In [2]:
```
## Задача поиска схожих по смыслу предложений
```

Мы будем ранжировать вопросы StackOverflow (https://stackoverflow.com) на основе семантического векторного представления

До этого в курсе не было речи про задачу ранжировния, 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 $a : X \to R$ такую, что

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

Embeddings

Будем использовать предобученные векторные представления слов на постах Stack Overflow.
A word2vec model trained on Stack Overflow posts (https://github.com/vefstathiou/SO_word2vec)

In [3]:
```
# !wget https://zenodo.org/record/1199620/files/SO_vectors_200.bin?download=1
```

In [4]:
```
# !pip install gensim
```

In [5]:
```
# !pip install spacy
```

In [6]:
```
from gensim.models.keyedvectors import KeyedVectors
wv_embeddings = KeyedVectors.load_word2vec_format("SO_vectors_200.bin", binary=True)
# wv_embeddings = KeyedVectors.load_word2vec_format("SO_vectors_200.bin?download=1", binary=True)
```

Как пользоваться этими векторами?

Посмотрим на примере одного слова, что из себя представляет embedding

In [7]:
```
word = 'dogs'
if word in wv_embeddings:
    print(wv_embeddings[word])
    print(wv_embeddings[word].dtype, wv_embeddings[word].shape)
```

```
[ 0.71915245 -1.542663   -0.69895816  0.1544462  -1.2378534   0.265825
  0.65233576 -2.1376913   1.5001364  -0.16768844 -0.71720433  1.6013266
 -3.577465   -2.1195807   1.0495411   1.7895333  -1.0391432  -0.3420887
  0.31547526  1.1174009   1.2566462  -2.4243934  -0.8998013   0.10239235
  0.3956912  -0.39445704 -0.30281302 -1.8980318  -0.898579   -0.9394918
  0.15756415  1.1858691  -0.40680015 -2.57257     0.6659513  -1.3002656
  0.7096     -0.0382328  -0.95050263  1.2861551   0.33719563 -1.7006972
 -1.3042057  -0.02094089 -1.160755   -0.905893   -0.39668226  0.8157642
 -0.2383732   0.4569941   0.996064    0.5717084  -2.1317208  -0.10221571
 -1.4585396   0.54236513  0.8941682  -0.9808877   0.9992245   1.1498358
  0.34307456 -0.97934765 -1.0703176   0.13549381  1.6083974  -1.650749
 -0.9470516  -0.7484174   0.783067   -1.0349045  -1.5558331  -1.9985139
 -1.5584247  -1.9947437   1.677691    0.80027777 -0.11727657  1.0046765
 -1.5823939  -0.17658693 -0.74325824 -0.59861195  1.2277637   0.9314538
  1.7851094  -0.6622601   1.2059362   1.6233172  -2.0946274  -0.65378034
 -1.0348548  -2.9950035   0.6232046  -0.7803712  -0.02439314  0.24627402
 -0.6572641   1.6109873   1.0002007  -0.45712122 -0.9289038  -0.7851508
 -1.2042036   1.6417075  -2.062653    1.1239882   0.5475801   0.07329568
 -1.128264   -1.7790279  -0.00789989 -1.4941639   0.8983379  -1.6846293
  0.2614029  -0.0750076  -1.7032906   0.38658255  2.5906367  -0.8526129
  0.7255648  -0.5983927  -0.14658462  2.0336826   0.92287123 -1.9994345
 -1.3363256   0.79072106 -1.8597356  -0.7381024  -0.84648645 -0.9843619
  0.47617173 -1.3043061  -0.17835413 -2.854961    0.94607943 -1.231004
  1.8338903   1.3013896  -0.35322648  0.8962776   0.97410715  0.11386268
  1.1253104   0.41263318  2.7715003  -1.346934    1.204982   -0.01670979
 -0.8669396  -1.0308735  -0.5844789   0.57534117 -0.41338485 -0.3636708
 -1.7987003  -1.0684701  -1.7527993  -0.23222889  0.38671398 -0.7563939
 -2.6288023   0.31214795 -1.0111287  -0.82584405  0.6313762  -0.06947021
  0.33279362 -1.014108    0.4834512  -0.8353998  -3.05015     0.18083014
  0.9232949  -0.5345982  -1.7460634  -0.60928285  1.3242307  -0.40261996
  0.08907793 -1.6861676   1.0402287  -0.6403309   0.01863923 -0.02392901
 -0.6080688  -1.5451736  -0.75956327 -3.8619454  -0.78160954  0.17197208
 -0.0061596  -1.2189773   0.9384584  -2.4180996  -0.7229557   0.34693366
  2.0731425  -3.0019922 ]
float32 (200,)
```

In [8]:
```
# print(f"Num of words: {len(wv_embeddings.index2word)}")
print(f"Num of words: {len(wv_embeddings.index_to_key)}")
```

```
Num of words: 1787145
```

In [9]:
```
wv_embeddings.index_to_key[:10]
```

Out[9]:
```
['use', 'code', 'using', 'like', 'will', 'want', 'need', 'get', 'file', 'one']
```

Найдем наиболее близкие слова к слову `dog` :

In [10]:
```
wv_embeddings.most_similar("dog")[:5]
```

Out[10]:
```
[('animal', 0.8564180135726929),
 ('dogs', 0.7880866527557373),
 ('mammal', 0.7623804211616516),
 ('cats', 0.7621253728866577),
 ('animals', 0.760793924331665)]
```

Вопрос 1:

- Входит ли слов `cat` топ-5 близких слов к слову `dog` ? Какое место?

Ответ

- Входит слово "cats" и стоит на 4м месте, а единственное число "cat" - не входит в топ 5

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 [11]:
```
import numpy as np
import re
# from nltk.tokenize import WordPunctTokenizer
# 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 [12]:
```
a = np.array([1,2,3])
b = np.array([5,10,-3])
a, b
c = a+b
c, c/2
np.zeros(200)
```

Out[12]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [13]:
```
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
    """
    word_tokens = tokenizer.tokenize(question)
    count_word = 0
    question_embedding = np.zeros(200)
    for word in word_tokens:
        if word in embeddings:
            question_embedding += embeddings[word]
            count_word += 1
    if count_word != 0:
        question_embedding = question_embedding/count_word
    return question_embedding
```

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)?

In [14]:
```
question_embedding = question_to_vec("I love neural networks", wv_embeddings, tokenizer, dim=200)
round(question_embedding[2], 2)
```

Out[14]:
```
-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:

$$\text{Hits@K} = \frac{1}{N}\sum_{i=1}^{N}[rank\_q_i^{'} \le K],$$

- $[x < 0] \equiv \begin{cases} 1, & x < 0 \\ 0, & x \ge 0 \end{cases}$ - индикаторная функция

- q_i - i-ый вопрос
- $q_i^{'}$ - его дубликат
- $rank\_q_i^{'}$ - позиция дубликата в ранжированном списке ближайших предложений для вопроса q_i.

DCG@K

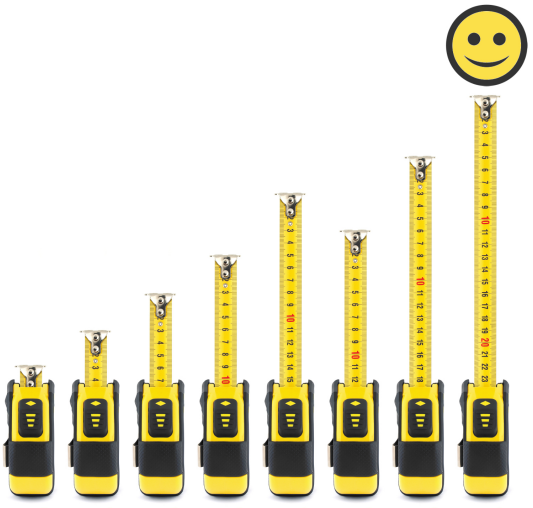
Второй метрикой будет упрощенная DCG метрика, учитывающая порядок элементов в списке путем домножения релевантности элемента на вес равный обратному логарифму номера позиции::

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

С такой метрикой модель штрафуется за большой ранк корректного ответа

Вопрос 3:

- Максимум `Hits@47 - DCG@1` ?



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 c++?"
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] $\text{Hits@1} = [rank\_q_i^{'} \leq 1)] = 0$
- [K = 4] $\text{Hits@4} = [rank\_q_i^{'} \leq 4] = 1$

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

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

Вопрос 4:

- Вычислите `DCG@10` , если $rank\_q_i^{'} = 9$(округлите до одного знака после запятой)

Ответ:

- [K = 10] $\text{DCG@10} = \frac{1}{\log_2(1+9)}\cdot[9 \leq 10] = \frac{1}{\log_2 10}$ = 0.3

In [15]:
```
import math
dcg10 = 1/math.log2(10)
print(round(dcg10,1))
```

```
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 [16]:
```
def hits_count(dup_ranks, k):
    """
        dup_ranks: list индексов дубликатов
        result: вернуть  Hits@k
    """
    hits_value = 0
    for rank in dup_ranks:
        hits_value += 1 if rank <= k else 0
    hits_value = hits_value/len(dup_ranks)
    return hits_value
```

In [17]:
```
import math
def dcg_score(dup_ranks, k):
    """
        dup_ranks: list индексов дубликатов
        result: вернуть DCG@k
    """
    dcg_value = 0
    for rank in dup_ranks:
        dcg_value += 1/math.log2(1 + rank) if rank <= k else 0
    dcg_value = 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 [18]:
```
import pandas as pd
```

In [19]:
```
copy_answers = ["How does the catch keyword determine the type of exception that was thrown",]

# наги кандидаты
candidates_ranking = [["How Can I Make These Links Rotate in PHP",
                       "How does the catch keyword determine the type of exception that was thrown",
                       "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 [20]:
```
# correct_answers - метрика для разных k
correct_answers = pd.DataFrame([[0, 1, 1, 1], [0, 1 / (np.log2(3)), 1 / (np.log2(3)), 1 / (np.log2(3))]],
                               index=['HITS', 'DCG'], columns=range(1,5))
correct_answers
```

Out[20]:

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

Данные

arxiv link (https://drive.google.com/file/d/1QqT4D0EoqJTy7v9VrNCYD-m964XZFR7_/edit)

`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 [21]:
```
# !unzip stackoverflow_similar_questions.zip
```

Считайте данные.

In [22]:
```
def read_corpus(filename):
    data = []
    for line in open(filename, encoding='utf-8'):
        data.append(line.split("\t"))
    return data
validation_data = read_corpus('./stackoverflow_similar_questions/data/validation.tsv')
```

Нам понадобиться только файл validation.

In [23]:
```
validation_data = read_corpus('./stackoverflow_similar_questions/data/validation.tsv')
```

Кол-во строк

In [24]:
```
len(validation_data)
```

Out[24]:
```
3760
```

Размер нескольких первых строк

In [25]:
```
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 [26]:
```
from sklearn.metrics.pairwise import cosine_similarity
from copy import deepcopy
```

In [27]:
```
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)]
    """
    quest_vec = question_to_vec(question, embeddings, tokenizer)
    ranks = []
    for i, candidate in enumerate(candidates):
        cand_vec = question_to_vec(candidate, embeddings, tokenizer)
        cos_sim = cosine_similarity([quest_vec], [cand_vec])[0][0]
        ranks.append((cos_sim, (i, candidate)))
    ranks = sorted(ranks,reverse=True)
    ranks = [r[1] for r in ranks]
    return ranks

# rank_candidates(questions[0], candidates[0], wv_embeddings, tokenizer)
```

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

In [28]:
```
questions = ['converting string to list', 'Sending array via Ajax fails']

candidates = [['Convert Google results object (pure js) to Python object', # первый эксперимент
               '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 эксперимент
               'WPF- How to update the changes in list item of a list',
               'select2 not displaying search results']]
```

In [29]:
```
for question, q_candidates in zip(questions, candidates):
        ranks = rank_candidates(question, q_candidates, wv_embeddings, tokenizer)
        print(ranks)
        print()
```

```
[(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?')]

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

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

In [30]:
```
# 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?')],
           [
            (1, 'WPF- How to update the changes in list item of a list'),
            (0, 'Getting all list items of an unordered list in PHP'), #скрыт
            (2, 'select2 not displaying search results'), #скрыт
            ]
          ] #скрыт
```

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?

102

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

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 [31]:
```
from tqdm.notebook import tqdm
```

In [32]:
```
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, wv_embeddings, tokenizer)
    wv_ranking.append([r[0] for r in ranks].index(0) + 1)
```

27% 1000/3760 [04:28<12:12, 3.77it/s]

In [33]:
```
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_count(wv_ranking, k)))
```

100% 6/6 [00:00<00:00, 333.38it/s]

```
DCG@   1: 0.223 | Hits@   1: 0.223
DCG@   5: 0.282 | Hits@   5: 0.335
DCG@  10: 0.301 | Hits@  10: 0.392
DCG@ 100: 0.347 | Hits@ 100: 0.622
DCG@ 500: 0.372 | Hits@ 500: 0.821
DCG@1000: 0.391 | Hits@1000: 1.000
```

Эмбеддинги, обученные на корпусе похожих вопросов

Улучшите качество модели.
Склеим вопросы в пары и обучим на них модель Word2Vec из gensim. Выберите размер window. Объясните свой выбор.

In [34]:
```
train_data = read_corpus('./stackoverflow_similar_questions/data/train.tsv')
```

In [35]:
```
# train_words = [tokenizer.tokenize(" ".join(w)) for w in train_data[:1000]]
train_words = [tokenizer.tokenize(" ".join(w)) for w in train_data]
# train_data[0], train_words
# len(train_words)

# определеям среднее кол-во слов в вопросе
print(f"среднее кол-во слов в вопросе: {np.array(list(map(len, train_words))).mean()}")
```

```
среднее кол-во слов в вопросе: 19.699199
```

In [36]:
```
# min_count = int – игнорирует все слова у которых частота ниже заданной (2, 100)
# window = int - размер контекстного окна -  максимальное расстояние между текущим и прогнозируемым словом в предлож
# параметр «window» должен быть достаточно большим, чтобы фиксировать синтаксические/семантические отношения. Как пр

# size – размер векторного представления слова (word embedding).
# negative – сколько неконтекстных слов учитывать в обучении, используя negative sampling.
# alpha – начальный learning_rate, используемый в алгоритме обратного распространения ошибки (Backpropogation).
# min_alpha – минимальное значение learning_rate, на которое может опуститься в процессе обучения.
# sg – если 1, то используется реализация Skip-gram; если 0, то CBOW.
```

In [37]:
```
from gensim.models import Word2Vec
embeddings_trained = Word2Vec(train_words, # data for model to train on
                 vector_size=200,                 # embedding vector size
                 min_count=5,             # consider words that occured at least 5 times
                 window=5).wv
# Параметр «window»:
# - определяет размер контекстного окна -  максимальное расстояние между текущим и прогнозируемым словом в предложен
# - должен быть достаточно большим, чтобы фиксировать синтаксические/семантические отношения. Как правило для англи
# - зависит от исходных данных, после склейки вопросов средняя длина предложений составила 20 слов (19.699).Так как
# Таким образом параметр «window» окно стоит рассматривать равным 5
# Однако, из-за особенностей данных (технические вопросы), есть смысл произвести дополнительную проверку на окне ме
```

In [38]:
```
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_trained, tokenizer)
    wv_ranking.append([r[0] for r in ranks].index(0) + 1)
```

```
27%                                    1000/3760 [04:41<12:52, 3.57it/s]
```

In [39]:
```
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_count(wv_ranking, k)))
```

```
100%                                    6/6 [00:00<00:00, 272.73it/s]

DCG@   1: 0.258 | Hits@   1: 0.258
DCG@   5: 0.326 | Hits@   5: 0.388
DCG@  10: 0.352 | Hits@  10: 0.468
DCG@ 100: 0.404 | Hits@ 100: 0.724
DCG@ 500: 0.430 | Hits@ 500: 0.926
DCG@1000: 0.438 | Hits@1000: 1.000
```

In [40]:
```
from nltk import SnowballStemmer
from nltk.corpus import stopwords


# Правим токенизацию так чтобы числа убрать, были только слова или слова с цифрами ( можно через метод isalpha)
class MyTokenizer2:
    def __init__(self):
        # Stemmer (Стеминг)
        self.stemmer = SnowballStemmer(language="english")
        # Stop words
        self.eng_stopwords = stopwords.words("english")

    def tokenize(self, text):
        # переводим в нижний регистр
        text = text.lower()
        find_words = re.findall('\w+', text)
        # Производим Стеминг (Stemmer)
        find_words = [self.stemmer.stem(w) for w in find_words if w.isalpha()]
        # убираем стоп-слова
        find_words = [w for w in find_words if w not in self.eng_stopwords]
        return find_words


tokenizer2 = MyTokenizer2()
# tokenizer2.tokenize(train_data[1][0].lower())
```

In [41]:
```
train_words = [tokenizer2.tokenize(" ".join(w)) for w in train_data]
```

Обучаем модель с параметром "window=5"

In [43]:
```
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_trained, tokenizer2)
    wv_ranking.append([r[0] for r in ranks].index(0) + 1)
```

In [44]:
```
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_count(wv_ranking, k)))
```

```
DCG@   1: 0.405 | Hits@   1: 0.405
DCG@   5: 0.505 | Hits@   5: 0.592
DCG@  10: 0.529 | Hits@  10: 0.667
DCG@ 100: 0.576 | Hits@ 100: 0.892
DCG@ 500: 0.588 | Hits@ 500: 0.977
DCG@1000: 0.590 | Hits@1000: 1.000
```

Обучаем модель с параметром "window=10"

In [42]:
```
from gensim.models import Word2Vec
embeddings_trained = Word2Vec(train_words, # data for model to train on
                 vector_size=200,                # embedding vector size
                 min_count=5,            # consider words that occured at least 5 times
                 window=5).wv
```

In [45]:
```
from gensim.models import Word2Vec
embeddings_trained = Word2Vec(train_words, # data for model to train on
                 vector_size=200,                # embedding vector size
                 min_count=5,            # consider words that occured at least 5 times
                 window=10).wv
```

In [46]:
```
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_trained, tokenizer2)
    wv_ranking.append([r[0] for r in ranks].index(0) + 1)
```

27% 1000/3760 [07:15<19:57, 2.30it/s]

In [47]:
```
# window 10
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_count(wv_ranking, k)))
```

```
100%                                                6/6 [00:00<00:00, 239.94it/s]

DCG@   1: 0.412 | Hits@   1: 0.412
DCG@   5: 0.515 | Hits@   5: 0.601
DCG@  10: 0.540 | Hits@  10: 0.680
DCG@ 100: 0.586 | Hits@ 100: 0.900
DCG@ 500: 0.596 | Hits@ 500: 0.978
DCG@1000: 0.599 | Hits@1000: 1.000
```

In [ ]:
```

```

In [48]:
```
# embeddings_trained.most_similar(['machine', 'learning'])
# embeddings_trained.most_similar(['machin', 'learn'])
# stemmer = SnowballStemmer(language="english")
# stemmer.stem("Learning")
```

In [49]:
```
from nltk.corpus import stopwords
import spacy

# Правим токенизацию так чтобы числа убрать, были только слова или слова с цифрами ( можно через метод isalpha)
class MyTokenizer3:
    def __init__(self):
        # Леммматизация
        self.nlp = spacy.load("en_core_web_sm")
        # Stop words
        self.eng_stopwords = stopwords.words("english")

    def tokenize(self, text):
        # переводим в нижний регистр
        doc = self.nlp(text.lower(), disable=["tok2vec", "parser", "ner" ,"textcat", "custom"])
        find_words = [w.lemma_ for w in doc]
        # убираем стоп-слова
        find_words = [w for w in find_words if w.isalpha() and w not in self.eng_stopwords]



        return find_words


tokenizer3 = MyTokenizer3()
```

In [50]:
```
train_words = [tokenizer3.tokenize(" ".join(w)) for w in train_data]
```

In [51]:
```
from gensim.models import Word2Vec
embeddings_trained = Word2Vec(train_words, # data for model to train on
                 vector_size=200,                 # embedding vector size
                 min_count=5,             # consider words that occured at least 5 times
                 window=10, sg=1).wv
```

In [52]:
```
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_trained, tokenizer3)
    wv_ranking.append([r[0] for r in ranks].index(0) + 1)
```

```
27%                                                1000/3760 [13:51<37:52, 1.21it/s]
```

In [53]:
```
# window 10
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_count(wv_ranking, k)))
```

```
100%                                                6/6 [00:00<00:00, 250.01it/s]

DCG@   1: 0.398 | Hits@   1: 0.398
DCG@   5: 0.494 | Hits@   5: 0.574
DCG@  10: 0.520 | Hits@  10: 0.654
DCG@ 100: 0.567 | Hits@ 100: 0.880
DCG@ 500: 0.579 | Hits@ 500: 0.969
DCG@1000: 0.582 | Hits@1000: 1.000
```

In [ ]:
```

```

In [54]:
```
from gensim.models import Word2Vec
embeddings_trained = Word2Vec(train_words, # data for model to train on
                 vector_size=200,                # embedding vector size
                 min_count=5,            # consider words that occured at least 5 times
                 window=10, sg=1).wv
```

In [55]:
```
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_trained, tokenizer3)
    wv_ranking.append([r[0] for r in ranks].index(0) + 1)
```

```
27%                                          1000/3760 [14:37<40:19, 1.14it/s]
```

In [56]:
```
# window 10
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_count(wv_ranking, k)))
```

```
100%                                         6/6 [00:00<00:00, 166.44it/s]

DCG@   1: 0.491 | Hits@   1: 0.491
DCG@   5: 0.585 | Hits@   5: 0.663
DCG@  10: 0.603 | Hits@  10: 0.719
DCG@ 100: 0.639 | Hits@ 100: 0.890
DCG@ 500: 0.650 | Hits@ 500: 0.971
DCG@1000: 0.653 | Hits@1000: 1.000
```

In [ ]:
```

```

Предварительный результат:

Выбор параметр размер контекстного окна («window»):

- Определяет размер контекстного окна - максимальное расстояние между текущим и прогнозируемым словом в предложении.
- Должен быть достаточно большим, чтобы фиксировать синтаксические/семантические отношения. Как правило для английского языка принято значение по умолчанию равное 5 (Значение по умолчанию https://radimrehurek.com/gensim/models/word2vec.html (https://radimrehurek.com/gensim/models/word2vec.html))
- Также выбор размера окна зависит от исходных данных, например, после склейки вопросов средняя длина предложений составила 20 слов (19.994), однако также надо учитывать исключение стоп-слов. Так как склеивались 2 вопроса, то под средней длиной вопроса можно взять 10 слов. Тогда окно должно быть равным 5
- Общий подход при выборе размера окна: окно большего размера, как правило, содержат больше информации о тематике/области, т.е. сильно важна контекстная связь с соседними словами. Окна меньшего размера, как правило, больше отражают особенности самого слова (напрмер фразы или синонимы).
- Необходимо учитывать специфику текста ,в нашем случае это вопросы, как правило вопросы задаются в минималистическом стиле, т.е. таким образом, что все слова в вопросе играют роль. В таком случае имеет смысл рассматривать увеличенный размер окна.
- Таким образом размер окна (параметр «window») стоит рассматривать не менее чем 5 и не более чем 10.

Использование предрасчитанной модели показало гораздно лучшие результаты чем самостоятельное обучение без использования дополнительной предобработки текстов:

Использование предобученной модели:

DCG@ 1: 0.223 | Hits@ 1: 0.223
DCG@ 5: 0.282 | Hits@ 5: 0.335
DCG@ 10: 0.301 | Hits@ 10: 0.392
DCG@ 100: 0.347 | Hits@ 100: 0.622
DCG@ 500: 0.372 | Hits@ 500: 0.821
DCG@1000: 0.391 | Hits@1000: 1.000

Использование собственно обученной модели на всех примерах примерах (размер окна=5):

DCG@ 1: 0.258 | Hits@ 1: 0.258
DCG@ 5: 0.326 | Hits@ 5: 0.388
DCG@ 10: 0.352 | Hits@ 10: 0.468
DCG@ 100: 0.404 | Hits@ 100: 0.724
DCG@ 500: 0.430 | Hits@ 500: 0.926
DCG@1000: 0.438 | Hits@1000: 1.000

Использование собственно обученной модели на всех примерах (размер окна=5) и проведена предобработка текста (стоп-слова стемминг и пр.):

DCG@ 1: 0.405 | Hits@ 1: 0.405
DCG@ 5: 0.505 | Hits@ 5: 0.592
DCG@ 10: 0.529 | Hits@ 10: 0.667

DCG@ 100: 0.576 | Hits@ 100: 0.892
DCG@ 500: 0.588 | Hits@ 500: 0.977
DCG@1000: 0.590 | Hits@1000: 1.000

Проверка на других размерах параметра контекстного окна: Размер окна=10
DCG@ 1: 0.412 | Hits@ 1: 0.412
DCG@ 5: 0.515 | Hits@ 5: 0.601
DCG@ 10: 0.540 | Hits@ 10: 0.680
DCG@ 100: 0.586 | Hits@ 100: 0.900
DCG@ 500: 0.596 | Hits@ 500: 0.978
DCG@1000: 0.599 | Hits@1000: 1.000

Использование лемматизация (маленький словарь: "en_core_web_sm") вместо стемминга. Размер окна=10

DCG@ 1: 0.398 | Hits@ 1: 0.398
DCG@ 5: 0.494 | Hits@ 5: 0.574
DCG@ 10: 0.520 | Hits@ 10: 0.654
DCG@ 100: 0.567 | Hits@ 100: 0.880
DCG@ 500: 0.579 | Hits@ 500: 0.969
DCG@1000: 0.582 | Hits@1000: 1.000

Использование лемматизация (маленький словарь: "en_core_web_sm") вместо стемминга. Размер окна=10 + добавлен параметр skip-gram (https://towardsdatascience.com/word2vec-skip-gram-model-part-1-intuition-78614e4d6e0b (https://towardsdatascience.com/word2vec-skip-gram-model-part-1-intuition-78614e4d6e0b))

DCG@ 1: 0.491 | Hits@ 1: 0.491
DCG@ 5: 0.585 | Hits@ 5: 0.663
DCG@ 10: 0.603 | Hits@ 10: 0.719
DCG@ 100: 0.639 | Hits@ 100: 0.890
DCG@ 500: 0.650 | Hits@ 500: 0.971
DCG@1000: 0.653 | Hits@1000: 1.000

Использование собственно обученной модели с использованием предобработки текстов показало лучше результаты чем использование предрасчитанной модели. Предобработка текста сильно улучшила показатели. Использовалась следующая предобработка: исключение стоп-слова, лемматизацию/стемминг, исключение пунктуации и только чисел. Также значимо повлиял выбор гиперпараметров, в том числе размер окна.

Лемматизация довольно затратный по ресурсам процесс, поэтому эксперимент проводился только с использованием маленького словаря: "en_core_web_sm". Примечательно, что Стемминг показал результат лучше чем Лемматизация с маленьким словарем.

Алгоритм Skip-gram довольно хорошо показал себя в качестве улучшения модели.

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".

Напишите свой вывод о полученных результатах.

- Какой принцип токенизации даёт качество лучше и почему?
- Помогает ли нормализация слов?
- Какие эмбеддинги лучше справляются с задачей и почему?
- Почему получилось плохое качество решения задачи?
- Предложите свой подход к решению задачи.

Вывод:

По итогам проведенных тестов выявлены следующие моменты:

1. При построении языковой модели необходимо учитывать природу и особеннности входных данных. Так например особенностью вопросов на stackoverflow является высокий уровень технических терминов и узкая тематика, а также стиль написания заголовков вопросов - в боьлшинстве случаев кратко по существу. Поэтому предобученная языковая модель на вопросах stackoverflow работала хуже чем обученная с нуля но на обучающей выборки данных именно stackoverflow.
2. При выборе приниципа токенизации необходимо учитывать проводимую предобработку данных. Одни методы предобработки имеет смысл производить до токенизации (например приведение слов текста в нижний регистр), а другие после токенизации (например, отбрасывание стоп-слов).
3. Нормализация слов является одним из основных методов предобработки текста. Если сравнивать Стемминг и Лемматизацию, то Лемматизация довольно затратный по ресурсам процесс, поэтому необходимо обосновано выбирать метод, в зависимости от доступных ресурсов и требуемой точности. При этом Лемматизация на маленьком словаре может проигрывать Стеммингу. Для текущей задачи Стемминг показал хорошие результаты. Дополнительно Стемминг применялся к словарю стоп-слов, чтобы обеспечить исключение слов после Стемминга. П.С. При нормализации текста (слов) важно помнить, что нормализовывать надо не только обучающую выборку, но и тестовые/валидационные данные с которыми будет работать обученная модель.
4. Использование эмбеддингов, построенных через Word2Vec библиотеки Gensim, показали гораздо лучшие результаты чем использование предобученных эмбендингов, вероятно это связано со специфичностью вопросов stackoverflow. Т.е. не всегда использование больших обученных моделей может показать хорошие результаты, иногда и небольшие модели но обученные на похожих данных показывают результаты гораздо лучше.
5. Алгоритм Skip-gram довольно хорошо показал себя в качестве улучшения модели.

6. Тексты вопросов содержит много технических терминов и сокращений, что необходмио учитывать.
7. В качестве развития можно подумать о следующих направлениях:

- использовании словаря технических синонимов, например JS = JavaScript и т.д.
- применить алгоритм CBOW (Continuous Bag of Words)
- использовать grid search для поиска гиперпараметров и т.д.