KAIST AI605 Assignment 2: Retrieval

TA in charge: Miyoung Ko (miyoungko@kaist.ac.kr)

Due Date: Apr 26 (Tue) 11:00pm, 2022

Your Submission

If you are a KAIST student, you will submit your assignment via KLMS. If you are a NAVER student, you will submit via Google Form.

You need to submit both (1) a PDF of this notebook, and (2) a link to CoLab for execution (.ipynb file is also allowed).

Use in-line LaTeX (see below) for mathematical expressions. Collaboration among students is allowed but it is not a group assignment so make sure your answer and code are your own. Make sure to mention your collaborators in your assignment with their names and their student ids.

Grading

The entire assignment is out of 20 points. You can obtain up to 2 bonus points (i.e. max score is 22 points). For every late day, your grade will be deducted by 2 points (KAIST students only). You can use one of your no-penalty late days (7 days in total). Make sure to mention this in your submission. You will receive a grade of zero if you submit after 7 days.

Environment

You will need Python 3.7+ and PyTorch 1.9+, which are already available on Colab:

==========

Collab link https://colab.research.google.com/drive/1F_9J_6aBjpL7CqZxIjXYqEnw7CRcXqXI?usp=sharing ==========

```
1 from platform import python_version
2 import torch
3
4 print("python", python_version())
5 print("torch", torch.__version__)
```

```
python 3.7.13
torch 1.11.0+cu113
```

You will use SQuAD, a classic machine reading comprehension dataset, in this assignment. Note that while this is an MRC dataset, we will also use it for retrieval by trying to find the correct document corresponding to the question among all the documents in the **validation** data

```
!pip install -q datasets
```

```
from datasets import load_dataset
from pprint import pprint


squad_dataset = load_dataset('squad')
pprint(squad_dataset['train'][0]) # 'context' contains the document
```

```
Reusing dataset squad (/root/.cache/huggingface/datasets/squad/plain_text/1.0.0/d6ec
100%                                                2/2 [00:00<00:00, 35.98it/s]
{'answers': {'answer_start': [515], 'text': ['Saint Bernadette Soubirous']},
 'context': 'Architecturally, the school has a Catholic character. Atop the '
            "Main Building's gold dome is a golden statue of the Virgin Mary. "
            'Immediately in front of the Main Building and facing it, is a '
            'copper statue of Christ with arms upraised with the legend '
            '"Venite Ad Me Omnes". Next to the Main Building is the Basilica '
            'of the Sacred Heart. Immediately behind the basilica is the '
            'Grotto, a Marian place of prayer and reflection. It is a replica '
            'of the grotto at Lourdes, France where the Virgin Mary reputedly '
            'appeared to Saint Bernadette Soubirous in 1858. At the end of the '
            'main drive (and in a direct line that connects through 3 statues '
            'and the Gold Dome), is a simple, modern stone statue of Mary.',
 'id': '5733be284776f41900661182',
 'question': 'To whom did the Virgin Mary allegedly appear in 1858 in Lourdes '
             'France?',
```

```
from scipy.sparse import csr_matrix
from tqdm import tqdm
from collections import Counter
from collections import defaultdict
import re
```

0. Processing of the database:

```
context  = squad_dataset['validation']['context'] # 'context' contains the docume
questions= squad_dataset['validation']['question']
print(len(context), len(questions))
context_train  = squad_dataset['train']['context']
question_train  = squad_dataset['train']['question']
print(len(context_train))
```

```
10570 10570
87599
```

```
def process(context, questions):
```

```
def process(context, questions):
    """documesnt is a list of sentences"""
    index     = -1
    documents = []
    questions_to_context = []
    senteces_dict        = {}
    # zip make a tuple with context and question
    for sentence, question in zip(context, questions):
        if sentence in senteces_dict:
            sentence_id = senteces_dict[sentence]
        else:
            # create the index to kink each sentece to a unique id
            index += 1
            sentence_id = index # new sentence with the new indx
            senteces_dict[sentence] = index
            # construct the list of documents with all the new sentences
            documents.append(sentence)
        # ling each question to the index of their answer in a matraix [[ id , qu
        questions_to_context.append([sentence_id, question])
    return documents, questions_to_context

```

```
light_context, light_qestions = process(context, questions)
print(len(light_qestions))
print(len(light_context))
light_context_train, light_qestion_train = process(context_train, question_train)
```

```
10570
2067
```

1. Measuring Similarity

We discussed in Lecture 04 that there are several ways to measure similarity between two vectors, such as L2 (Euclidean) distance, L1 (Manhattan) distance, inner product, and cosine distance. Here, only L1 and L2 (and angular distance) are *metric* (see *Definition* at https://en.wikipedia.org/wiki/Metric_(mathematics)).

A map is a metric or distance function if : on a set X is a function (called distance function or simply distance) $d : X \times Y \to \mathbb{R}$, such that for all $x, y, z \in X$, the following three axioms hold:

$$\begin{aligned} &1. d(x,y) = 0 \iff x = y && \text{identity of indiscernibles} \\ &2. d(x,y) \quad = \quad d(y,x) && \text{symmetry} \\ &3. d(x,z) \quad \leq \quad d(x,y) + d(y,z) && \text{triangle inequality} \end{aligned}$$

> **Problem 1.1** *(2 points)* Using the definition of metric above, prove that L1 distance is a metric.

We define the $\mathcal{L}_1$ as :

$$d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$$
$$(X, Y) \to \sum_{i=0}^{n} |x_i - y_i|$$

Symmetry:

Obvious thanks to the symmetry of the absolute value : $|x_i - y_i| = |y_i - x_i|$

Triangle inequality :

We have : $\forall (x, y) \in \mathbb{C}^2 |x + y| \leq |x| + |y|$, so if we take with a $i \in [[1, n]]$, $x = x_i - z_i$ and $y = z_i - y_i$, we got :

$\forall i \in [[1, n]] \text{ and } \forall (x_i, y_i, z_i) \in \mathbb{R}^3$

$$|x_i - z_i + z_i - y_i| \leq |x_i - z_i| + |y_i - z_i|$$

Hence : $\forall i \in [[1, n]] \text{ and } \forall (x_i, y_i, z_i) \in \mathbb{R}^3$

$$|x_i - y_i| \leq |x_i - z_i| + |y_i - z_i|$$

So as $f : x \in \mathbb{R}_*^+ \to \sum x_i$ is strictly growing for all $x_i \geq 0$ we got :

$$\sum_{i=1}^{n} |x_i - y_i| \leq \sum_{i=1}^{n} (|x_i - z_i| + |y_i - z_i|)$$

Hence :

$$d(x, y) \leq d(x, z) + d(z, y)$$

Identity of indiscernibles:

$$\begin{aligned} d(x, y) = 0 &\iff \sum_{i=1}^{n} |x_i - y_i| = 0 \\ &\iff |x_i - y_i| = 0, \forall i \in [[1, n]] \\ &\iff x_i = y_i, \forall i \in [[1, n]] \\ &\iff x = y \end{aligned}$$

Because $f : x \in \mathbb{R}_*^+ \to \sum x_i$ is strictly growing for all $x_i \geq 0$. So if a fonction strictly growing is null, that means all its elements are null.

> **Problem 1.2** *(2 points)* Prove that negative inner product is NOT a metric.

The negative inner product, the canonical inner product of $\mathbb{R}^n$ multiplied by -1:

$$\langle \mathbf{x}, \mathbf{y} \rangle = -(\sum_{i=1}^{n} x_i y_i).$$

If the negative inner product do not respect one of the three stated properties it cannot be a metric.

Lets d be the negative inner product, $d : (x, y) \rightarrow -(\sum_{i=1}^{n} x_i y_i)$

Lets take : x = (1, 1) , y = (1, 1) and z = (1, 1) Hence : d(x,y) = -2 , d(z,y) = -2 and d(z,x) = -2

So $d(x, z) + d(z, y) = -4 \leq -2 = d(x, y)$, hence, the triangle inequality is not respected.

The negative inner product is not a metric

> **Problem 1.3** *(2 points)* Prove that cosine distance (1 - cosine similarity) is NOT a metric.

Cosine similarity is defined like : $S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},.$

Lets define

$$d : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$$
$$(x, y) \rightarrow 1 - S_C(x, y)$$

We use the same methode as in problem 1.2 : Lets take : x = (1, 0) , y = (0, 1) like that :

$$S_c(x, y) = \frac{0 \times 1 + 0 \times 1}{(0 + 1)(0 + 1)} = 0$$

Hence : $d(x, y) = 1$ We take z = (a, b)

$$d(x, z) = 1 - \frac{1 \times a + 0 \times b}{\sqrt{(a^2 + b^2)}} = 1 - \frac{a}{\sqrt{(a^2 + b^2)}}$$
$$d(y, z) = 1 - \frac{1 \times b + 0 \times a}{\sqrt{(a^2 + b^2)}} = 1 - \frac{b}{\sqrt{(a^2 + b^2)}}$$
$$d(x, z) + d(y, z) = 2 - \frac{a + b}{\sqrt{(a^2 + b^2)}}$$

Now we have to find a and b such has :

$$d(x, z) + d(y, z) < 1 \iff 1 < \frac{a + b}{\sqrt{(a^2 + b^2)}}$$

Wich is clearly true for $\forall (a, b) \in \mathbb{R}_+^*$

Hence if x = (1, 0) , y = (0, 1) and z = (2, 3) :

$$d(x, z) + d(y, z) = 2 - \frac{5}{\sqrt{13}} < 1 = d(x, y)$$

hence, the triangle inequality is not respected.

The cosine distance is not a metric

> **Problem 1.4 (bonus)** *(2 points)* Given a model that can perform nearest neighbor search in L2 space, can you modify your query and your key vectors to perform maximum inner product search? (Hint: Recall the difference between MIPS and L2

> NNS in Lecture 04. Can you modify key vectors so that the difference becomes 0?)

2. Sparse Search

We first create an abstract class for performing similarity search as follows (`raise NotImplementedError()` means you have to override these methods when you subclass the class):

```
class SimilaritySearch(object):
    def __init__(self):
        self.is_trained = False

    def train(self, documents: list):
        raise NotImplementedError()

  #Add documents (a list of text)
    def add(self, documents: list):
        raise NotImplementedError()

  #Returns the indices of top-k documents among the added documents
  #that are most similar to the input query
    def search(self, query: str, k: int) -> list:
        raise NotImplementedError()

```

You will use the same space-based tokenizer that you used in Assignment 1, with lowercasing to make it case insensitive.

```
def tokenizer(sentence: str):
    return (sentence.lower().split(' '))
```

```
def unique_tokenizer(sentence):
    """this tokenizer elininates the occurances of a word"""
    return (list(set(sentence.lower().split(' '))))
```

```
def document_tokenizer(document: list):
    """document is a list of sentences"""
    words = []
    for i in range(len(document)):
        for j in range(len(tokenizer(document[i]))):
            words.append(tokenizer(document[i])[j])
    return(words)
```

```
def documents_tokenizer(documents):
    """document is a list of ducuments"""
    words = []
    for document in documents:
        words = words + document_tokenizer(document)
    return (list(set(words)))
```

```
def documents_tokenizer_not_unique(documents):
    """document is a list of ducuments"""
    words = []
    for document in documents:
        words = words + document_tokenizer(document)
    return (list(words))
```

> **Problem 2.1** *(2 points)* We will first start with Bag of Words that we discussed in Lecture 08. Using the definition in the class (don't worry about the exact definition though), implement `BagOfWords` class that subclasses `SimilaritySearch` class.

```
class BagOfWords (SimilaritySearch):
    def __init__(self):
        super(BagOfWords, self).__init__()
        self.document = []
        self.word_to_index = {}
        self.words  = []
        self.matrix = None
        self.vocab  = defaultdict(int)

    def normed(self, vector: torch.tensor) -> torch.tensor:
        norm_vector = vector / torch.norm(vector)
        return norm_vector

    def matrix_bag(self, document: list)-> torch.tensor:
        """ document is a list of string
            return an array with is the bag of word matrix"""
        rows    = []
        columns = []
        values  = []
        new_values =[]
        if isinstance(document, (list,)):
            for idx, row in enumerate((document)): # for each document in the dat
                # we put each sentences (row) in lower case and split it on space
                a = row.lower().split(" ")
                for i in range (len(a)):
                    # to remove all the non alpha caracter before adding them to
                    a[i]= re.sub(r'[^a-zA-Z ]', '', a[i])

                #Counter fonction gives us the number of occurences of each words
```

```
                word_freq = dict(Counter(a))

                for word, freq in word_freq.items():
                # for each unique word in the review.

                    # we will check if its there in the vocabulary that we build
                    # dict.get() function will return the values, if the key does
                    col_index = self.word_to_index.get(word, -1) # retreving the

                    # if the word exists
                    if col_index !=-1:
                        # we are storing the index of the document
                        rows.append(idx)
                        # we are storing the dimensions of the word
                        columns.append(col_index)
                        # we are storing the frequency of the word
                        values.append(freq)

            emb_matrix =(csr_matrix((values, (rows,columns)), dtype=float,
                                    shape=( len(document),len(self.word_to_index)
            emb_matrix = torch.tensor(emb_matrix) # we convert the array to a ten
            emb_matrix_normed= torch.stack([self.normed((emb_matrix[row,:]).clone
                                 for row in range(emb_matrix.size()[0])]) # we nor
                                                                          # a norme
            emb_matrix_sparce = emb_matrix_normed.to_sparse() # we sparce the mat
            return  emb_matrix_sparce
        else:
            print("you need to pass list of strings")


    def train(self, documents: list) -> None:
        """generate the word_to_index dictionnary"""
        self.matrix  = None

        # check if the input is a list type or not
        if isinstance(documents, (list)):

            # counstruction of the vocab and word_to_index dictionnary
            for document in documents[0]:
                for word in unique_tokenizer(document):
                    # we remove all the non ASCII characters (, ; . ! ? ... ect)
                    word =(re.sub(r'[^a-zA-Z ]', '', word))
                    self.vocab[word] += 1

            for index,word in enumerate (list(self.vocab.keys())):
                if word not in self.word_to_index.keys():
                    word =(re.sub(r'[^a-zA-Z ]', '', word))
                    self.word_to_index [word] = index

            self.is_trained = True
        else:
            print("You need to pass list of sentance")
```

```
            self.is_trained = False


    def add(self, documents: list) -> None:
        a= []
        if(self.is_trained == True):
            for document in documents:
                m   = self.matrix_bag(document).to_dense()
                elt = list(torch.tensor_split(m , m.size()[0], dim= 0))
                for i in range(len (elt)):
                    elt[i] = elt[i].squeeze()
                a = a+elt

            sub_matrix = torch.stack(a)

            if self.matrix  == None: # the self.matrix has not been contructed ye
                self.matrix = sub_matrix
            else :
                self.matrix = torch.cat((self.matrix, sub_matrix))

            print('Embedded matrix : ', '\n', self.matrix, '\n')
        else:
            print ("The vocad is not done yet, use the train method to generate t

    def search(self, question: str, k: int, pt=True) -> list:
        # as matrix_bag return a sparce matrix we use .to_dense()
        question_matrix = self.matrix_bag([question]).to_dense()
        similarity_vector=torch.zeros (self.matrix.size()[0])

        for row in range(self.matrix.size()[0]):
            # we do the scalr product between query and each row
            # to find the closser row of query, eg the one with the greater
            # scalar product
            similarity_vector [row] = self.matrix[row] @ question_matrix.T

        if (pt):
            print("score vector", similarity_vector)

        # topk(int k) return the k greater values of a tensor and their index
        index_k_max_values = similarity_vector.squeeze().topk(k)

        return (index_k_max_values[1])

    def getWordToIndex(self) -> None:
        print(self.word_to_index)

    def getMatrix(self)-> None:
        return self.matrix
```

```
# Small test

documents = [["I am clément",
```

```
4               "I am a student",
5               "i am a the cat",'I am a Clément a cat and a cat']]
6
7 BOW = BagOfWords()
8 BOW.train(documents)
9 BOW.getWordToIndex()
10 BOW.add(documents)
11
12 BOW.search("I am Clément and cat", 3)
13
```

```
{'am': 0, 'clment': 1, 'i': 2, 'a': 3, 'student': 4, 'cat': 5, 'the': 6, 'and': 7}
Embedded matrix :
 tensor([[0.5774, 0.5774, 0.5774, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5000, 0.0000, 0.5000, 0.5000, 0.5000, 0.0000, 0.0000, 0.0000],
        [0.4472, 0.0000, 0.4472, 0.4472, 0.0000, 0.4472, 0.4472, 0.0000],
        [0.2425, 0.2425, 0.2425, 0.7276, 0.0000, 0.4851, 0.0000, 0.2425]],
       dtype=torch.float64)

score vector tensor([0.7746, 0.4472, 0.6000, 0.6508])
tensor([0, 3, 2])
```

```
1 # Big test
2
3 bow = BagOfWords()
4 bow.train([light_context]) # we train it on light_context and test it on light qu
5 bow.add([light_context])
6
7 # test 0 has to be in the list of top_k index
8 topk_idx = bow.search(light_context[0], 10)
9 print("top K index of dcuments", topk_idx)
```

```
Embedded matrix :
 tensor([[0.0476, 0.0476, 0.0476,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0556, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        ...,
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0546, 0.0546, 0.0546]],
       dtype=torch.float64)

score vector tensor([1.0000, 0.7214, 0.3138,  ..., 0.3184, 0.3838, 0.5060])
top K index of dcuments tensor([   0,   24,    1,    6, 1486, 1373,   12, 1000,   36,
```

> **Problem 2.2** *(2 points)* Using the definition in Lecture 08 (don't worry about the exact definition though), implement `TFIDF` class that subclasses `BagOfWords` class. Use natural log (instead of log with base 10).

The TF IDF is defined as below :

$$w_{i,j} = tf_{i,j} \times log(\frac{N}{df_i})$$

With :

$$w_{ij} = \text{the weight of i, j}$$
$$tf_{i,j} = \text{the number of occurrences of the word i in the document j}$$
$$df_i = \text{the number of document containing i}$$
$$N = \text{the total number of document}$$

```
class TFIDF(BagOfWords):

    def __init__(self):
        super().__init__()
        self.IDF = None

    def scores(self, TF: torch.tensor, IDF: torch.tensor) -> torch.tensor:
        """ It return the sum of the colomn of a the result matrix  """
        # we use the argument dim =1 to sum over the columns
        return torch.sum((TF * IDF ), dim=1)

    def train(self, documents: list)->None:
        super().train(documents)
        def inv(a):
          return 1/a
        DF_inv = list(map(inv, list(self.vocab.values())))
        DF  = torch.tensor(list(self.vocab.values()))
        x   = torch.tensor([len(documents[0]) * DF_inv[i] for i in range (len(DF)
        self.IDF = torch.log(x).unsqueeze(dim=0)


    def search(self, imput: str, k: int, pt=True) -> torch.tensor:

        words = unique_tokenizer(imput)
        index_list = []

        # we retrive all the index of the words of the imput
        for index, word in enumerate(self.word_to_index):
            if word in words:
                index_list.append(index)
        words_indexes = torch.tensor(index_list)

        # with the parameter dim = 1 to selct columns, with torch.index_select
        TF  = self.matrix.index_select(dim=1, index=words_indexes)
        IDF = self.IDF.index_select(dim=1, index= words_indexes)
        scores = self.scores(TF, IDF)
        if (pt):
            print('idf', IDF)
            print('scores', scores)

```

```
        index_k_max_values = scores.topk(k)[1]

        return index_k_max_values
```

```
# Small test

documents =  [["I am clément",
               "I am a student",
               "i am a the cat",'I am a Clément a cat and a cat']]

testTFIDF = TFIDF()
testTFIDF.train(documents)
testTFIDF.getWordToIndex()
testTFIDF.add(documents)

testTFIDF.search("I am Clément and cat", 3)

```

```
{'am': 0, 'clment': 1, 'i': 2, 'a': 3, 'student': 4, 'cat': 5, 'the': 6, 'and': 7}
Embedded matrix :
 tensor([[0.5774, 0.5774, 0.5774, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5000, 0.0000, 0.5000, 0.5000, 0.5000, 0.0000, 0.0000, 0.0000],
        [0.4472, 0.0000, 0.4472, 0.4472, 0.0000, 0.4472, 0.4472, 0.0000],
        [0.2425, 0.2425, 0.2425, 0.7276, 0.0000, 0.4851, 0.0000, 0.2425]],
       dtype=torch.float64)

idf tensor([[0.0000, 0.0000, 0.6931, 1.3863]])
scores tensor([0.0000, 0.0000, 0.3100, 0.6725], dtype=torch.float64)
tensor([3, 2, 0])
```

```
# Big test

tfidf = TFIDF()
tfidf.train([light_context])
tfidf.add([light_context])
topk_idx = tfidf.search(light_context[0], 10)
print("top K index of dcuments", topk_idx)
```

```
Embedded matrix :
 tensor([[0.0476, 0.0476, 0.0476,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0556, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        ...,
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0546, 0.0546, 0.0546]],
       dtype=torch.float64)

idf tensor([[ 6.2476,  0.6390,  5.8421,  2.6922,  4.5428,  5.6879,  0.7617,  0.7291,
          3.7020,  1.1118,  5.2360,  4.4558, -0.0642,  6.9407,  4.0229,  2.1491,
          4.8613,  2.9517,  3.2518,  4.5428,  2.4027,  6.2476,  4.5893,  4.9258,
          4.7435,  4.4150,  2.2357,  1.1281,  3.5563,  2.6640,  2.6366,  1.4918,
          0.0410,  7.6339,  3.8727,  3.7218,  2.8381,  1.0232,  2.7587,  4.4984,
          5.6879,  6.2476,  0.4699,  0.1467,  3.7837,  1.4790,  2.9895,  5.2360,
```

```
          5.1489,  4.0785,  4.8006,  0.0660,  4.1681,  0.5942,  4.5893,  1.8138,
          0.6408,  1.1834,  4.0229,  4.8613]])
scores tensor([13.8288,  2.9092,  3.3688,  ...,  0.5285,  0.1864,  0.4096],
       dtype=torch.float64)
top K index of dcuments tensor([ 0, 24,  8,  5, 22, 26,  7,  4, 20, 53])
```

> **Problem 2.3** *(2 points)* Use `TFIDF` to masure the recall rate of the correct document when 10 documents (contexts) are retrieved (this is called **Recall@10**) in SQuAD **validation** set.

```
def recall(model: SimilaritySearch, questions: list, k: int)-> float:
    common = 0
    total  = 0
    #for index, question in (tqdm(questions)):
    for index_and_question in (tqdm(questions)):
        #print(question)
        #topk_idx = model.search(questions, k, pt =False)
        topk_idx = model.search(index_and_question[1], k, pt =False)
        #if index in topk_idx: # that mean the index of the query is in the resul
        if index_and_question[0] in topk_idx:
            common +=1
        total += 1 # all the try
    return common / total # the right answers over all the try over
```

```
print("Recall@10 of Bag of words:" ,'\n', recall(bow, light_qestions, 10))
#result 0.5043519394512772
# the cell take 20 min to run
```

```
100%|██████████| 10570/10570 [19:38<00:00,  8.97it/s]Recall@10 of Bag of words:
 0.5043519394512772
```

```
print("Recall@10 of TFIDF:" ,'\n', recall(tfidf, light_qestions, 10))
```

```
100%|██████████| 10570/10570 [01:13<00:00, 143.97it/s]Recall@10 of TFIDF:
 0.7642384105960265
```

3. Dense Search

To obtain the embedding of each document and query, you will use pretrained word embeddings. Recall that most word embeddings are trained in a self-supervised way from a large text corpus. Here, we will use BERT word embeddings, which are also self-supervised. You will compute the document's embedding by simply averaging the embeddings of all words in the document (same for the query), and then normalizing it. This way, inner product effectively becomes cosine similarity.

BERT word embeddings can be easily obtained by using `transformers` library by Hugging Face. First, install the library:

```
!pip install -q transformers
```

You will then download the necessary tokenizer and the model.

```
from transformers import AutoTokenizer, AutoModel

model_name = 'bert-base-uncased'
tokenizer  = AutoTokenizer.from_pretrained(model_name)
model      = AutoModel.from_pretrained(model_name)
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when initiali
- This IS expected if you are initializing BertModel from the checkpoint of a model t
- This IS NOT expected if you are initializing BertModel from the checkpoint of a mod
```

Note that the tokenizer behaves a little differently from what you have done so far. It is a subword tokenizer and it inserts special tokens at the first and at the last, which should be ignored when you are computing the average.

```
text = "Hello KAIST!"
tokens = tokenizer.tokenize(text)
input_ = tokenizer(text)
input_tensor = tokenizer(text, return_tensors='pt')

print(tokens) # ['hello', 'kai', '##st', '!'] Note that (1) ## indicates subword,
print(input_['input_ids']) # [101, 7592, 11928, 3367, 999, 102], where the first
print(input_tensor['input_ids']) # same as line 7 but in PyTorch tensor
print(tokenizer.convert_ids_to_tokens(input_['input_ids'])) # you will verify tha
```

```
['hello', 'kai', '##st', '!']
[101, 7592, 11928, 3367, 999, 102]
tensor([[  101,  7592, 11928,  3367,   999,   102]])
['[CLS]', 'hello', 'kai', '##st', '!', '[SEP]']
```

The model contains not only the word embeddings but also the BERT model parameters. Here, you will only use embeddings (you will use the full model in Assignment 4).

```
output = model.embeddings(input_tensor['input_ids'])
print(output.squeeze())
print(output.size()) # [1, 6, 768], where the first dim is batch size, second
#dim is number of tokens, and third
                    #is hidden size

```

```
tensor([[ 0.1686, -0.2858, -0.3261,  ..., -0.0276,  0.0383,  0.1640],
        [ 0.3739, -0.0156, -0.2456,  ..., -0.0317,  0.5514, -0.5241],
        [-0.7024, -0.1194, -0.3570,  ..., -0.2413,  0.8315, -0.1742],
        [ 0.0478, -0.3367,  0.5948,  ...,  0.2664, -0.7800,  0.8754],
        [ 0.6424, -0.4226, -0.4063,  ...,  0.6261,  0.5611,  0.5059],
        [-0.3251, -0.3188, -0.1163,  ..., -0.3960,  0.4112, -0.0776]],
       grad_fn=<SqueezeBackward0>)
torch.Size([1, 6, 768])
```

> **Problem 3.1** *(2 points)* Implement `BERTEmbeddingSearch` class that subclasses `SimilaritySearch` class, using PyTorch's tensor native operation for the dense search.

```
### TEST ####

word = "cat"
input_tensor = tokenizer(word, return_tensors='pt')
vector = model.embeddings(input_tensor['input_ids'])
print(vector)
# we are going to use this method to create a dictionnary which link
#the words of the documents to their embedded vectors
```

```
tensor([[[ 0.1686, -0.2858, -0.3261,  ..., -0.0276,  0.0383,  0.1640],
         [ 0.6106, -0.5393, -0.0042,  ...,  0.3292,  1.0776,  1.1441],
         [-0.4815, -0.0189,  0.0092,  ..., -0.2806,  0.3895, -0.2815]]],
       grad_fn=<NativeLayerNormBackward0>)
```

```
class BERTEmbeddingSearch(SimilaritySearch):
    def __init__(self , model, tokenizer):
        super(BERTEmbeddingSearch).__init__()
        self.matrix    = None
        self.model     = model
        self.tokenizer = tokenizer
        self.word_to_vector = {}

    def matrix_bag(self, document: list, word_to_vector: dict)-> torch.tensor:
        """ The new matrix_bag take the dictionnary word to index créated in
        the training with BERT embedding. We need to do that to pass this doction
        to FAISS """
        doc_vector  = torch.stack([self.word_to_vector[word]
                                   for word in tokenizer.tokenize(document)
                                   if word in self.word_to_vector])
        # we do the mean on the lines, so we use torch.mean on dim = 0
        vector_mean =  torch.mean(doc_vector, dim = 0)
        emb_matrix  = vector_mean/torch.norm(vector_mean)
        return emb_matrix

    def train(self, documents: list) -> None:
        """generate the word_to_index dictionnary"""
```

```
        self.matrix = None
        word_set    = []
        # check if the input is a list type or not
        if isinstance(documents, (list)):

            for document in documents:
                word_set+= tokenizer.tokenize(document)
            word_set = list(set(word_set)) #to get all the words in the document

            for word in word_set: # for each word in the input.

                # we use the embedding of bert for each words to create a word_to
                # each word to its embedding
                input_tensor = tokenizer(word, return_tensors='pt')
                vector       = model.embeddings(input_tensor['input_ids'])
                self.word_to_vector[word] =  vector[0,1] # we don't take the spec


            self.is_trained  = True
            print("training complete")

        else:
            print("You need to pass list of sentance")
            self.is_trained  = False

    def getWordToVector(self)-> dict:
        """ to retrieve the word_to_index dict wich will be used in the FAISS """
        return self.word_to_vector

    def add(self, documents : list)->None:
        if self.is_trained == True:
            # we stack each embedded sentence of the document
            sub_matrix = torch.stack([self.matrix_bag(document, self.word_to_vect
            if self.matrix == None:
                self.matrix = sub_matrix
            else :
                self.matrix = torch.cat((self.matrix, sub_matrix))
            print('Embedded matrix : ', '\n', self.matrix, self.matrix.size() )

    def search(self, questions, k :int, pt= False) -> list:

        emb_query = self.matrix_bag(questions, self.word_to_vector)
        scores    = self.matrix.squeeze() @ emb_query # scalar product like befor

        index_k_max_values  = scores.topk(k)
        return index_k_max_values[1]
```

```
# Small test

documents = ["I am clément cat  ", "the KAIST",
             "I am a student",
             "i am a the cat", "I am Clément and cat"]
```

```
 6
 7
 8 testBERT = BERTEmbeddingSearch(model, tokenizer)
 9 testBERT.train(documents)
10 testBERT.add(documents)
11 print("top K index of dcuments", topk_idx)
12 testBERT.search("I am Clément and cat", 3)
13
```

```
training complete
Embedded matrix :
 tensor([[-0.0116,  0.0109, -0.0152,  ...,  0.0497,  0.0858, -0.0107],
        [-0.0435,  0.0400, -0.0299,  ...,  0.0122,  0.0488, -0.0071],
        [-0.0043,  0.0464, -0.0142,  ...,  0.0398,  0.0907, -0.0150],
        [ 0.0096,  0.0278, -0.0039,  ...,  0.0342,  0.0979,  0.0097],
        [-0.0123,  0.0187, -0.0174,  ...,  0.0573,  0.0961, -0.0006]],
       grad_fn=<StackBackward0>) torch.Size([5, 768])
top K index of dcuments tensor([ 0, 24,  8,  5, 22, 26,  7,  4, 20, 53])
tensor([4, 0, 3])
```

```
1 word_to_vector1 = testBERT.getWordToVector()
```

```
1 # Big test
2
3 bert = BERTEmbeddingSearch(model, tokenizer)
4 bert.train(light_context)
5 bert.add(light_context)
6 topk_idx = bert.search(light_context[0], 10)
7 print("top K index of dcuments", topk_idx)
8 word_to_vector = bert.getWordToVector()
```

```
Token indices sequence length is longer than the specified maximum sequence length fo
training complete
Embedded matrix :
 tensor([[ 0.0016,  0.0375, -0.0302,  ...,  0.0361,  0.0526, -0.0182],
        [ 0.0085,  0.0368, -0.0257,  ...,  0.0398,  0.0579, -0.0152],
        [ 0.0130,  0.0278, -0.0202,  ...,  0.0369,  0.0598, -0.0025],
        ...,
        [-0.0009,  0.0579, -0.0449,  ...,  0.0461,  0.0715,  0.0218],
        [-0.0003,  0.0544, -0.0337,  ...,  0.0417,  0.0489,  0.0090],
        [ 0.0025,  0.0465, -0.0357,  ...,  0.0464,  0.0665,  0.0136]],
       grad_fn=<StackBackward0>) torch.Size([2067, 768])
top K index of dcuments tensor([   0,    1,   24,    6,   21,   53,   25,   12,   42,
```

> **Problem 3.2** *(2 points)* Use `BERTEmbeddingSearch` to measure the recall at 10 for SQuAD validation dataset. How does it compare to TFIDF?

```
1 print("Recall@10 of BERTEmbeddingSearch:" ,'\n', recall(bert, light_qestions, 10)
```

```
100%|██████████| 10570/10570 [00:08<00:00, 1180.87it/s]Recall@10 of BERTEmbeddingSear
```

```
0.5599810785241249
```

> **Problem 3.3** *(2 points)* Implement `BERTEmbeddingFaiss` that subclasses `SimilaritySearch` and uses Faiss `IndexFlatIP` instead of PyTorch native tensor operation for search. Refer to the Faiss wiki (https://github.com/facebookresearch/faiss/wiki/Getting-started) for instructions.

```
!pip install faiss-cpu
```

```
Requirement already satisfied: faiss-cpu in /usr/local/lib/python3.7/dist-packages (1
```

```
import faiss
```

```
dim = 768 # dimention of BERT embedding
import numpy
```

```
class BERTEmbeddingFaiss(BERTEmbeddingSearch):
  #def __init__(self):
  #      super(BERTEmbeddingSearch).__init__()
  #      self.matrix = None

  def train(self, documents: list, word_to_vector)-> None:
      self.word_to_vector = word_to_vector
      if isinstance(documents, (list)):
        self.matrix = faiss.IndexFlatIP(dim)
        self.is_trained = True
        print("training complete")
      else:
        print("You need to pass list of sentance")
        self.is_trained = False

  def add(self, documents : list)-> None:
        if self.is_trained == True:
            # we stack the embedded matrix of each rows/sentences
            sub_matrix = torch.stack([self.matrix_bag(document,self.word_to_vecto
            for document in documents])
            # we have to comvert our embedded matrix (a tensor) into an array
            # numpy to work with FAISS, for that we use .numpy() function (and .d
            a = sub_matrix.detach().cpu().numpy()
            #we use the add function provide by FAISS
            self.matrix.add(a)
            print('Embedded matrix : ', '\n', self.matrix)

```

```
  def search(self, imput, k :int, pt= False) -> list:
        emb_imput = self.matrix_bag(imput, self.word_to_vector).unsqueeze(dim = 0
        # we use the search function provide buy FAISS
        scores    = self.matrix.search(emb_imput.detach().cpu().numpy(), k )
        index_k_max_values = scores[1][0]
        return index_k_max_values
```

```
# Small test

documents = ["I am clément cat  ", "the KAIST",
             "I am a student",
             "i am a the cat", "I am Clément and cat"]


testFAISS = BERTEmbeddingFaiss(model, tokenizer)
testFAISS.train(documents, word_to_vector1)
testFAISS.add(documents)
testFAISS.search("I am Clément and cat", 3)
```

```
training complete
Embedded matrix :
 <faiss.swigfaiss.IndexFlatIP; proxy of <Swig Object of type 'faiss::IndexFlatIP *' a
array([4, 0, 3])
```

```
# Big test

FAISS = BERTEmbeddingFaiss(model, tokenizer)
FAISS.train(light_context, word_to_vector)
FAISS.add(light_context)
topk_idx = FAISS.search(light_context[0], 10)
print("top K index of dcuments", topk_idx)
```

```
training complete
Embedded matrix :
 <faiss.swigfaiss.IndexFlatIP; proxy of <Swig Object of type 'faiss::IndexFlatIP *' a
top K index of dcuments [   0    1   24    6   21   53   25   12   42 1530]
```

```
print("Recall@10 of BERTEmbeddingFaiss:" ,'\n', recall(FAISS, light_qestions, 10)
```

```
100%|██████████| 10570/10570 [00:28<00:00, 366.82it/s]Recall@10 of BERTEmbeddingFaiss
 0.5599810785241249
```

BERT and FAISS model have the same recall result wich is logical as there are using the same embedded vector, it is juste the data structure wich changes not the values.

> **Problem 3.4** *(2 points)* Compare the speed between `BERTEmbeddingSearch` and `BERTEmbeddingFaiss` on SQuAD. To make the measurement accurate, perform search many times (at least more than 1000) and take the average.

```
from time import perf_counter
import random
def time(model, light_qestions = light_qestions, rep = 2000):
    """ perf counter return time in second so we multiplie the final result by
    1000 to have it in ms  """
    t1 = []
    for _ in range(rep):
        # we take random questions in the question set
        j = random.randint(0, len(light_qestions)-1)
        t1_start = perf_counter()
        model.search (light_qestions[j][1],10, pt = False)
        t1_stop = perf_counter()
        t1.append(t1_stop - t1_start)
    return(sum(t1)/len(t1) *1000)
time(bert)
```

```
0.6390464445071302
```

```
tBERT = time(bert)
tFAISS = time (FAISS)

print (" average time for : BERT  ", tBERT , "ms" ,'\n ')
print (" average time for : FAISS ", tFAISS , "ms" '\n ')
```

```
 average time for : BERT   0.621808752006018 ms

 average time for : FAISS  2.2434146490058993 ms
```

✓ 19 min 38 s terminée à 16:18

Impossible d'établir une connexion avec le service reCAPTCHA. Veuillez vérifier votre connexion Internet, puis actualiser la page pour afficher une image reCAPTCHA.