ML_for_NLP_Project_1_md

November 18, 2023

0.1 NFCorpusBM25.ipynb :

[ ]:
```
!pip install rank_bm25
!git clone https://github.com/cr-nlp/project1-2023.git

import urllib.request as re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
import numpy as np
from collections import defaultdict
import nltk
from sklearn.metrics import ndcg_score
from rank_bm25 import BM25Okapi
nltk.download('stopwords')
nltk.download('punkt')


def loadNFCorpus():
        dir = "./project1-2023/"

        filename = dir +"dev.docs"
        dicDoc={}
        with open(filename) as file:
                lines = file.readlines()
        for line in lines:
                tabLine = line.split('\t')
                key = tabLine[0]
                value = tabLine[1]
                dicDoc[key] = value

        filename = dir + "dev.all.queries"
        dicReq={}
        with open(filename) as file:
                lines = file.readlines()
        for line in lines:
                tabLine = line.split('\t')
                key = tabLine[0]
                value = tabLine[1]
```

```
                dicReq[key] = value

        filename = dir + "dev.2-1-0.qrel"
        dicReqDoc=defaultdict(dict)
        with open(filename) as file:
                lines = file.readlines()
        for line in lines:
                tabLine = line.strip().split('\t')
                req = tabLine[0]
                doc = tabLine[2]
                score = int(tabLine[3])
                dicReqDoc[req][doc]=score

        return dicDoc, dicReq, dicReqDoc


def text2TokenList(text):
        stopword = stopwords.words('english')
        #print("LEN DE STOPWORD=",len(stopword))
        word_tokens = word_tokenize(text.lower())
        word_tokens_without_stops = [word for word in word_tokens if word not␣
 ↪in stopword and len(word)>2]
        return word_tokens_without_stops


def run_bm25_only(startDoc,endDoc):

        dicDoc, dicReq, dicReqDoc = loadNFCorpus()

        docsToKeep=[]
        reqsToKeep=[]
        dicReqDocToKeep=defaultdict(dict)

        ndcgTop=10
        #print("ndcgTop=",ndcgTop,"nbDocsToKeep=",nbDocsToKeep)

        i=startDoc
        for reqId in dicReqDoc:
                if i > (endDoc - startDoc) :  #nbDocsToKeep:
                        break
                for docId in dicReqDoc[reqId]:
                        dicReqDocToKeep[reqId][docId] = dicReqDoc[reqId][docId]
                        docsToKeep.append(docId)
                        i = i + 1
                reqsToKeep.append(reqId)
        docsToKeep = list(set(docsToKeep))
```

```
        # Creates list of voc for docs and reqs:
        allVocab ={}
        for k in docsToKeep:
                docTokenList = text2TokenList(dicDoc[k])
                for word in docTokenList:
                        if word not in allVocab:
                                allVocab[word] = word
        allVocabListDoc = list(allVocab)

        allVocab ={}
        for k in reqsToKeep:
                docTokenList = text2TokenList(dicReq[k])
                for word in docTokenList:
                        if word not in allVocab:
                                allVocab[word] = word
        allVocabListReq = list(allVocab)

        corpusDocTokenList = []
        corpusReqTokenList = {}

# Creates token lists for docs and reqs as well as dict of docs and reqs:
  # corpusDocName == docsToKeep and corpusReqName == reqsToKeep
        corpusDocName=[]
        corpusDicoDocName={}
        i = 0
        for k in docsToKeep:
                docTokenList = text2TokenList(dicDoc[k])
                corpusDocTokenList.append(docTokenList)
                corpusDocName.append(k)
                corpusDicoDocName[k] = i
                i = i + 1

        corpusReqName=[]
        corpusDicoReqName={}
        i = 0
        for k in reqsToKeep:
                reqTokenList = text2TokenList(dicReq[k])
                corpusReqTokenList[k] = reqTokenList
                corpusReqName.append(k)
                corpusDicoReqName[k] = i
                i = i + 1

        bm25 = BM25Okapi(corpusDocTokenList)

        ndcgCumul=0
        corpusReqVec={}
        ndcgBM25Cumul=0
```

```
        nbReq=0

        for req in corpusReqTokenList:
                j=0
                reqTokenList = corpusReqTokenList[req]
                doc_scores = bm25.get_scores(reqTokenList)
                trueDocs = np.zeros(len(corpusDocTokenList))

                for docId in corpusDicoDocName:
                        if req in dicReqDocToKeep:
                                if docId in dicReqDocToKeep[req]:
                                        posDocId = corpusDicoDocName[docId]
                                        trueDocs[posDocId] =␣
 ↪dicReqDocToKeep[req][docId]

                ndcgBM25Cumul = ndcgBM25Cumul + ndcg_score([trueDocs],␣
 ↪[doc_scores],k=ndcgTop)
                nbReq = nbReq + 1

        ndcgBM25Cumul = ndcgBM25Cumul / nbReq

        print("ndcg bm25=",ndcgBM25Cumul)
        return ndcgBM25Cumul

nb_docs = 3192 #all docs
#nb_docs = 150 #for tests
run_bm25_only(0,nb_docs)
```

```
Requirement already satisfied: rank_bm25 in /usr/local/lib/python3.10/dist-
packages (0.2.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(from rank_bm25) (1.23.5)
fatal: destination path 'project1-2023' already exists and is not an empty
directory.
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

```
ndcg bm25= 0.43555347139300205
```

[ ]:
```
0.43555347139300205
```

It is now time to try and improve that model. I will start with a lemmatization and stemming in order to preprocess the corpus.

[ ]:
```
from nltk.stem import WordNetLemmatizer, PorterStemmer
from nltk.corpus import wordnet
```

```
# Downloading additional NLTK data required for lemmatization
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')

# Function to get wordnet part of speech from a simple part of speech tag
def get_wordnet_pos(word):
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}

    return tag_dict.get(tag, wordnet.NOUN)

# Lemmatization function
def lemmatize_text(text):
    lemmatizer = WordNetLemmatizer()
    lemmatized_tokens = [lemmatizer.lemmatize(w, get_wordnet_pos(w)) for w in␣
 ↪nltk.word_tokenize(text)]
    return lemmatized_tokens

# Stemming function
def stem_text(text):
    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(w) for w in nltk.word_tokenize(text)]
    return stemmed_tokens
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data…
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
```

Now let's add these functions to the text2TokenList function.

[ ]:
```
def myText2TokenList(text, method="none"):
  stopword = stopwords.words('english')
  word_tokens = word_tokenize(text.lower())
  word_tokens_without_stops = [word for word in word_tokens if word not in␣
 ↪stopword and len(word)>2]
  # Applying the specified method
  if method == "lemmatize":
      return [lemmatize_text(word)[0] for word in word_tokens_without_stops]
  elif method == "stem":
      return [stem_text(word)[0] for word in word_tokens_without_stops]
  else:
```

```
    return word_tokens_without_stops
```

We also can add synonyms to enhance the preprocessing. Let's keep in mind that depending on the context it can be less suitable to use them.

[ ]:
```
# Function to add synonyms
def add_synonyms(tokens, max_synonyms=3):
    enriched_tokens = []
    for token in tokens:
        # Add the original token
        enriched_tokens.append(token)

        # Retrieve synonyms from WordNet
        synonyms = set()
        for syn in wordnet.synsets(token):
            for lemma in syn.lemmas()[:max_synonyms]:
                synonym = lemma.name().replace('_', ' ').replace('-', ' ').
 ↪lower()
                synonyms.add(synonym)

        # Add a limited number of synonyms to avoid excessive noise
        enriched_tokens.extend(list(synonyms)[:max_synonyms])

    return enriched_tokens
```

Let's now try the provided model with a better preprocesing:

[ ]:
```
def updatedBm25(startDoc, endDoc, method=text2TokenList, model=BM25Okapi):

        dicDoc, dicReq, dicReqDoc = loadNFCorpus()

        docsToKeep=[]
        reqsToKeep=[]
        dicReqDocToKeep=defaultdict(dict)

        ndcgTop=10

        i=startDoc
        for reqId in dicReqDoc:
                if i > (endDoc - startDoc) :  #nbDocsToKeep:
                        break
                for docId in dicReqDoc[reqId]:
                        dicReqDocToKeep[reqId][docId] = dicReqDoc[reqId][docId]
                        docsToKeep.append(docId)
                        i = i + 1
                reqsToKeep.append(reqId)
        docsToKeep = list(set(docsToKeep))
```

```
        # Creates list of voc for docs and reqs:
        allVocab ={}
        for k in docsToKeep:
                docTokenList = method(dicDoc[k])
                for word in docTokenList:
                        if word not in allVocab:
                                allVocab[word] = word
        allVocabListDoc = list(allVocab)

        allVocab ={}
        for k in reqsToKeep:
                docTokenList = method(dicReq[k])
                for word in docTokenList:
                        if word not in allVocab:
                                allVocab[word] = word
        allVocabListReq = list(allVocab)

        corpusDocTokenList = []
        corpusReqTokenList = {}

# Creates token lists for docs and reqs as well as dict of docs and reqs:
  # corpusDocName == docsToKeep and corpusReqName == reqsToKeep
        corpusDocName=[]
        corpusDicoDocName={}
        i = 0
        for k in docsToKeep:
                docTokenList = method(dicDoc[k])
                corpusDocTokenList.append(docTokenList)
                corpusDocName.append(k)
                corpusDicoDocName[k] = i
                i = i + 1

        corpusReqName=[]
        corpusDicoReqName={}
        i = 0
        for k in reqsToKeep:
                reqTokenList = method(dicReq[k])
                corpusReqTokenList[k] = reqTokenList
                corpusReqName.append(k)
                corpusDicoReqName[k] = i
                i = i + 1

        chosenModel = model(corpusDocTokenList)

        ndcgCumul=0
        corpusReqVec={}
        ndcgBM25Cumul=0
```

```
        nbReq=0

        for req in corpusReqTokenList:
                j=0
                reqTokenList = corpusReqTokenList[req]
                doc_scores = chosenModel.get_scores(reqTokenList)
                trueDocs = np.zeros(len(corpusDocTokenList))

                for docId in corpusDicoDocName:
                        if req in dicReqDocToKeep:
                                if docId in dicReqDocToKeep[req]:
                                        posDocId = corpusDicoDocName[docId]
                                        trueDocs[posDocId] =␣
 ↪dicReqDocToKeep[req][docId]

                ndcgBM25Cumul = ndcgBM25Cumul + ndcg_score([trueDocs],␣
 ↪[doc_scores],k=ndcgTop)
                nbReq = nbReq + 1

        ndcgBM25Cumul = ndcgBM25Cumul / nbReq

        #print("ndcg bm25=",ndcgBM25Cumul)
        return ndcgBM25Cumul
```

[ ]:
```
def tokenListLematized(text):
  return myText2TokenList(text, method="lemmatize")

def tokenListStemmed(text):
  return myText2TokenList(text, method="stem")

print("Provided: ", updatedBm25(0,nb_docs))
print("Lematized: ",updatedBm25(0,nb_docs, tokenListLematized))
print("Stemmed: ", updatedBm25(0,nb_docs, tokenListStemmed))
```

```
Provided:  0.43555347139300205
Lematized:  0.4438042665332428
Stemmed:  0.4480795606587508
```

[ ]:
```
def add_synonymsProvided(text):
  return add_synonyms(text2TokenList(text))

def add_synonymsLem(text):
  return add_synonyms(tokenListLematized(text))

def add_synonymsStem(text):
  return add_synonyms(tokenListStemmed(text))
```

```
print("Synonyms: ",updatedBm25(0,nb_docs, add_synonymsProvided))
print("SynonymsLem: ",updatedBm25(0,nb_docs, add_synonymsLem))
print("SynonymsStem: ",updatedBm25(0,nb_docs, add_synonymsStem))
```

```
Synonyms:  0.41618666163242013
SynonymsLem:  0.4086184450478863
SynonymsStem:  0.3820931500611803
```

We can see that the stemming method alone is the one that gives the better ngcd, we'll then keep this one in preprocessing.

To make easier the work we will separate the run_bm25_only function into three of them.

[ ]:
```
def load_and_tokenize(startDoc,endDoc):
  dicDoc, dicReq, dicReqDoc = loadNFCorpus()

  docsToKeep=[]
  reqsToKeep=[]
  dicReqDocToKeep=defaultdict(dict)

  i=startDoc
  for reqId in dicReqDoc:
    if i > (endDoc - startDoc) :  #nbDocsToKeep:
      break
    for docId in dicReqDoc[reqId]:
      dicReqDocToKeep[reqId][docId] = dicReqDoc[reqId][docId]
      docsToKeep.append(docId)
      i = i + 1
    reqsToKeep.append(reqId)
  docsToKeep = list(set(docsToKeep))

  # Creates list of voc for docs and reqs:
  allVocab ={}
  for k in docsToKeep:
    docTokenList = tokenListStemmed(dicDoc[k])
    for word in docTokenList:
      if word not in allVocab:
        allVocab[word] = word
  allVocabListDoc = list(allVocab)

  allVocab ={}
  for k in reqsToKeep:
    docTokenList = tokenListStemmed(dicReq[k])
    for word in docTokenList:
      if word not in allVocab:
        allVocab[word] = word
  allVocabListReq = list(allVocab)
```

```
  corpusDocTokenList = []
  corpusDocTokenDict = {}
  corpusReqTokenDict = {}

  # Creates token lists for docs and reqs as well as dict of docs and reqs:
    # corpusDocName == docsToKeep and corpusReqName == reqsToKeep
  corpusDocName=[]
  corpusDicoDocName={}
  i = 0
  for k in docsToKeep:
    docTokenList = tokenListStemmed(dicDoc[k])
    corpusDocTokenDict[k] = docTokenList
    corpusDocTokenList.append(docTokenList)
    corpusDocName.append(k)
    corpusDicoDocName[k] = i
    i = i + 1

  corpusReqName=[]
  corpusDicoReqName={}
  i = 0
  for k in reqsToKeep:
    reqTokenList = tokenListStemmed(dicReq[k])
    corpusReqTokenDict[k] = reqTokenList
    corpusReqName.append(k)
    corpusDicoReqName[k] = i
    i = i + 1

  return dicDoc, dicReq, dicReqDoc, corpusDocTokenList, corpusDocTokenDict,␣
 ↪corpusReqTokenDict, corpusDicoDocName, dicReqDocToKeep

def apply_bm25_model(corpusDocTokenList, corpusReqTokenDict, corpusDicoDocName,␣
 ↪dicReqDocToKeep):
  bm25 = BM25Okapi(corpusDocTokenList)
  ndcgBM25Cumul=0
  nbReq=0
  ndcgTop=10
  doc_scores_dict = {}
  doc_scores_dict_named = {}

  for req in corpusReqTokenDict:
    j=0
    reqTokenList = corpusReqTokenDict[req]
    doc_scores = bm25.get_scores(reqTokenList)
    doc_scores_dict[req] = doc_scores
    trueDocs = np.zeros(len(corpusDocTokenList))

    for docId in corpusDicoDocName:
```

```
      if req in dicReqDocToKeep:
        if docId in dicReqDocToKeep[req]:
          posDocId = corpusDicoDocName[docId]
          trueDocs[posDocId] = dicReqDocToKeep[req][docId]

    ndcgBM25Cumul = ndcgBM25Cumul + ndcg_score([trueDocs],␣
 ↪[doc_scores],k=ndcgTop)
    nbReq = nbReq + 1

  ndcgBM25Cumul = ndcgBM25Cumul / nbReq
  print("ndcg bm25=",ndcgBM25Cumul)
  return ndcgBM25Cumul, doc_scores_dict
```

[ ]:
```
dicDoc, dicReq, dicReqDoc, corpusDocTokenList, corpusDocTokenDict,␣
 ↪corpusReqTokenDict, corpusDicoDocName, dicReqDocToKeep =␣
 ↪load_and_tokenize(0,nb_docs)
ndcgBM25Cumul, doc_scores_dict = apply_bm25_model(corpusDocTokenList,␣
 ↪corpusReqTokenDict, corpusDicoDocName, dicReqDocToKeep)
#print(doc_scores_dict)
```

```
ndcg bm25= 0.4480795606587508
```

Let's try to use Word2Vec.

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

# Train Word2Vec model on the preprocessed corpus
word2vec_model = Word2Vec(corpusDocTokenList, vector_size=300, window=3,␣
 ↪min_count=1, workers=4)
```

[ ]:
```
# Function to vectorize a document using the Word2Vec model
def vectorize_with_word2vec(text, model):
    vectorized = [model.wv[word] for word in text if word in model.wv]
    return np.mean(vectorized, axis=0) if vectorized else np.zeros(model.
 ↪vector_size)
```

[ ]:
```
# Vectorize documents
doc_vectors = {doc_id: vectorize_with_word2vec(doc_tokens, word2vec_model) for␣
 ↪doc_id, doc_tokens in corpusDocTokenDict.items()}
```

[ ]:
```
# Vectorize queries
query_vectors = {query_id: vectorize_with_word2vec(query_tokens,␣
 ↪word2vec_model) for query_id, query_tokens in corpusReqTokenDict.items()}
```

[ ]:
```
from sklearn.metrics.pairwise import cosine_similarity

# Function to retrieve and rank documents for a given query
def retrieve_and_rank(query_vec, doc_vectors):
```

```
    scores = {doc_id: cosine_similarity(query_vec.reshape(1, -1), doc_vec.
 ↪reshape(1, -1))[0][0]
              for doc_id, doc_vec in doc_vectors.items()}
    ranked_docs = sorted(scores.items(), key=lambda x: x[1], reverse=True)
    return scores, ranked_docs
```

[ ]:
```
ranked_results = {}
scores = {}
for query_id, query_vec in query_vectors.items():
    scores[query_id], ranked_results[query_id] = retrieve_and_rank(query_vec,␣
 ↪doc_vectors)

#print('ranked_results', ranked_results['PLAIN-1'])
#print('scores', scores['PLAIN-1'])
```

[ ]:
```
def combine_scores(bm25_scores, ranked_results, alpha=0.5):
    combined_scores = {}

    for query_id in bm25_scores:
        # Get BM25 scores for the current query
        bm25_score_list = bm25_scores[query_id]

        # Get Word2Vec scores for the current query
        w2v_score_dict = ranked_results.get(query_id, {})

        # Combine the scores
        combined_query_scores = {}
        for doc_idx, bm25_score in enumerate(bm25_score_list):
            # Assuming document IDs in bm25_scores are in the same order as in␣
 ↪w2v_score_dict
            doc_id = list(w2v_score_dict.keys())[doc_idx] if doc_idx <␣
 ↪len(w2v_score_dict) else None
            if doc_id:
                w2v_score = w2v_score_dict.get(doc_id, 0)
                combined_score = alpha * bm25_score + (1 - alpha) * w2v_score
                combined_query_scores[doc_id] = combined_score

        # Store the combined scores for the query
        combined_scores[query_id] = combined_query_scores

    return combined_scores
```

[ ]:
```
combined_scores = combine_scores(doc_scores_dict, scores, alpha=0.5)
#print(combined_scores["PLAIN-1"])
```

[ ]:
```
# Code to save the combined scores to be able to use the ranking in an App.
import json
```

```
file_path = '/combined_scores.json'

with open(file_path, 'w') as file:
    json.dump(combined_scores, file)
print(f"The dictionary has been exported to {file_path}")
```

```
The dictionary has been exported to /combined_scores.json
```

[ ]:
```
from sklearn.metrics import ndcg_score
import numpy as np

def evaluate_with_ndcg(combined_scores, relevance_judgments):
    ndcg_values = []

    for query_id in combined_scores:
        # Get the combined scores and the relevance judgments for the current␣
 ↪query
        query_scores = combined_scores[query_id]
        query_judgments = relevance_judgments[query_id]

        # Sort the documents by their combined scores
        sorted_docs = sorted(query_scores, key=query_scores.get, reverse=True)

        # Create lists of true relevances and predicted scores in the sorted␣
 ↪order
        true_relevances = [query_judgments.get(doc_id, 0) for doc_id in␣
 ↪sorted_docs]
        predicted_scores = [query_scores[doc_id] for doc_id in sorted_docs]

        # Calculate NDCG for the current query
        ndcg_value = ndcg_score([true_relevances], [predicted_scores])
        ndcg_values.append(ndcg_value)

    # Calculate the average NDCG across all queries
    average_ndcg = np.mean(ndcg_values) if ndcg_values else 0
    return average_ndcg
```

[ ]:
```
mean_ndcg = evaluate_with_ndcg(combined_scores, dicReqDoc)
print(mean_ndcg)
```

```
0.5912885589752738
```

We had 0.44 and now we have 0.5912885589752738 as ndgc. With for word2vec vector_size = 100, window = 3, min_count = 1