Τμήμα Μηχανικών Η/Υ & Πληροφορικής
Πανεπιστήμιο Πατρών

Ανάκτηση Πληροφορίας

Εργαστηριακή Άσκηση
Χειμερινό Εξάμηνο 2023

ΛΕΥΤΕΡΗΣ ΑΜΙΤΣΗΣ 1072464 up1072464@ac.upatras.gr
ΣΑΡΑΝΤΗΣ ΠΑΠΑΧΡΙΣΤΟΦΙΛΟΥ 1072600 up1072600@ac.upatras.gr

Εισαγωγή

Κατά την εκπόνηση αυτής της εργασίας υλοποιήθηκαν δύο μοντέλα ανάκτησης πληροφορίας το Vector Space Model και το colBert. Το Vector Space Model δημιουργεί διανύσματα τόσο στην συλλογή με τα docs όσο και στα queries για τα οποία θα αναζητήσει τα σχετικά αρχεία. Τα διανύσματα αυτά δημιουργούνται από τους όρους(terms) τους αφού τους ανατεθούν τα κατάλληλα βάρη και έχουν την μορφή D=(tι,wdi; tj,wdj;…;tp,wdp) για τα αρχεία και Q=(qa,wqa; qb,wqb;…;qr,wqr) για τα queries. Επομένως το VSM είναι ενα σύστημα που υπολογίζει το συνημίτονο της γωνίας που σχηματίζουν τα διανύσματα του query και των docs. Επιστρέφει ένα ranking με τα διανύσματα που εμφανίζουν την μικρότερη γωνία άρα και που συγκλίνουν πιο κοντά στο διάνυσμα του query. Στο project αυτό υλοποιήσαμε δύο εκδόσεις του Vector Space Model η μία είναι η Best fully weighted system tfc*nfx και η άλλη είναι η Best weighted probabilistic weight nxx*bpx. Επιλέχθηκαν αυτοί οι μέθοδοι διότι είναι πιο αποτελεσματικοί λόγω της ελαστικότητας τους στην μορφή των queries (μικρά ή μεγάλα , ανεκτικά σε πιο λυτό λόγο) αφού χρησιμοποιούν length normalization στα documents και enhanced query weighting στα queries. Το άλλο μοντέλο που χρησιμοποιήσαμε είναι το προεκπαιδευμένο και υλοποιημένο colBert. Το Bert είναι ένα μοντέλο νευρωνικού δικτύου που παίρνει για είσοδο διανύσματα που προέρχονται από τα tokens του κειμένου και δίνει στην έξοδο ένα sequence από σχετικά embeddings. Το colBert είναι ένα μοντέλο που χρησημοποιεί ως βάση του το Bert για να κωδικοποιήσει

το query και το document και χρησιμοποιεί ένα φθηνό βήμα αλληλεπίδρασης για να μοντελοποιήσει τις ομοιότητες τους.

- ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT
  Omar Khattab, Matei Zaharia

- Pretrained Transformers for Text Ranking: BERT and Beyond
  Jimmy Lin, Rodrigo Nogueira, Andrew Yates

- Gerard Salton, Christopher Buckley,Term-weighting approaches in automatic text retrieval, Information Processing & Management, Volume 24, Issue 5, 1988,Pages 513-523,ISSN 0306-4573,https://doi.org/10.1016/0306-4573(88)90021-0.

Παρατηρήσεις

Η υλοποίηση του project πραγματοποιήθηκε σε στο visual studio code σε python και για ορισμένα ερωτήματα χρησιμοποιήθηκε το jupyter notebook.
Για την χρήση του μοντέλου ColBERT, απαιτείται η εγκατάσταση των Python 3.7+, Pytorch 1.9+ και της βιβλιοθήκης Hugging Face Transformers. Αρχικά, δημιουργήσαμε ένα περιβάλλον conda χρησιμοποιώντας την εντολή: **conda env create -f /home/sarantis/Documents/InformationRetrieval/Project/ColBERT/conda_env.yml**
Η εντολή δεν περιέχει το κομμάτι "_cpu", διότι γίνεται χρήση της κάτας γραφικών του συστήματός μας.Το περιβάλλον που δημιουργήσαμε ενεργοποιείται ως εξής: **conda activate colbert**

Για την αξιοποίηση της gpu εγκαταστάθηκε το cuda toolkit, μέσω της ιστοσελίδας: **https://developer.nvidia.com/cuda-downloads**

Υλοποίηση

Ερώτημα 1 - Ανάγνωση και Επεξεργασία της Συλλογής

Για την υλοποίηση αυτού του ερωτήματος δημιουργήθηκε το function **create_df** στο python script **creating_aplo.py**, που περιέχει functions που βοηθάνε στην υλοποίηση όλων των ερωτημάτων. Στα πλαίσια αυτού του ερωτήματος χρησημοποίθηκε η βιβλιοθήκη os για το διάβασμα της συλλογής καθώς και η βιβλιοθήκη pandas για την μετατροπή του dictionary σε dataframe. Η μετατροπή αυτή έγινε αφού κρίναμε ότι σε μια τέτοια μορφή θα μπορέσουν να γίνουν πιο εύκολα οι αριθμητικές πράξεις του δεύτερου ερωτήματος. Αρχικά ξεκινάμε με την δημιουργία του dictionary. Δημιουργείται ένα εμφολευμένο function read_text_file(Εικόνα 1) το οποίο παίρνει σαν όρισμα το path της τοποθεσίας της συλλογής. Το function αυτό δημιουργεί ένα dictionary για να αποθηκεύει πόσες φορές εμφανίστηκε το token στο κείμενο(times) καθώς και μία μεταβλητή με το όνομα του αρχείου, παίρνει μία προς μία τις γραμμές του doc αποθηκεύει το token της κάθε γραμμης και δημιουργεί και ένα dictionary για να κρατάει σε ποια θέση εμφανίστηκε (αυτό τελικά δεν χρειάστηκε στην υλοποίηση). Έπειτα για κάθε token κρατάμε πόσες φορές εμφανίστηκε συνολικά και πόσες μέσα στο αρχείο που διαβάζουμε καθώς και το όνομα του αρχείου, έτσι με τα δεδομένα αυτά δημιουργούμε το simple dictionary. Έπειτα ξεκινάμε και καλούμε την read_text_file για κάθε file στο doc(Εικόνα 2) και ολοκληρώνουμε την δημιουργία του simple dictionary. Ύστερα κρατάμε τα ονόματα των docs τα κάνουμε sorting για να μπουν στην σειρά ,αρχικοποιούμε ένα νέο dictionary όπου για κάθε doc δημιουργεί μία θέση για κάθε token και της δίνει την τιμή 0 (term frequency). Έπειτα διαβάζουμε το simple dictionary και για κάθε token βάζουμε στην θέση του σωστού αρχείου το σωστό tf για το κάθε token. Τέλος μετατρέπουμε το λεξικό αυτό σε pandas dataframe που έχει για indexes τα ονόματα των αρχείων , για columns τα tokens που εμφανίστηκαν σε όλα τα docs και σε κάθε κελί βρίσκεται ο αριθμός των εμφανίσεων του κάθε token στο αντίστοιχο doc. Άρα καταλήγουμε με ένα dataframe με τα “vectors” του κάθε doc.

```
def read_text_file(file_path):

    with open(file_path, 'r') as f:
        tokens=[]
        showed_up={}
        times={}
        f_name=''
        for line in f:
            f_name=os.path.basename(f.name)
            check_token = line.strip()
            tokens.append(check_token)
        for index,token in enumerate(tokens):
            if token in showed_up:
                showed_up[token].append(index+1)
                times[token]+=1
            else:
                showed_up[token] = [index+1]
                times[token] = 1

            if token in simple_dictionary:
                simple_dictionary[token][0]+=1
                simple_dictionary[token].append((f_name,times[token]))
            else:
                simple_dictionary[token]=[1,(f_name,times[token])]
```

Εικόνα 1.

```
for file in os.listdir():
    file_path = os.path.join(path, file)

    read_text_file(file_path)


data_dict = {}
doc_ids = []

for data in simple_dictionary.values():
    for item in data[1:]:
        doc_ids.append(item[0])


doc_names = sorted(set(doc_ids))

for token in simple_dictionary:
    data_dict[token] = {doc_id: 0 for doc_id in doc_names}


for word, word_data in simple_dictionary.items():
    for doc_id, count in word_data[1:]:
        data_dict[word][doc_id] = count

df = pd.DataFrame(data_dict)
df.index.name = 'Document'

return df
```

Εικόνα 2.

Ερώτημα 2 - Υλοποίηση Vector Space μοντέλου

Για την υλοποίηση κρίναμε σκόπιμο να χρησιμοποιήσουμε τα δύο πιο αποτελεσματικά weighting systems για συλλογές με ποικιλία στο μέγεθος των εγγράφων και για κείμενα πιο απλού-λυτού λόγου. Τα συστήματα αυτά είναι το Best fully weighted system και το Best weighted probabilistic weight. Η υλοποιήση πραγματοποίηθηκε στο script **classes.py** στην κλάση VSM που παίρνει ως όρισμα το dataframe του προηγούμενου ερωτήματος. Τα functions της κλάσης είναι τα εξής:

tf → f/maxf :

```
def tf(self,df):
    tf_df=pd.DataFrame()
    tf_df=df
    tf_df=tf_df.div(tf_df.max(axis=1),axis=0)
```

idf→ log(N/ni) :

```
def idf(self):
    idf_arr=[]
    idf_arr=np.count_nonzero(self.df, axis=0)
    idf_arr=idf_arr.astype(float)
    for index, value in enumerate(idf_arr):
        if value == 0:
            continue
        else:
            idf_arr[index] = math.log10(len(self.df.index) / value)
    return idf_arr
def idf_to_df(self):
    idf_df=pd.DataFrame(data=self.idf().reshape(1, -1),columns=self.df.columns)
    return idf_df
```

tf*idf :

```
def calculate_W(self,df):
    W=self.tf(df).mul(self.idf(),axis=1)

    return W
```

(tf*idf)^2 (σε επόμε function θα προστεθεί και η ρίζα):

```
def file_DTW(self,df):
    return  self.calculate_W(df).apply(lambda row: row**2, axis=1)
```

πρόσθεση όλων των όρων και υπολογισμός της ρίζας για υπολογισμό του doc-weight:

```
def paranomastis(self):
    paranomastis=self.file_DTW(self.df).sum(axis=1)
    paranomastis=np.sqrt(paranomastis)
```

Υπολογισμός βάρους για τα αρχεία:

```
def weight(self):
    W = self.calculate_W(self.df)
    result_df = self.calculate_W(self.df).div(self.paranomastis().values, axis=0)
    return result_df
```

Δημιουργία vectors για τα querys:

```
def get_query(self,n):
    questions = []
    with open("/home/lefteris/Ceid/anaktisi_pliroforias/ir_2023-2024/Queries_20", 'r') as file:
        lines = file.readlines()
        i = 0

        while i < len(lines):
            questions.append(lines[i].strip())
            i += 1

            vsm_common = self.idf_to_df()

    query1=questions[n].split()
    query1=self.filter_w(query1)
    data_q = {}
    for token in query1:
        if token in data_q:
            data_q[token] += 1
        else:
            data_q[token] = 1


    query_pd=pd.DataFrame(data_q,index=["0"])
    df_query_big = pd.DataFrame(np.zeros((1, 11368)),columns=vsm_common.columns)
    query_pd.columns = query_pd.columns.str.upper()

    for col in query_pd.columns:
        df_query_big[col] = query_pd[col].values

    return df_query_big
```

Υπολογισμός βάρους για τα queries:

```
def query_res(self,n):
    df_query=self.get_query(n)
    res_df= self.tf(df_query)

    res_df=0.5*res_df
    for col in res_df.columns:
        res_df[col] = res_df[col].apply(lambda x: x + 0.5 if x > 0 else 0)
    idf=self.idf_to_df()

    query_res= idf  * res_df
    return query_res
```

Σε αυτή τη φάση έχουμε τα παρακάτω βάρη:

Doc Weight:

$$\frac{\mathrm{tf} \cdot \log \frac{N}{n}}{\sqrt{\sum_{vector} \left(\mathrm{tf}_i \cdot \log \frac{N}{n_i} \right)^2}}$$

Query Weight:

$$\left(0.5 + \frac{0.5\ \mathrm{tf}}{\max \mathrm{tf}} \right) \cdot \log \frac{N}{n}$$

Υπολογισμός similarity / cos για το best fully weighted system :

```
def term_A(self,n):
    doc_W = self.weight()
    term_a = doc_W * self.query_res(n).iloc[0]
    term_as=term_a.sum(axis=1)
    return term_as
def term_B(self,n):
    DocW2=self.file_DTW(self.weight())
    DocW_s=DocW2.sum(axis=1)
    query_tb=self.query_res(n).apply(lambda row: row**2, axis=1)
    query_tb1=query_tb.sum(axis=1)

    term_b=np.sqrt(DocW_s * query_tb1.loc[0])
    return term_b
def Calc_Weights(self,n):
    res=self.term_A(n) / self.term_B(n)

    ten_largest_values = pd.Series(res).nlargest(10)
    indices_of_ten_largest_values = ten_largest_values.index
    print("Three largest values:\n", ten_largest_values)
    print("Indices of three largest values:", indices_of_ten_largest_values)
    return indices_of_ten_largest_values
```

doc weight για το probabilistic weight:

```
def Weighted_create(self):
    Dtw2=self.tf(self.df)
    Dtcw=0.5*Dtw2
    for col in Dtcw.columns:
        Dtcw[col] = Dtcw[col].apply(lambda x: x + 0.5 if x > 0 else 0)
    return Dtcw
```

Στο query weight εμφανίζεται ένας πολλαπλασιασμός παραπάνω. Αυτό οφείλεται στο ότι πολλαπλασιασμε το self.df με το query_v2 και να μηδενίσουν όλοι οι όροι που δεν βρίσκονται στο query. Έπειτα μετράμε όπως και σε προηγούμενα function τις μη μηδενικές τιμές των tokens που υπάρχουν στα queries και τέλος κάνουμε τις απαραίτητες πράξεις για τον υπολογισμό του βάρους.

query weight για το probabilistic weight:

```
def safe_log10(self,value):
    if value > 0 and value<len(self.df.index):
        res=math.log10((len(self.df.index) - value) / value)
        if res>=0:
            return res
        else:
            return 0
    else:
        return 0
def weighted_query(self,n):
    query_v2=self.get_query(n)
    test = self.df * query_v2.iloc[0]
    test = test.apply(lambda x: (x !=0).sum())
    test = test.apply(self.safe_log10)
    test_df=test.to_frame()
    test_df=test_df.transpose()
    return test_df
```

Τα probabilistic weight για τα docs είναι:

$$0.5 + \frac{0.5\ \text{tf}}{\max \text{tf}}$$

Τα probabilistic weight για τα querys είναι:

$$\log \frac{N - n}{n}$$

Υπολογισμός συνημιτόνου για το best weighted probabilistic weight:

```
def term_Av2(self,n):
    term_a_v2=self.Weighted_create() * self.weighted_query(n).iloc[0]
    term_a_v2=term_a_v2.sum(axis=1)

    return term_a_v2
def term_Bv2(self,n):
    Dtw2_termb=self.Weighted_create().apply(lambda row: row**2, axis=1)
    quert_v2_termb=self.weighted_query(n).apply(lambda row: row**2, axis=1)
    Dtw2_termb=Dtw2_termb.sum(axis=1)
    quert_v2_termb=quert_v2_termb.sum(axis=1)
    term_b_v2=np.sqrt(Dtw2_termb * quert_v2_termb.loc[0])
    return term_b_v2
def propabilistic_res(self,n):
    res_2=self.term_Av2(n) / self.term_Bv2(n)


    ten_largest_values_2 = pd.Series(res_2).nlargest(10)
    indices_of_ten_largest_values_2 = ten_largest_values_2.index

    print("Three largest values:\n", ten_largest_values_2)
    print("Indices of three largest values:", indices_of_ten_largest_values_2)
    return indices_of_ten_largest_values_2
```

Επιπλέον υπάρχουν τα δύο functions ένα που χρησιμοποιείται για φιλτράρισμα κάποιων λέξεων και το τελευταίο που καλούμε για να πάρουμε τα αποτελέσματα όλων των querys και να γίνει η αξιολόγηση των μοντέλων.

```
def filter_w(self,array):
    filter_words = ["ON","IN","FROM","THE","OF","WHAT","WHERE","THIS"]
    array = [word.upper() for word in array if word.upper() not in filter_words]
    return array
def get_res(self):
    results1=[]
    results2=[]
    for res in range(0,20):
        results2.append(self.propabilistic_res(res).tolist())
        results1.append(self.Calc_Weights(res).tolist())
    results1 = [[int(element) for element in sublist] for sublist in results1]
    results2 = [[int(element) for element in sublist] for sublist in results2]
    # for item in results1:
    #     item.reverse()
    return results1,results2
```

Ερώτημα 3 – Μοντέλο ColBERT

Για την υλοποίηση του ερωτήματος 3, ακολουθήσαμε τα βήματα που μας υποδεκνύονται στο github repository, https://github.com/stanford-futuredata/ColBERT. Τα παρακάτω βήματα εκτελούνται στο αρχείο colbert.ipynb.

Step 0: Προεπεξεργασία Συλλογής κειμένων και ερωτημάτων. Το μοντέλο ColBERT λειτουργεί με Tab-Seperated-Files, συνεπώς διαβάσαμε το περιεχόμενο του φακέλου docs και του αρχείου Queries_20 και δημιουργήσαμε δύο TSV αρχεία. Τα αρχεία αυτά είναι το collection.tsv, το οποίο περιέχει όλα τα κείμενα με το id τους και το queries.tsv, το οποίο περιέχει όλα τα ερωτήματα με τον αριθμό τους.

Στον παρακάτω κώδικα αρχικά, γίνεται η εισαγωγή των απαραίτητων βιβλιοθηκών και modules που θα χρησιμοποιηθούν σχετικά με το ColBERT μοντέλο. Στη συνέχεια, ορίζουμε το μονοπάτι που περιέχει τον φάκελο docs με τα αρχεία που θέλουμε να διαβάσουμε και δημουργούμε την λίστα data για την αποθήκευση του περιεχομένου τους. Μέσω μίας for διαβάζουμε το ID και το περιεχόμενο κάθε κειμένου και τα κάνουμε append στην λίστα data. Τα δεδομένα που συλλέχθηκαν από τα έγγραφα αποθηκεύονται σε ένα αρχείο TSV. Κάθε γραμμή περιέχει το Document ID, ένα tab, και το περιεχόμενο του εγγράφου.

```
import os
import sys
import numpy as np
import matplotlib as plt
sys.path.insert(0, '../')

from colbert.infra import Run, RunConfig, ColBERTConfig
from colbert.data import Queries, Collection
from colbert import Indexer, Searcher

documents_directory = '/home/sarantis/Documents/InformationRetrieval/Project/docs/'

data = []

next_document_id = 0
for line_idx, filename in enumerate(sorted(os.listdir(documents_directory))):

    with open(os.path.join(documents_directory, filename), 'r') as file:
        content = file.read().replace('\n', ' ')

    data.append((next_document_id, content))

    next_document_id += 1

output_file = '/home/sarantis/Documents/InformationRetrieval/Project/file.tsv'
with open(output_file, 'w', encoding='utf-8', newline='\n') as tsvfile:
    for document_id, content in data:
        tsvfile.write(f"{document_id}\t{content}\n")

print(f"TSV file created at: {output_file}")
```

Με παρόμοιο τρόπο, αντιμετωπίζουμε και το αρχείο Queries_20 με σκοπό τη δημιουργία του TSV με τα ερωτήματα. Ορίζεται το μονοπάτι του αρχείου Queries_20 και μία λίστα για την προσθήκη του περιεχομένου του. Έπειτα, γίνεται ανάγνωση των ερωτημάτων από το αρχείο, χωρισμός τους σε λέξεις και σε κάθε αλλαγή γραμμής append ο αριθμός του ερωτήματος και οι λέξεις του. Τα δεδομένα που προέρχονται από τα ερωτήματα αποθηκεύονται σε ένα αρχείο TSV. Κάθε γραμμή περιέχει τον αριθμό του ερωτήματος και τις λέξεις που το απαρτίζουν.

```
queries_file_path = '/home/sarantis/Documents/InformationRetrieval/Project/Queries_20'

data = []

with open(queries_file_path, 'r', encoding='utf-8') as queries_file:
    for query_number, query in enumerate(queries_file, start=1):
        words = query.strip().split()

        data.append((str(query_number), words))

output_file = '/home/sarantis/Documents/InformationRetrieval/Project/queries.tsv'
with open(output_file, 'w', encoding='utf-8', newline='\n') as tsvfile:
    for query_number, words in data:
        tsvfile.write(f"{query_number}\t{' '.join(words)}\n")

print(f"TSV file created at: {output_file}")
```

Δημιουργούμε τις απόλυτες διαδρομές για τα δύο αρχεία TSV που προέκυψαν. Επίσης, χρησιμοποιούμε τις κλάσεις Queries και Collection από το ColBERT για τη φόρτωση των ερωτημάτων και της συλλογής αντίστοιχα. Τέλος, επιστρέφεται ένα μήνυμα που δείχνει πόσα ερωτήματα και πόσα έγγραφα φορτώθηκαν.

```
dataroot='/home/sarantis/Documents/InformationRetrieval/Project/'
dataset=''
datasplit=''

queries = os.path.join(dataroot, dataset, datasplit, 'queries.tsv')
collection = os.path.join(dataroot, dataset, datasplit, 'file.tsv')

queries = Queries(path=queries)
collection = Collection(path=collection)

f'Loaded {len(queries)} queries and {len(collection):,} passages'
```

Step 1: Για το βήμα 1 κατεβάσαμε το αρχείο **pre-trained ColBERTv2 checkpoint.** Το checkpoint αυτό, είναι ήδη trained με το MARCO Passage Ranking task και το χρησιμοποιούμε καθώς, σύμφωνα με την εκφώνηση καλούμαστε να εφαρμόσουμε το μοντέλο, χωρίς να το εκπαιδεύσουμε από την αρχή.

Step 2: Το βήμα 2, αφορά το indexing της συλλογής με σκοπό την γρήγορα ανάκτηση. Αρχικά, καθορίζουμε τον αριμθό των bits που χρησιμοποιούνται για την κωδικοποίηση κάθε διάστασης και το το μέγιστο μήκος που θα έχουν τα έγγραφα. Ακόμα, καθορίζουμε το path του checkpoint-colbertv2.0 και το όνομα του ευρετηρίου που θ χρησιμοποιηθεί.

```
nbits = 2   # encode each dimension with 2 bits
doc_maxlen = 300   # truncate passages at 300 tokens

checkpoint = 'downloads/colbertv2.0'
index_name = f'{dataset}.{datasplit}.{nbits}bits'
```

Στο σημείο αυτό χρησιμοποιούμε το Παράδειγμα Χρήσης που παρέχεται στο github repository. Εδώ ο κώδικας εκτελείται σε περιβάλλον ColBERT, χρησιμοποιούμε τις κλάσεις Run και RunConfig. Καθορίζουμε την χρήση μίας gpu και ότι εκτελούμε σε περιβάλλον notebook και στη συνέχεια δημιουργούμε, το indexer για το ColBERT.

```
with Run().context(RunConfig(nranks=1, experiment='notebook')):
    config = ColBERTConfig(doc_maxlen=doc_maxlen, nbits=nbits)

    indexer = Indexer(checkpoint=checkpoint, config=config)
    indexer.index(name=index_name, collection=collection, overwrite=True)
```

Step 3: Το βήμα 3, αφορά το searching της συλλογής μέσω των queries. Ομοίως με το indexing αξιοποιούμε το παράδειγμα χρήσης από το github του Stanford.

```
with Run().context(RunConfig(experiment='notebook')):
    searcher = Searcher(index=index_name, collection=collection)
```

Ρυθμίζοντας μία παράμετρο k, ανακτούμε τα k πιο σχετικά κείμενα με ένα ερώτημα.

```
query = queries[14]
print(f"#> {query}")

results = searcher.search(query, k=10)
real_id=0
#query-1 apo tin arxi

# Print the top-k retrieved passages
for passage_id, passage_rank, passage_score in zip(*results):
    print(f"{passage_id}\t [{passage_rank}] \t\t {passage_score:.1f} \t\t {searcher.collection[passage_id]}")
```

Ακολουθεί αποτέλεσμα της εκτέλεσης του κώδικα, όπου για το ερώτημα 14, μας επιστράφηκαν τα 10 πιο σχετικά κείμενα(IDs και Περιεχόμενο) μαζί με το rank σχετικότητάς τους.

Αποτέλεσμα για το query 14:

```
#> What non-invasive tests can be performed for the evaluation of exocrine pancreatic function in patients with CF
```

```
795     [1]     19.7    PABA SCREENING TEST FOR EXOCRINE PANCREATIC FUNCTION IN INFANTS AN
879     [2]     18.9    DIAGNOSIS OF EXOCRINE PANCREATIC INSUFFICIENCY IN CYSTIC FIBROSIS
147     [3]     18.5    USE OF SERUM AMYLASE ISOENZYMES IN EVALUATION OF PANCREATIC FUNCTI
1091    [4]     18.1    PABATEST FOR DIAGNOSIS OF EXOCRINE PANCREATIC INSUFFICIENCY LETTER
778     [5]     17.7    MECONIUM PLUG SYNDROME CYSTIC FIBROSIS AND EXOCRINE PANCREATIC DEF
923     [6]     17.7    MECONIUM SCREENING FOR CYSTIC FIBROSIS IN OUR HANDS THE BMCTEST ME
306     [7]     17.6    RECURRENT ACUTE PANCREATITIS IN PATIENTS WITH CYSTIC FIBROSIS WITH
171     [8]     17.6    ADDITIONAL DATA ON HEPATIC FUNCTION TESTS IN CYSTIC FIBROSIS FIFTY
566     [9]     17.3    DISTRIBUTION OF SERUM AMYLASE ISOENZYMES IN CYSTIC FIBROSIS HOMOZY
452     [10]    17.3    QUANTITATIVE EVALUATION OF SERUM PANCREATIC ISOAMYLASES IN CYSTIC
```

Δημιουργία csv αρχείου με τα αποτελέσματα, τα οποία δίνει το μοντέλο ColBERT

Στην συλλογή μας έχουμε συνολικά 1209 των οποίων τα id κυμαίνονται από το 1 έως το 1239. Ωστόσο, το tsv αρχείο που δημιουργήσαμε νωρίτερα αποθηκεύει τα κείμενα με την σειρά και με συνεχόμενα id’s στο διάστημα [0, 1208]. Αυτό σημαίνει πως για να υπάρξει σωστή αξιολόγηση του μοντέλου θα πρέπει να αντιστοιχήσουμε τα νέα id’s που προέκυψαν με τα αρχικά. Εντοπίζουμε, λοιπόν, τα id’s που δεν υπάρχουν στο διάστημα [1, 1209] χρησιμοποιώντας την παρακάτω συνάρτηση:

```python
def find_missing_docs():
    docs_directory = '/home/sarantis/Documents/InformationRetrieval/Project/docs'

    total_docs = 1209
    expected_titles = range(1, 1240)

    existing_titles = [int(doc.split('.')[0]) for doc in os.listdir(docs_directory)]

    missing_docs = set(expected_titles) - set(existing_titles)

    return sorted(missing_docs)
```

Στην συνέχεια, χρησιμοποιούμε την λίστα με τα id’s που λείπουν και διορθώνουμε τα queries που προέκυψαν στο tsv αρχείο, κάνοντάς τα ίδια με τα αρχικά εντός της λίστας results_10. Συνεπώς, η λίστα αυτή περιέχει για κάθε ερώτημα τα διορθωμένα id’s των δέκα πιο σχετικών κειμένων με το ερώτημα αυτό.

```python
def is_between(number, lower_bound, upper_bound):
    return lower_bound <= number <= upper_bound
results_10=[]
missing_docs = find_missing_docs()
for query_index, query_tuple in enumerate(queries):
    query = query_tuple[1]
    token=[]
    print(f"#> Query {query_index + 1}: {query}")

    results = searcher.search(query, k=10)

    for passage_id, passage_rank, passage_score in zip(*results):

        title = passage_id
        for i in range(0, len(missing_docs)-2):
            if is_between(title, missing_docs[i], missing_docs[i + 1]):
                title = title + i + 2
                while(title>missing_docs[i + 1]):
                    title+=1
                    i+=1
                break
            elif title < 129:
                title += 1
                break

        passage_content = searcher.collection[passage_id]
        print(f"{title}\t [{passage_rank}] \t\t {passage_score:.1f} \t\t {passage_content}")
        token.append(title)
    results_10.append(token)
    print("\n")
```

Για καλύτερη αξιολόγηση του μοντέλου μέσω precision και recall, δημιουργούμε άλλες τέσσερις λίστες, τις results_8, results_6, results_4, results_2, οι οποίες περιέχουν τα 8, 6, 4 και 2 πιο σχετικά κείμενα αντίστοιχα.

```python
print(results_10)
results_8 = []
results_6 = []
results_4 = []
results_2 = []

for sublist in results_10:
    results_8.append(sublist[:-2])
    results_6.append(sublist[:-4])
    results_4.append(sublist[:-6])
    results_2.append(sublist[:-8])

print(results_8)
print(results_6)
print(results_4)
print(results_2)
```

Προκειμένου να αξιοποιήσουμε τα αποτελέσματα που εξάγαμε από το ColBERT δημιουργούμε το αρχείο colBERT.csv. Το αρχείο αυτό περιέχει για τα 20 ερωτήματα τα id’s των 10 πιο σχετικών κειμένων, πιο κάτω τα id’s των 8 πιο σχετικών κειμένων, τα id’s των 6 πιο σχετικών κειμένων, τα id’s των 4 πιο σχετικών κειμένων και τα id’s των 2 πιο σχετικών κειμένων.

```python
import csv
csv_file_path='../colBERT.csv'
with open(csv_file_path, mode='w', newline='') as file:
    csv_writer = csv.writer(file)

    csv_writer.writerows(results_10)
    csv_writer.writerow([])
    csv_writer.writerows(results_8)
    csv_writer.writerow([])
    csv_writer.writerows(results_6)
    csv_writer.writerow([])
    csv_writer.writerows(results_4)
    csv_writer.writerow([])
    csv_writer.writerows(results_2)

print(f'Data has been written to {csv_file_path}')

with open(csv_file_path, mode='r') as file:
    csv_reader = csv.reader(file)

    read_data_lists = list(csv_reader)

print(f'Data has been read from {csv_file_path}: {read_data_lists}')
```

Ερώτημα 4 – Συγκρίσεις

Για τις ανάγκες αυτού του ερωτήματος δημιουργήθηκε η κλάση Metrics στο script **classes.py** . Οι μετρικές που επιλέξαμε να υλοποιήσουμε είναι ο Αρμονικός Μέσος ώστε να παρατηρήσουμε πιο απο τα μοντέλα εξισορροπεί καλύτερα την σχέση precision-recall και η NDCG για να δούμε πόσο σχετικά είναι τα κείμενα που ανακτούμε στο κάθε μοντέλο.

Αρχικά ξεκινάμε με το function το οποίο διαβάζει το Relevant_20 με τα relevant κείμενα του κάθε query για τα οποία βάζει στο array relevant_docs_for_q τον αριθμό τους, ενώ στο array rel_docs βάζει τα ονόματα των σχετικών αρχείων.

```
def rel_num_list(self):
    rq_path='/home/lefteris/Ceid/anaktisi_pliroforias/ir_2023-2024/Relevant_20'
    relevant_docs_for_q=[]
    rel_docs=[]
    with open(rq_path, 'r') as f:
        for line in f:
            token_num = line.split()
            rel_docs.append(token_num)
            relevant_docs_for_q.append(len(token_num))
    return relevant_docs_for_q,rel_docs
```

Έπειτα είναι το function το οποίο συγκρίνει τα αρχεία που έγιναν retrieve για το query i (i ο αριθμός των querys) με τα σχετικά κείμενα του συγκεκριμένου query και επιστρέφει τον αριθμό των σχετικών.

```
def find_rel(self,i):
    idc,rd=self.rel_num_list()
    rel = [int(item1) for item1 in self.res1[i] for item2 in rd[i] if int(item1) == int(item2)]
    return len(rel)
```

Το επόμενο function είναι αυτό που υπολογίζει το recall με χρήση του find_rel function για το κάθε query και επιστρέφει τον μέσο όρο τους.

```
def recall(self):
    rdq,idc=self.rel_num_list()
    recall = []
    for i in range(0,20):
        recall_q=self.find_rel(i) / rdq[i]
        recall.append(recall_q)
    print(f'The recall metric of the model is: {np.mean(recall)} \n\n')
    return np.mean(recall)
```

Παρόμοια είναι και η συνάρτηση υπολογισμού του precision και επιστρέφει πάλι τον μέσο όρο όλων των querys.

```
def precision(self):
    precision=[]
    for i in range(0,20):
        precision_q=self.find_rel(i) / self.num_of_retrieved
        precision.append(precision_q)

    print(f'The precision metric of the model is: {np.mean(precision)} \n\n')
    return np.mean(precision)
```

Υπολογισμός μέσης αρμονικής τιμής του μοντέλου.

```
def MesiArmonikiTimi(self):
    pre=self.precision()
    rec=self.recall()
    F = ( 2 * rec * pre ) / (pre+rec)
    return F
```

Έπειτα ακολουθεί το function που διαβάζει το cfquery_detailed doc και κρατάει σε ένα array για κάθε query τα σχετικά κείμενα και τον βαθμό σχετικότητάς τους. Ο βαθμός σχετικότητας είναι στο πεδίο [0,2] και υπολογίζεται απο το άθροισμα όλων των βαθμολογιών, δηλαδή για sum>=6 βαθμός=2 για 4<=sum<6 βαθμός=1 και για sum<4 βαθμός=0.

```
def get_res_list(self):
    result_list=[]
    with open('/home/lefteris/Ceid/anaktisi_pliroforias/ir_2023-2024/cfquery_detailed', 'r') as file:
        values = []
        flag = 0

        for line in file:
            parts = line.split()

            if parts and parts[0] == 'RD':
                flag = 1
                values = []

                for i in range(1, len(parts)):
                    if i % 2 == 0:
                        values.append(self.sum_of_digits(parts[i]))
                    else:
                        values.append(int(parts[i]))

                result_list.append(values)
            elif not parts:
                flag = 0
                values = []
            elif flag == 1:
                for i in range(len(parts)):
                    if i % 2 == 1:
                        values.append(self.sum_of_digits(parts[i]))
                    else:
                        values.append(int(parts[i]))
    return result_list
```

Τέλος είναι ο υπολογισμός του NDCG που ξεκινάει με αρχικοποίηση του τελικού πίνακα final και μιας flag μεταβλητής append_next. Ύστερα ξεκινάμε να διαβάζουμε ένα προς ένα τα εμφολευμένα array που το καθένα περιέχει στις ζυγές τους θέσεις τα ονόματα των σχετικών αρχείων και στις μονές τον βαθμό σχετικότητας. Έτσι ελέγχουμε όλες τις ζυγές θέσεις και αν δούμε κοινό αρχείο μεταξύ των σχετικών και των ανεκτιμένων κρατάμε τον βαθμό σχετικότητας ενώ αν δεν το βρούμε βάζουμε ένα 0. Έτσι δημιουργούνται τα διανύσματα μέσα στο final.Ύστερα δημιουργούμε την sorted_final για τον υπολογισμό του ideal dcg.

```
def DCG(self):
    final = []
    append_next=0
    for j in range(20):
        comp = []
        for num in self.res1[j]:
            found = 0
            for index,num2 in enumerate(self.get_res_list()[j]):
                if index % 2 == 0 or append_next==1:
                    if append_next==1 :
                        comp.append(num2)
                        append_next=0
                    if num == num2:
                        append_next=1
                        found=1
                if found == 0 and index == len(self.get_res_list()[j]) -1:

                    comp.append(0)
            if len(comp)==self.num_of_retrieved:
                final.append(comp)
    sortedfinal=[]
    for item in final:
        s=sorted(item)
        s.reverse()
        sortedfinal.append(s)
    return final,sortedfinal
```

Έπειτα έχουμε το function που υπολογίζει και για τα δύο arrays τους σωρευθείς φορείς και έπειτα κάνει το discount. Ύστερα υπολογίζουμε το normalized DCG που θα χρησιμοποιήσουμε για την αξιολόγηση των μοντέλων μας

```
def create_gvec(self,vec):
    for item in vec:
        sumi=0
        for i in range(len(item)):
            if i!=0:
                item[i] = item[i] / np.log2(i+1)
                item[i]=item[i]+sumi
                item[i] = round(item[i], 2)
            sumi=item[i]
    ml=[]
    for i in range(len(vec[0])):
        test=0
        for j in range(len(vec)):
            test+=vec[j][i]
        test_f= test / len(vec)
        test_f=round(test_f,2)
        ml.append(test)
    return ml
def calculate_ndcg(self):
    dcg , ndcg = self.DCG()
    dcg= self.create_gvec(dcg)
    ndcg= self.create_gvec(ndcg)
    res = []
    for i in range(len(dcg)):
        res.append(dcg[i] / ndcg[i])
    return res
```

ΠΕΙΡΑΜΑΤΙΚΗ ΥΛΟΠΟΙΗΣΗ:

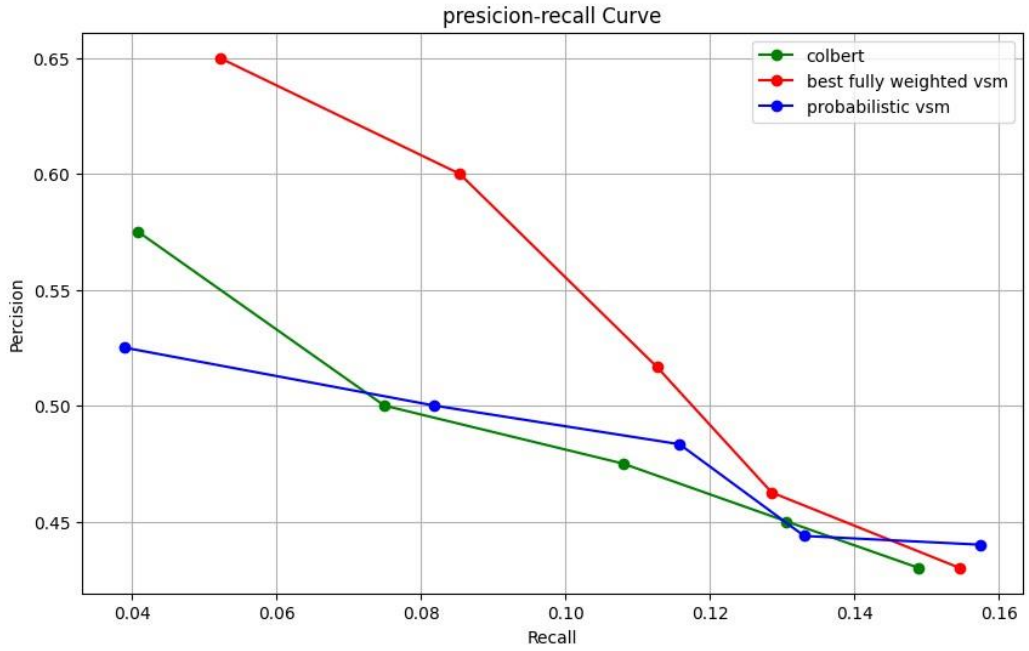
Στο jupyter notebook test.ipynb έχει γραφτεί σε python το πειραματικό κομμάτι του project. Κάνουμε import τα scripts classes και creating για να πάρουμε τα απαραίτητα functions και κλάσεις καθώς και το matplotlib.pyplot για δημιουργία γραφημάτων και απεικονίσεων των αποτελεσμάτων. Επιπλέον να σημειωθεί ότι τα αποτελέσματα που εξάγαμε από το colBERT όπως προαναφέρθηκε μπήκαν στο colBERT.csv για να μπορέσουμε να τα αξιοποιήσουμε σε αυτό το notebook. Ετσι λοιπόν δημιουργήσαμε 5 set δεδομένων για κάθε μοντέλο (για το VSM διαφορετικά sets για την Best fully weighted εκδοχή και διαφορετικά για την probabilistic εκδοχή). Έπειτα δημιουργούμε ένα Metrics αντικείμενο για κάθε σετ και ένα array για κάθε μοντέλο (πάλι ξεχωριστά για τις δύο εκδοχές του VSM) και για κάθε μετρική precision,recall,harmonic mean,dcg. Τέλος κάνουμε plot τα αποτελέσματα και εξάγουμε συμπεράσματα.

Αποτελέσματα - Παρατηρήσεις

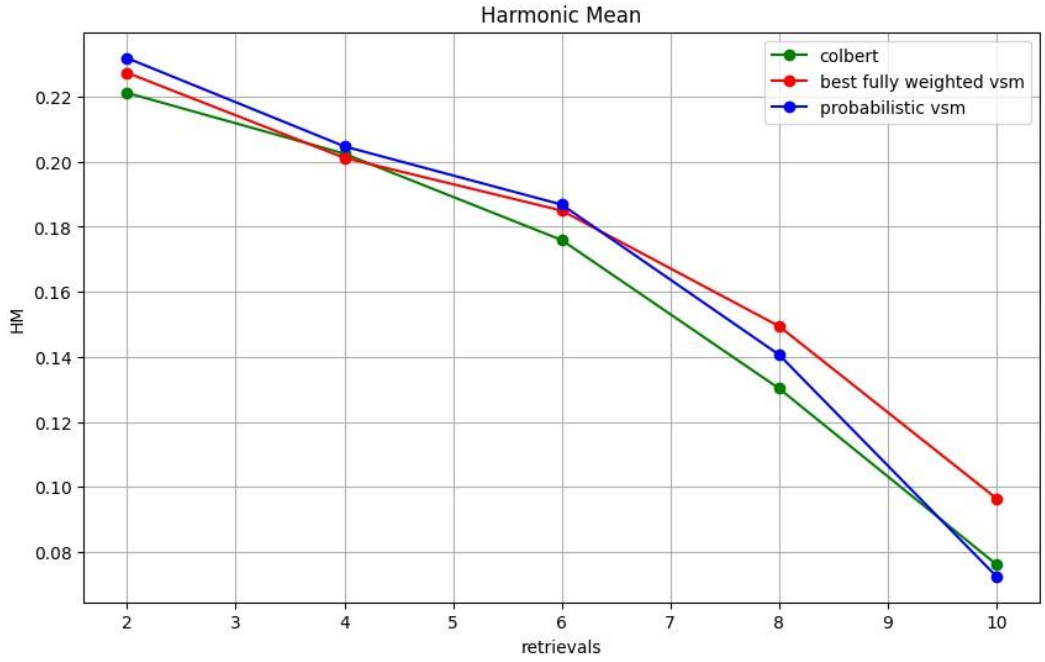
Στα πλαίσια αυτής της εργασίας καλούμαστε να συγκρίνουμε δύο μοντέλα ανάκτησης πληροφορίας με διαφορετικές προσεγγίσεις ως προς το τι θεωρούν σχετικό με το ερώτημα που δόθηκε . Το vector space model θεωρεί σχετικά τα κείμενα που παρουσιάζουν ίδιες λέξεις με αυτές του ερωτήματος χωρίς να δίνει σημασία σε συνώνυμες λέξεις και τα συμφραζόμενα του κειμένου. Από την άλλη το colbert δημιοργεί contextual embeddings τα οποία δεν δίνουν τόση σημασία στις ίδιες λέξεις αλλά ψάχνουν και συνώνυμα και στα συμφραζόμενα.Αρχικά θα ξεκινήσουμε συγκρίνοντας τις δύο εκδοχές του VSM και αφού καταλήξουμε στην καλύτερη θα την συγκρίνουμε με το colBERT.

ΑΠΟΤΕΛΕΣΜΑΤΑ

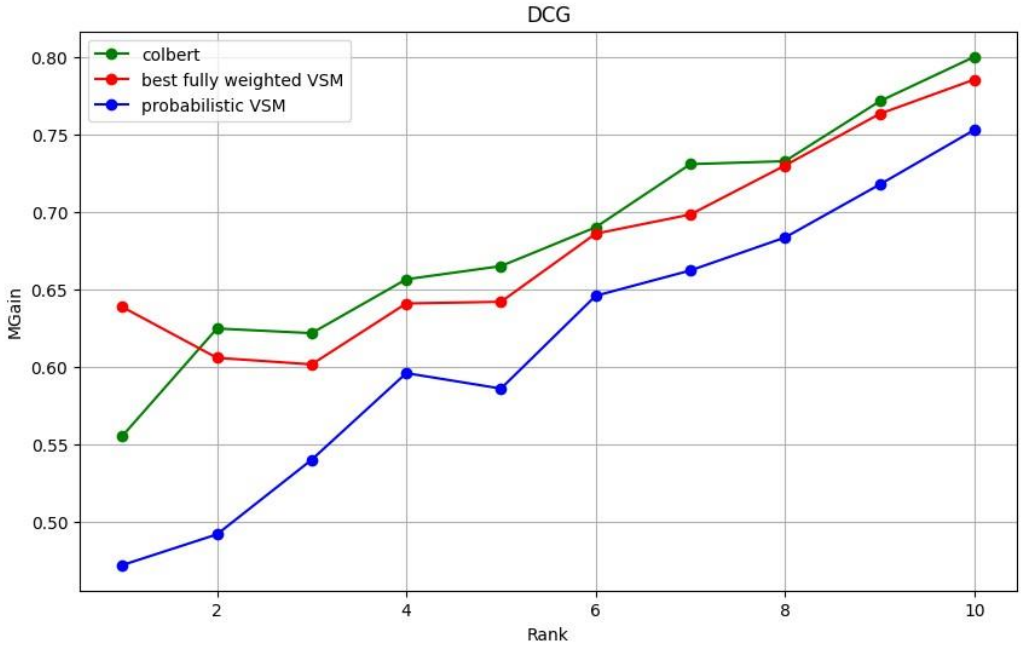
PRESICION-RECALL CURVE



HARMONIC MEAN CURVE



NDCG CURVE



ΣΥΜΠΕΡΑΣΜΑΤΑ

Probabilistic εκδοχή VSM vs Fully weighted VSM

Όπως είναι προφανές απο τις παραπάνω καμπύλες η probabilistic εδκοχή του VSM είναι πολύ κατώτερη από την fully weighted τόσο στο precision-recall όσο και στο gain. Πρέπει να σημειωθεί πως έχει έναν αξιόλογο αρμονικό μέσο για ένα μικρό αριθμό ανάκτησης αλλά δεν έχει κάποιο άλλο αξιοσημείωτο γνώρισμα για ανταγωνιστεί το fully weighted.

Best Fully Weighted VSM

Το Best fully weighted VSM όπως βλέπουμε έχει το υψηλότερο precision-recall με διαφορά. Αυτό οφείλεται στο length normalization του αρχείου καθώς και ο πρώτος όρος στο βάρος του query 0.5 + (0.5 * tf)/maxtf ο οποίος μειώνει σε πάρα πολύ μεγάλο βαθμό την επιρροή των πολυεμφανιζόμενων tokens. Επομένως μεταξύ των δύο είναι η καλύτερη εκδοχή του VSM με την οποία και θα συγκρίνουμε το μοντέλο colBERT.

Best Fully Weighted VSM vs colBERT

Όπως είναι προφανές από την precision-recall curve και την harmonic mean curve το VSM είναι ανώτερο τόσο στο precision όσο και στο recall. Ωστόσο το αξιοσημείωτο είναι ότι το colBERT όπως βλέπουμε στο nDCG curve έχει περισσότερο gain από το VSM παρόλο που το δεύτερο(VSM) ανέκτησε παραπάνω σχετικά αρχεία. Αυτό οφείλεται στην ικανότητα του colBERT να βρίσκει συνώνυμες λέξεις, μοτίβα καθώς και το να έχει μια αντίληψη των συμφραζόμενων και έτσι κατάφερε να ανακτήσει παραπάνω αρχεία που σχετίζονται με τα querys σε μεγάλο βαθμό και άρα που επιφέρουν μεγαλύτερο gain (είχαν μεγαλύτερο βαθμό). Αυτό υποδηλώνει ότι σε μία μεγαλύτερη συλλογή που ενδεχομένως να μην υπήρχαν πολλά αρχεία που να παρουσίαζουν κοινές λέξεις με τα ερώτήματα το colBERT θα απέδιδε πολύ καλύτερα.