример 2>, ...

```
In [15]: from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

```
In [16]: import shutil
shutil.copy('/content/drive/MyDrive/Colab Notebooks/stackoverflow_similar_questions.zip
```

```
Out[16]: '/content/stackoverflow_similar_questions.zip'
```

```
In [17]: !unzip stackoverflow_similar_questions.zip
```

Archive: stackoverflow_similar_questions.zip
 creating: data/


```
inflating: data/.DS_Store
creating: __MACOSX/
creating: __MACOSX/data/
inflating: __MACOSX/data/._.DS_Store
inflating: data/train.tsv
inflating: data/validation.tsv
```

Считайте данные.

```
In [18]: def read_corpus(filename):
          data = []
          for line in open(filename, encoding='utf-8'):
              data.append(line.split('\t'))
          return data
```

Нам понадобится только файл validation.

```
In [19]: validation_data = read_corpus('./data/validation.tsv')
```

Кол-во строк

```
In [20]: len(validation_data)
```

```
Out[20]: 3760
```

```
In [21]: validation_data[0][:10]
```

```
Out[21]: ['How to print a binary heap tree without recursion?',
          'How do you best convert a recursive function to an iterative one?',
          'How can i use ng-model with directive in angular js',
          'flash: drawing and erasing',
          'toggle react component using hide show classname',
          'Use a usercontrol from another project to current webpage',
          '~ Paths resolved differently after upgrading to ASP.NET 4',
          'Materialize datepicker - Rendering when an icon is clicked',
          'Creating PyPi package - Could not find a version that satisfies the requirement iso8601',
          'How can I analyze a confusion matrix?']
```

Размер нескольких первых строк

```
In [22]: for i in range(5):
          print(i + 1, len(validation_data[i]))
```

```
1 1001
2 1001
3 1001
4 1001
5 1001
```

Ранжирование без обучения

Реализуйте функцию ранжирования кандидатов на основе косинусного расстояния.

Функция должна по списку кандидатов вернуть отсортированный список пар (позиция

в исходном списке кандидатов, кандидат). При этом позиция кандидата в полученном списке является его рейтингом (первый - лучший). Например, если исходный список кандидатов был [a, b, c], и самый похожий на исходный вопрос среди них - c, затем a, и в конце b, то функция должна вернуть список [(2, c), (0, a), (1, b)].

```
In [23]: from sklearn.metrics.pairwise import cosine_similarity
         from copy import deepcopy
```

Обратим внимание, что для нулевого вектора, cosine_similarity выдаёт 0.

```
In [24]: print(cosine_similarity([[0, 0, 0]], [[10, 10, 20]]))
         print(cosine_similarity([[0, 0, 0]], [[0, 0, 0]]))
```

```
[[0.]]
[[0.]]
```

```
In [25]: def rank_candidates(question, candidates, embeddings, tokenizer, dim=200):
         """
             question: строка
             candidates: массив строк(кандидатов) [a, b, c]
             result: пары (начальная позиция, кандидат) [(2, c), (0, a), (1, b)]
         """
         '''your code'''

         result = [(i, x) for i, x in enumerate(candidates)]
         q_vec = question_to_vec(question, embeddings, tokenizer, dim)
         result.sort(reverse=True, key=lambda x: cosine_similarity([q_vec], [question_to_vec
         return result
```

Протестируйте работу функции на примерах ниже. Пусть $N = 2$, то есть два эксперимента

```
In [26]: questions = ['converting string to list', 'Sending array via Ajax fails']

         candidates = [['Convert Google results object (pure js) to Python object', # первый экс
             'C# create cookie from string and send it',
             'How to use jQuery AJAX for an outside domain?'],

             ['Getting all list items of an unordered list in PHP', # второй эксп
             'WPF- How to update the changes in list item of a list',
             'select2 not displaying search results']]
```

```
In [27]: for question, q_candidates in zip(questions, candidates):
         ranks = rank_candidates(question, q_candidates, ww_embeddings, tokenizer.tokeni
         print(ranks)
         print()
```

```
[(1, 'C# create cookie from string and send it'), (0, 'Convert Google results object (pu
re js) to Python object'), (2, 'How to use jQuery AJAX for an outside domain?')]
```

```
[(1, 'WPF- How to update the changes in list item of a list'), (0, 'Getting all list ite
ms of an unordered list in PHP'), (2, 'select2 not displaying search results')]
```

Для первого эксперимента вы можете полностью сравнить ваши ответы и правильные ответы. Но для второго эксперимента два ответа на кандидаты будут скрыты(*)

```
In [ ]: # должно вывести
results = [(1, 'C# create cookie from string and send it'),
           (0, 'Convert Google results object (pure js) to Python object'),
           (2, 'How to use jQuery AJAX for an outside domain?')],
           [(*, 'Getting all list items of an unordered list in PHP'), #скрыт
            (*, 'select2 not displaying search results'), #скрыт
            (*, 'WPF- How to update the changes in list item of a list')] #скрыт
### тут должен быть другой порядок
```

Последовательность начальных индексов вы должны получить для эксперимента 1 1, 0, 2.

Вопрос 5:

- Какую последовательность начальных индексов вы получили для эксперимента 2 (перечисление без запятой и пробелов, например, 102 для первого эксперимента? Ответ: 102

Теперь мы можем оценить качество нашего метода. Запустите следующие два блока кода для получения результата. Обратите внимание, что вычисление расстояния между векторами занимает некоторое время (примерно 10 минут). Можете взять для validation 1000 примеров.

```
In [29]: from tqdm.notebook import tqdm
```

```
In [30]: def validate(embeddings, tokenizer, max_validation_examples=1000):
          wv_ranking = []
          max_validation_examples = 1000
          for i, line in enumerate(tqdm(validation_data)):
              if i == max_validation_examples:
                  break
              q, *ex = line
              ranks = rank_candidates(q, ex, embeddings, tokenizer)
              wv_ranking.append([r[0] for r in ranks].index(0) + 1)

          for k in tqdm([1, 5, 10, 100, 500, 1000]):
              print("DCG@%4d: %.3f | Hits@%4d: %.3f" % (k, dcg_score(wv_ranking, k), k, hits_
```

```
In [ ]: validate(wv_embeddings, tokenizer.tokenize)
```

```
DCG@ 1: 0.285 | Hits@ 1: 0.285
DCG@ 5: 0.342 | Hits@ 5: 0.393
DCG@ 10: 0.360 | Hits@ 10: 0.449
DCG@ 100: 0.406 | Hits@ 100: 0.679
DCG@ 500: 0.431 | Hits@ 500: 0.879
DCG@1000: 0.444 | Hits@1000: 1.000
```

Эмбединги, обученные на корпусе похожих вопросов

```
In [31]: train_data = read_corpus('./data/train.tsv')
```

Улучшите качество модели.

Склеим вопросы в пары и обучим на них модель Word2Vec из gensim. Выберите размер window. Объясните свой выбор. Ответ: так как средняя длина предложения порядка 8-9 слов, то есть 16-18 слов на один список токенов, то разумно выбрать window = 2 или 3, так что размер окна составит порядка 30 % от средней длины. Потом мы проведем сравнение различных окон.

```
In [32]: from gensim.models import Word2Vec
```

Перепишем конвертер вопроса в вектор с добавлением препроцессинга слов.

Добавленная функция будет выполнять одно из следующих действий:

- Удалять стоп-слова
- Приводить к нижнему регистру
- Проводить нормализацию

Также будем пропускать вопрос, если он оказался нулевым (то есть с нима невозможно сравнивать другие вопросы).

```
In [33]: def question_to_vec(question, embeddings, tokenizer, preproc=None, preproc_params=None,

tokens = tokenizer(question)
if preproc:
    tokens = preproc(tokens, **preproc_params)
q_vec = np.zeros(dim)
count = 0
for token in tokens:
    if token in embeddings:
        q_vec += embeddings[token]
        count += 1
if count != 0:
    q_vec = q_vec / count

return q_vec
```

```
In [34]: def rank_candidates(question, candidates, embeddings, tokenizer, preproc=None, preproc_

result = [(i, x) for i, x in enumerate(candidates)]
q_vec = question_to_vec(question, embeddings, tokenizer, preproc, preproc_params, d
if (q_vec == 0).all():
    return None

result.sort(reverse=True, key=lambda x: cosine_similarity([q_vec], [question_to_vec
return result
```

```
In [35]: def validate(embeddings, tokenizer, preproc=None, preproc_params=None, max_validation_e
```

```

wv_ranking = []
max_validation_examples = 1000
for i, line in enumerate(tqdm(validation_data)):
    if i == max_validation_examples:
        break
    q, *ex = line
    ranks = rank_candidates(q, ex, embeddings, tokenizer, preproc, preproc_params, d
#вопрос не является нулевым
    if ranks:
        wv_ranking.append([r[0] for r in ranks].index(0) + 1)

print(f'len of wv_ranking={len(wv_ranking)}')
for k in tqdm([1, 5, 10, 100, 500, 1000]):
    print("DCG@%4d: %.3f | Hits@%4d: %.3f" % (k, dcg_score(wv_ranking, k), k, hits_

```

1.Изучение влияния различных методов токенизации

```

In [36]: import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt')
from nltk.tokenize import WordPunctTokenizer

```

```

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

```

1.1 Добавление/удаление стопслов

Без удаления, дефолтный токенайзер

```

In [37]: def make_words(train_data, tokenizer, preproc=None, params=None):
words = []
for line in train_data:
    # склеиваем предложения в одну строку
    q = ".".join(line)
    q_tokens = tokenizer(q)
    if preproc:
        q_tokens = preproc(q_tokens, **params)
    words.append(q_tokens)
return words

```

```

In [38]: def preproc(tokens, stopWords=None, lower=False, normalizer=None):
if stopWords:
    tokens = [token for token in tokens if token not in stopWords]

if lower:
    tokens = [token.lower() for token in tokens]

if normalizer:
    tokens = list(map(normalizer, tokens))
return tokens

```

```

In [ ]: embeddings_trained = Word2Vec(make_words(train_data, tokenizer.tokenize), size=200, min

```

```
In [ ]: validate(embeddings_trained, tokenizer.tokenize)
```

len of wv_ranking=1000

```
DCG@ 1: 0.251 | Hits@ 1: 0.251
DCG@ 5: 0.307 | Hits@ 5: 0.358
DCG@ 10: 0.336 | Hits@ 10: 0.445
DCG@ 100: 0.386 | Hits@ 100: 0.696
DCG@ 500: 0.414 | Hits@ 500: 0.913
DCG@1000: 0.423 | Hits@1000: 1.000
```

С удалением, дефолтный токенайзер

```
In [39]: from nltk.corpus import stopwords
nltk.download('stopwords')
stopWords = set(stopwords.words('english'))
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data] Unzipping corpora/stopwords.zip.
```

```
In [ ]: params = {'stopWords':stopWords, 'lower':False, 'normalizer':None}
```

```
In [ ]: embeddings_trained = Word2Vec(make_words(train_data, tokenizer.tokenize, preproc, param
```

```
In [ ]: validate(embeddings_trained, tokenizer.tokenize, preproc, params)
```

len of wv_ranking=1000

```
DCG@ 1: 0.320 | Hits@ 1: 0.320
DCG@ 5: 0.407 | Hits@ 5: 0.485
DCG@ 10: 0.434 | Hits@ 10: 0.567
DCG@ 100: 0.485 | Hits@ 100: 0.818
DCG@ 500: 0.502 | Hits@ 500: 0.954
DCG@1000: 0.507 | Hits@1000: 1.000
```

Вывод: удаление стоп-слов значительно улучшает качество. Далее будем удалять их везде.

1.2 Различные токенайзеры

Посмотрим, как работает написанный токенайзер.

```
In [ ]: tokenizer.tokenize('#I LOVE you, Australia!!>+ 3.65$ ...')
```

```
Out[ ]: ['I', 'LOVE', 'you', 'Australia', '3', '65']
```

- У нас нет приведения к нижнему регистру
- Удаляются все знаки препинания
- Удаляются все небуквенные символы

```
In [ ]: word_tokenize('#I LOVE you, Australia! 3.65$ ...')
```

```
Out[ ]: ['#', 'I', 'LOVE', 'you', ',', 'Australia', '!', '3.65', '$', '...']
```

```
In [ ]: WordPunctTokenizer().tokenize('#I LOVE you, Australia! 3.65$ ...')
```

```
Out[ ]: ['#', 'I', 'LOVE', 'you', ',', 'Australia', '!', '3', '.', '65', '$', '...']
```

Как видим, первый токенайзер грамотнее обработал число.

```
In [ ]: params = {'stopWords':stopWords, 'lower':False, 'normalizer':None}
```

```
In [ ]: embeddings_trained = Word2Vec(make_words(train_data, word_tokenize, preproc, params), s
```

```
In [ ]: validate(embeddings_trained, word_tokenize, preproc, params)
```

```
len of ww_ranking=1000
```

```
DCG@ 1: 0.272 | Hits@ 1: 0.272
DCG@ 5: 0.349 | Hits@ 5: 0.420
DCG@ 10: 0.367 | Hits@ 10: 0.476
DCG@ 100: 0.420 | Hits@ 100: 0.736
DCG@ 500: 0.446 | Hits@ 500: 0.937
DCG@1000: 0.453 | Hits@1000: 1.000
```

```
In [ ]: embeddings_trained = Word2Vec(make_words(train_data, WordPunctTokenizer().tokenize, pr
```

```
In [ ]: validate(embeddings_trained, WordPunctTokenizer().tokenize, preproc, params)
```

```
len of ww_ranking=1000
```

```
DCG@ 1: 0.298 | Hits@ 1: 0.298
DCG@ 5: 0.387 | Hits@ 5: 0.467
DCG@ 10: 0.407 | Hits@ 10: 0.530
DCG@ 100: 0.460 | Hits@ 100: 0.792
DCG@ 500: 0.481 | Hits@ 500: 0.949
DCG@1000: 0.486 | Hits@1000: 1.000
```

Вывод: по не совсем понятным причинам лучшим оказался написанный токенайзер.

Будем использовать его

1.3 Размер окна

```
In [ ]: best_tokenizer = tokenizer.tokenize

for window in [5, 7, 9, 11]:
    embeddings_trained = Word2Vec(make_words(train_data, best_tokenizer, preproc, param
    print(f'##### window = {window} #####')
    validate(embeddings_trained, best_tokenizer, preproc, params)
```

```
##### window = 5 #####
```

len of wv_ranking=1000

```
DCG@ 1: 0.334 | Hits@ 1: 0.334
DCG@ 5: 0.423 | Hits@ 5: 0.504
DCG@ 10: 0.449 | Hits@ 10: 0.586
DCG@ 100: 0.498 | Hits@ 100: 0.825
DCG@ 500: 0.515 | Hits@ 500: 0.960
DCG@1000: 0.520 | Hits@1000: 1.000
##### window = 7 #####
```

len of wv_ranking=1000

```
DCG@ 1: 0.345 | Hits@ 1: 0.345
DCG@ 5: 0.430 | Hits@ 5: 0.507
DCG@ 10: 0.458 | Hits@ 10: 0.593
DCG@ 100: 0.508 | Hits@ 100: 0.830
DCG@ 500: 0.525 | Hits@ 500: 0.964
DCG@1000: 0.529 | Hits@1000: 1.000
##### window = 9 #####
```

len of wv_ranking=1000

```
DCG@ 1: 0.354 | Hits@ 1: 0.354
DCG@ 5: 0.442 | Hits@ 5: 0.521
DCG@ 10: 0.469 | Hits@ 10: 0.607
DCG@ 100: 0.517 | Hits@ 100: 0.838
DCG@ 500: 0.533 | Hits@ 500: 0.963
DCG@1000: 0.537 | Hits@1000: 1.000
##### window = 11 #####
```

len of wv_ranking=1000

```
DCG@ 1: 0.356 | Hits@ 1: 0.356
DCG@ 5: 0.445 | Hits@ 5: 0.525
DCG@ 10: 0.471 | Hits@ 10: 0.605
DCG@ 100: 0.520 | Hits@ 100: 0.840
DCG@ 500: 0.535 | Hits@ 500: 0.960
DCG@1000: 0.540 | Hits@1000: 1.000
```

Вывод: увеличения размера окна увеличивает качество. Далее будем использовать значение 9.

1.4 min_count

In [40]:

```
best_window = 9
best_tokenizer = tokenizer.tokenize
params = {'stopWords':stopWords, 'lower':False, 'normalizer':None}

for min_count in [25, 50, 100]:
    embeddings_trained = Word2Vec(make_words(train_data, best_tokenizer, preproc, param
    print(f'##### min_count = {min_count} #####')
    validate(embeddings_trained, best_tokenizer, preproc, params)
```

```
##### min_count = 25 #####
```

len of wv_ranking=1000

```
DCG@ 1: 0.359 | Hits@ 1: 0.359
DCG@ 5: 0.444 | Hits@ 5: 0.521
DCG@ 10: 0.471 | Hits@ 10: 0.604
DCG@ 100: 0.520 | Hits@ 100: 0.842
```



```

DCG@ 500: 0.536 | Hits@ 500: 0.963
DCG@1000: 0.540 | Hits@1000: 1.000
##### min_count = 50 #####

len of wv_ranking=1000

DCG@ 1: 0.355 | Hits@ 1: 0.355
DCG@ 5: 0.438 | Hits@ 5: 0.513
DCG@ 10: 0.465 | Hits@ 10: 0.596
DCG@ 100: 0.516 | Hits@ 100: 0.839
DCG@ 500: 0.532 | Hits@ 500: 0.963
DCG@1000: 0.536 | Hits@1000: 1.000
##### min_count = 100 #####

len of wv_ranking=1000

DCG@ 1: 0.346 | Hits@ 1: 0.346
DCG@ 5: 0.437 | Hits@ 5: 0.515
DCG@ 10: 0.464 | Hits@ 10: 0.599
DCG@ 100: 0.514 | Hits@ 100: 0.842
DCG@ 500: 0.530 | Hits@ 500: 0.963
DCG@1000: 0.534 | Hits@1000: 1.000

```

```
In [41]: best_min_count = 25
```

Вывод: качество не зависит от параметра min_count в данном случае.

1.5 Приведение к нижнему регистру

```
In [43]: params = {'stopWords':stopWords, 'lower':True, 'normalizer':None}
          make_words(train_data, best_tokenizer, preproc, params)[1]
```

```
Out[43]: ['which',
          'html',
          '5',
          'canvas',
          'javascript',
          'use',
          'making',
          'interactive',
          'drawing',
          'tool',
          'event',
          'handling',
          'geometries',
          'three',
          'js']
```

```
In [44]: embeddings_trained = Word2Vec(make_words(train_data, best_tokenizer, preproc, params),
          validate(embeddings_trained, best_tokenizer, preproc, params))
```

```

len of wv_ranking=999

DCG@ 1: 0.390 | Hits@ 1: 0.390
DCG@ 5: 0.488 | Hits@ 5: 0.570
DCG@ 10: 0.510 | Hits@ 10: 0.637
DCG@ 100: 0.559 | Hits@ 100: 0.874

```

DCG@ 500: 0.572 | Hits@ 500: 0.973
DCG@1000: 0.575 | Hits@1000: 1.000

2. Нормализация слов

```
In [45]: from nltk.stem import SnowballStemmer
nltk.download('wordnet')
nltk.download('omw-1.4')
from nltk.stem import WordNetLemmatizer
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
```

```
In [46]: stem = SnowballStemmer(language='english').stem
nlp = WordNetLemmatizer().lemmatize
```

```
In [48]: for normalizer, normalizer_name in zip([stem, nlp], ["stem", "nlp"]):
    params = {'stopWords':stopWords, 'lower':True, 'normalizer':normalizer}
    embeddings_trained = Word2Vec(make_words(train_data, best_tokenizer, preproc, param
    print(f'##### normalizer = {normalizer_name} #####')
    validate(embeddings_trained, best_tokenizer, preproc, params)
```

normalizer = stem

len of wv_ranking=1000

DCG@ 1: 0.434 | Hits@ 1: 0.434
DCG@ 5: 0.518 | Hits@ 5: 0.591
DCG@ 10: 0.542 | Hits@ 10: 0.664
DCG@ 100: 0.588 | Hits@ 100: 0.884
DCG@ 500: 0.600 | Hits@ 500: 0.978
DCG@1000: 0.603 | Hits@1000: 1.000

normalizer = nlp

len of wv_ranking=999

DCG@ 1: 0.416 | Hits@ 1: 0.416
DCG@ 5: 0.509 | Hits@ 5: 0.591
DCG@ 10: 0.530 | Hits@ 10: 0.654
DCG@ 100: 0.578 | Hits@ 100: 0.884
DCG@ 500: 0.589 | Hits@ 500: 0.971
DCG@1000: 0.592 | Hits@1000: 1.000

Замечание:

Решить эту задачу с помощью обучения полноценной нейронной сети будет вам предложено, как часть задания в одной из домашних работ по теме "Диалоговые системы".

Вывод

- Какой принцип токенизации даёт качество лучше и почему? Удаление стоп-слов, приведение к нижнему регистру значительно улучшают качество, как и увеличение

размера окна. Увеличение минимальной встречаемости слов до 25 также помогло немного повысить качество, но дальнейшее увеличение данного параметра делало только хуже. Лучшим токенизатором стал собственно написанный (через регулярные выражения), а не из nltk.

- Помогает ли нормализация слов? Нормализация повысила качество. Стемминг отработал лучше, чем лемматизация
- Какие эмбединги лучше справляются с задачей и почему? Лучше справляются собственно обученные, что в целом очевидно: модель улавливает сочетания характерные для нашего датасета, а не для всех вопросов на stackoverflow
- Почему получилось плохое качество решения задачи? Для начала, не совсем понятно, какой результат нас бы удовлетворил. Но если принять, что мы хотим, что хотя бы половина запросов попадала на первую строчку, то можно предположить, что есть гораздо более значимые параметры, которые стоит подобрать, так же как и методе токенизации. Связано это с тем, что я не знаком с работой Word2Vec
- Предложите свой подход к решению задачи. Можно увеличить размер эмбединга. И тюнить параметры, описанные выше. Для других подходов стоит лучше понимать, как работает модель word2vec. Также возможно стоит посмотреть другие меры схожести помимо косинусной.

результат на предобученных эмбедингах:

DCG@ 1:	0.285		Hits@ 1:	0.285
DCG@ 5:	0.342		Hits@ 5:	0.393
DCG@ 10:	0.360		Hits@ 10:	0.449
DCG@ 100:	0.406		Hits@ 100:	0.679
DCG@ 500:	0.431		Hits@ 500:	0.879
DCG@1000:	0.444		Hits@1000:	1.000

первый результат на обученных эмбедингах:

DCG@ 1:	0.251		Hits@ 1:	0.251
DCG@ 5:	0.307		Hits@ 5:	0.358
DCG@ 10:	0.336		Hits@ 10:	0.445
DCG@ 100:	0.386		Hits@ 100:	0.696
DCG@ 500:	0.414		Hits@ 500:	0.913
DCG@1000:	0.423		Hits@1000:	1.000

лучший результат на обученных эмбедингах:

DCG@ 1:	0.416		Hits@ 1:	0.416
DCG@ 5:	0.509		Hits@ 5:	0.591
DCG@ 10:	0.530		Hits@ 10:	0.654
DCG@ 100:	0.578		Hits@ 100:	0.884
DCG@ 500:	0.589		Hits@ 500:	0.971
DCG@1000:	0.592		Hits@1000:	1.000