Τμήμα Μηχανικών Η/Υ & Πληροφορικής

Πανεπιστήμιο Πατρών

Ανάκτηση Πληροφορίας

Εργαστηριακή Άσκηση

Χειμερινό Εξάμηνο 2024

Ελένη Μαράκη AM: 1084534 up1084534@upnet.gr

Δήμητρα Μαλαταρά AM: 1090056 up1090056@upnet.gr

Εισαγωγή

- Θεωρητικα για τα μοντέλα που θα υλοποιηθούν με αναφορές

Boolean Model:

Η ανάκτηση εγγράφων βασιζεται στην ικανοποίηση, ή μη, μιας λογικής πρότασης. Η λογική πρόταση αυτή εκφράζει την αναγκαιότητα, ύπαρξης λέξεων (/tokens) στα αρχεία της συλλογής. Χρησιμοποιούνται οι λογικοί τελεστές AND, OR και NOT, για την δημιουργία των λογικών εκφράσεων. Στο μοντέλο αυτό η έννοια του μερικού ταιριάσματος είναι άτοπη, καθώς η ανάκτηση βασίζεται πλήρως σε δυαδικά κριτήρια απόφασης.

Vector Space Model:

Το διανυσματικό μοντέλο χώρου *(Salton et al., 1975; Salton and Buckley, 1988)* αναγνωρίζει ότι το δυαδικό ταίριασμα του Boolean μοντέλου είναι περιοριστικό και προτείνει την επιλογή να υπάρξει μερικό ταίριασμα. Αναθέτοντας μη δυαδικά βάρη στους όρους ευρετηρίασης στα ερωτήματα και τα έγγραφα που χρησιμοποιούμε για να βρούμε την ομοιότητα μεταξύ τους. Η ταξινόμηση γίνεται με βάση το βαθμό ομοιότητας(κατά φθίνουσα σειρά) όπου επιλέγονται και αρχεία που ταιριάζουν μόνο εν μέρει με τους όρους του κάθε ερωτήματος. Εν κατακλείδι παρατηρούμε ότι τα ταξινομημένα έγγραφα αποτελούν μια καλύτερη λύση σε σχέση με το σύνολο των εγγράφων που αναρτώνται στο Boolean model. Καθώς ο χρήστης μπορεί να δει

πρώτα τα πιο σχετικά έγγραφα και όχι το σύνολο όλων τον σχετικών εγγράφων χωρίς διαβάθμιση.

- Τεχνικές και βιβλιοθήκες με αναφορές

Για το Boolean Model και το Vector Space Model χρησιμοποιήθηκαν οι εξής βιβλιοθήκες:

- os
- nltk
- nltk.stem
- PorterStemmer from nltk.stem
- stopwords from nltk.corpus
- Counter from collections
- numpy
- matplotlib.pyplot
- math

Υλοποίηση

- Αλγοριθμικη λογική της υλοποίησης
- Παρουσίαση των σημαντικότερων σημείων της υλοποίησης

Προεπεξεργασία:

Τα κείμενα της συλλογής που παρέχεται ακολουθούν πολύ συγκεκριμένη δομή. Το γεγονός αυτό περιορίζει το πλήθος των κατηγοριών προεπεξεργασίας που είναι αναγκαία. Αναλυτικότερα, η έλλειψη κεφαλαίων γραμμάτων και ειδικών χαρακτήρων και ο διαχωρισμός των λέξεων του αρχείου με “\n”, θέτει την λεξιλογική ανάλυση περιττή. Παρ΄ όλα αυτά χρησιμοποιείται η βιβλιοθήκη nltk, για την αφαίρεση των stopwords, εξαλείφοντας λέξεις με υψηλή συχνότητα που δεν προσφέρουν πληροφορία. Ακόμα, με την ενσωμάτωση του Porter Stemmer πραγματοποιείται μείωση του χώρου δομής ευρετηριοδίτησης, αφαιρώντας περιττά προθέματα και επιθέματα, επιτυγχάνωντας σύμπτυξη παρόμοιων όρων. Είναι εφικτή λοιπόν η αντιστοίχηση όρων διαφορετικών αρχείων, παρά των πιθανών διαφορών στην μορφή τους. Τέλος, υπολογίζεται η συχνότητα εμφάνισης κάθε όρου (Term Frequency -TF), η οποία είναι αναγκαία για την αργότερη υλοποίηση του TF-IDF.

Για το Boolean Model:

-Inverted_Index.py: Η δομή του ευρετηρίου είναι η ακόλουθη. Το ευρετήριο συμπεριλαμβάνει όλα τα tokens των κειμένων και για το καθένα από αυτά τα κείμενα στα οποία εμφανίζεται. Για την υλοποίηση του Boolean μοντέλου παραπάνω πληροφορίες είναι περιττές καθώς δεν χρειάζονται για την λειτουργία του. Αυτή η δομή του ευρετηρίου διευκολύνει την εξέταση ύπαρξης ή όχι, των tokens που καθορίζουν την λογική έκφραση εισόδου του μοντέλου.

Στην υλοποίηση του συγκεκριμένου μοντέλου, στο οποίο ήταν αναγκαία η χρήση προτάσεων (Queries) τα οποία δεν ακολουθούσαν δομή λογικών εκφράσεων, ο μετασχηματισμός του σε λογικές προτάσεις επιτεύχθηκε με την συνένωση των tokens των query με λογικούς τελεστές. Καθώς ο λογικός τελεστής AND περιόριζε αρκετά το πλήθος των σχετικών κειμένων, δημιουργώντας μία δύσκολα ικανοποιήσιμη λογική πρόταση, θεωρήθηκε προτιμότερη η χρήση του λογικού τελεστή OR. Με την ενσωμάτωση αυτού σημειώνεται ότι τα κείμενα τα οποία θα επιλέγονται από τα μοντέλα αυξάνονται σε πλήθος, μειώνοντας όμως και την πιθανή σχετικότητά τους με το query.

Για την υλοποίηση του Boolean Μοντέλου που θα αναγνωρίζει την λογική έκφραση και θα επιστρέφει τα σχετικά κείμενα, επιλέχθηκε η υλοποίηση μιας απλής και αποδοτικής διαδικασίας. Με την ανάγνωση της εισόδου του μοντέλου διαχωρίζονται η λέξεις της λογικής έκφρασης, αναγνωρίζοντας ξεχωριστά τον κάθε λογικό τελεστή, από τις υπόλοιπες λέξεις. Η προσέγγιση που ακολουθήθηκε, είναι η αναπαράσταση της εισόδου ως μιας λογικής έκφρασης, με τους ίδιους λογικούς και αντί των λέξεων εισόδου, μία λίστα με τα ids των κειμένων που περιέχουν την κάθε λέξη. Στην περίπτωση του NOT, ο λογικός τελεστής αντικαθίστανται από την λίστα όλων των κειμένων μείων την λίστα των κειμένων που περιέχουν την επόμενη λέξη εισόδου. Η λογική έκφραση εισόδου αναπαρίσταται ως μία αντίστοιχη λογική έκφραση από ids κειμένων που την ικανοποιουν και λύνεται με την συνάρτηση eval της python. Τελικά, τα ids των κειμένων που ικανοποιούν την λογική έκφραση θα είναι αυτά που το μοντέλο θα επιστρέψει ως σχετικά.

Για το Vector Space Model:

-Inverted_Index.py: Η αλγοριθμική λογική της υλόποίησης είναι η δημιουργία ενός ανεστραμμένου ευρετηρίου, δηλαδή μια δομή δεδομένων που αντιστοιχίζει όρους στα αρχεία στα οποία εμφανίζονται και τις συχνότητες τους(TF).

-Vector_Space_Model.py: Για κάθε όρο που υπάρχει στα **έγγραφα**, υπολογίζουμε το TF-IDF βάρος. Τα βάρη αυτά αποθηκεύονται σαν διανύσματα για κάθε έγγραφο. Το TF είναι η συχνότητα εμφάνισης ενός όρου, και δίνει μεγαλύτερη σημασία σημαντικούς όρους μέσα στο έγγραφο. Το IDF υποδεικνύει την αντίστροφη συχνότητα εγγράφων, μειώνοντας τη σημασία των κοινών όρων. Το γινόμενο TF-IDF δίνει μεγαλύτερη βαρύτητα σε συχνούς όρους ενώ παράλληλα σπάνιοι σε ολόκληρο το collection.

Ο τύπος είναι:

TF = (term frequency)/(max term frequency)

IDF = log10(total docs/docs that have that term)

Επίσης για κάθε **ερώτημα** το οποίο αναλύεται σε stemmed tokens δημιουργούμε tf-idf διάνυσμα για τους όρους του ερωτήματος.

Τύπος που χρησιμοποιήθηκε:

TF = (term frequency in the query/total amount of terms in the query)

το **IDF** υπολογίζεται από τα δεδομένα του ανεστραμμένου αρχείου.

Στη συνέχεια υπολογίζουμε το **cosine similarity**:
η συσχέτιση κάθε εγγράφου με το ερώτημα υπολογίζεται μέσω του παρακάτω τύπου:

$$sim(q,d_j) = \cos(\theta) = \frac{\vec{d}_j \bullet \vec{q}}{\left|\vec{d}_j\right| \times \left|\vec{q}\right|}$$

όπου d,q είναι τα διανύσματα για τα βάρη των όρων στο κάθε έγγραφο και ερώτημα αντίστοιχα.

Τέλος, τα ερωτήματα κατατάσσονται με βάση την ομοιότητα τους με το ερώτημα και επιστρέφουμε τα top-k πιο σχετικά έγγραφα.

Η επιλογή του συνδυασμου **TF-IDF** και **cosine similarity** έγινε καθώς η ανάκτηση με αυτό τον τρόπο είναι ευαίσθητη στους μοναδικούς όρους των ερωτημάτων και των κειμένων. Επίσης αποτελεί μια πολύ καλή μέθοδο κατάταξης σχετικοτητας.

Κάποια από τα προβλήματα που συναντήσαμε ήταν αρχικά στην υλοποίηση του tf-idf για τα ερωτήματα, πιο συγκεκριμένα στον υπολογισμό του idf. Χρειάστηκε δηλαδή να χρησιμοποιήσουμε το DF απο το inverted index, που αφορά όλα τα έγγραφα από τη συλλογή.

Συγκρίσεις μεταξύ των μοντέλων:

Με τις μετρικές απόδοσης, Precision και Recall μπορούμε να αξιολογήσουμε την αποδοτικότητα του κάθε μοντέλου.
Στο **Boolean Model** η απόδοση για τα πρώτα κείμενα εξαρτάται εξ΄ολοκλήρου από τους λογικούς τελεστες (AND/OR). Επομένως, επιστρέφει είτε όλα τα σχετικά είτε κανένα κείμενο.
Στο **VSM** ταξινομούμε τα έγγραφα με βάση τη συσχέτιση (cosine similarity), άρα έχουμε μεγαλύτερη ακρίβεια στα πρώτα αποτελέσματα.

Στη περίπτωση που αξιολογούμε την απόδοση ανα ερώτημα, στο boolean model η ανάκληση είναι πιθανότατα χαμηλή καθώς πρέπει να ικανοποιούνται αυστηρά κριτήρια. Ενώ το vsm μας δίνει πιο έμπιστα αποτελέσματα με καλύτερη ανάκληση αλλά και ακρίβεια.

Όπως θα δούμε και παρακάτω στο cell όπου τρέχει κάθε μοντέλο, ο χρόνος εκτέλεσης του boolean ειναι γρηγορότερος απο αυτον του vsm. Αναμενόμενο καθώς η ευρετηρίαση στο vsm απαιτεί τον υπολογισμό tf-idf(χρονοβόρο), ενώ στο boolean model γινεται με απλούστερο τρόπο(only inverted index). Επίσης το recall στο boolean model ολοκληρώνεται άμεσα-εξαιτίας των λογικών πράξεων- ενώ στο vsm χρειάζεται να υπολογιστεί η συσχέτιση.

Με βάση τα παραπάνω, το **VSM** είναι γενικά πιο αποδοτικό για μεγαλύτερες και πιο σύνθετες συλλογές καθώς ενσωματώνει το μερικό ταίριασμα, ενώ το **Boolean Model** είναι καλύτερο για συγκεκριμένα και προκαθορισμένα ερωτήματα.

Αναφορές

Salton, G., Buckley, C., 1988. Term-weighting Approaches in Automatic Text Retrieval. Inf Process Manage 24, 513–523. https://doi.org/10.1016/0306-4573(88)90021-0

Salton, G., Wong, A., Yang, C.S., 1975. A Vector Space Model for Automatic Indexing. Commun ACM 18, 613–620. https://doi.org/10.1145/361219.361220

Παράρτημα

- Κωδικας.
- Σχόλια πάνω στον κωδικα που παρουσιάζεται.

main.py

```
import os
import matplotlib.pyplot as plt
import nltk
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
from collections import Counter
import numpy as np

from Inverted_Index import InvertedIndex
from Boolean_Model import BooleanModel
```

```
from Vector_Space_Model import VectorSpaceModel
nltk.download('stopwords')
#%% md
- Initialization
#%%
stemmer = PorterStemmer()
stopwords = stopwords.words("english")
iv = InvertedIndex()
#%% md
- Preprocess

The documents in out collection already follow a predefined format. There
are no capital letter, no special characters and each word is seperated
with newline. Therefore the preprocess actions required are minimal. The
first step followed is tokenization of the documents, removal of stopwords
that may be included and stemming for the remaining words. The processed
words from the document are used to create two inverted indexes. One
including only the document tokens and which document/s they are located
in. The simplicity of this inverted index rests in its usage by the Boolean
Model, which only considers the presence or not of the tokens in each
document.  The second inverted index also includes the TF and DF of each
token, which will be later used for the TF-IDF calculation in the Vector
Space Model.
#%%
def preprocess():
    doc_path = os.getcwd()
    docs_folder_path = os.path.join(doc_path, "collection", "docs")
    docs=[]

    for doc_id in os.listdir(docs_folder_path):
        docs.append(doc_id)
        file_path = os.path.join(docs_folder_path, doc_id)
        with open(file_path, 'r', encoding='utf-8') as file:
            for word in file:
                word = word.strip()
                if word not in stopwords:
                    stem = stemmer.stem(word)
                    iv.add_word(stem, doc_id)

    iv.print()
    count = len(docs)
    return set(map(int,docs)), count

doc_ids, doc_count = preprocess()
#%% md
- Query Preprocess

The queries from the collection need to be preprocessed in a similar way to
the documents. Each query is tokenized and the term frequency (TF) of each
token is calculated, in order to later be used by the VSM. This ensures
that the query representation aligns with the document representations,
enabling meaningful comparisons during similarity calculations.
#%%
```

```
def preprocess_querries():
    doc_path = os.getcwd()
    queries_folder_path = os.path.join(doc_path, "collection",
"Queries.txt")
    relevant_folder_path = os.path.join(doc_path, "collection",
"Relevant.txt")

    query_data = {}
    query_id = 1
    loaded_relevant_docs = []

    with open(queries_folder_path, encoding='utf-8') as queries_file:
        for line in queries_file:
            tokens = line.strip().split()
            processed_tokens = [word.lower() for word in tokens if
word.lower() not in stopwords]
            stemmed_tokens = [stemmer.stem(word) for word in
processed_tokens]
            tf = Counter(stemmed_tokens)
            total_terms = len(processed_tokens)
            query_data[f"Q{query_id}"] = {
                "stemmed_tokens": stemmed_tokens,
                "tf": tf,
                "total_terms": total_terms
            }
            query_id += 1

    with open (relevant_folder_path, encoding='utf-8') as relevant_file:
        for line in relevant_file:
            relevant_ids = list(map(int, line.strip().split()))
            filtered_relevant_ids = [doc_id for doc_id in relevant_ids if
doc_id in doc_ids]
            loaded_relevant_docs.append(filtered_relevant_ids)

    return query_data, loaded_relevant_docs

queryTokens, loadedRelevantDocs = preprocess_querries()
#%% md
- Boolean Model

The Boolean Model accepts logic statements as inputs. Therefore, the
collection queries need to be transformed. To do this, the tokens are
joined by an OR logical operator. The usage of the AND logical operator
would also be possible, but the amount of relevant documents retrieved by
the model would be significantly smaller. The OR operator assures that a
significant amount of documents will be retrieved, increasing with it the
possibility of less relevant documents to be selected. Whereas the AND
operator insures that only highly significant documents will be retrieved,
limiting the other possibly relevant documents and risking even an empty
selection.
#%%
def run_boolean():
    boolean_model = BooleanModel(iv.boolean_inv_index)
```

```
    boolean_results = []

    for query_id, query_data in queryTokens.items():
        tokens = query_data["stemmed_tokens"]
        boolean_query = " OR ".join(tokens)
        boolean_result = boolean_model.process_query(boolean_query)
        int_boolean_result = set(map(int, boolean_result))
        boolean_results.append(int_boolean_result)
    return boolean_results

booleanResults = run_boolean()
print("Boolean Model Results", booleanResults)
#%% md
- VSM

The Vector Space Model uses the inverted index and the term frequency to
calculate the TF-IDF for both the documents and the input queries. The
cosine similarity, between the  vector of the querys' token and each
document vector, is used to express the  relevance between query and
document. The documents are ranked from most to least relevant and only the
top k(=100) are retrieved.
#%%
def run_vsm(index, count, query_data):

    vsm = VectorSpaceModel(index, count)
    vsm.doc_tfidf()

    vsmResults = []
    for query_id, query_info in query_data.items():
        if not query_info:
            continue

        retrieved_list = vsm.search_tokens(query_info, top_k=100)
        ids = {int(doc_id_str) for doc_id_str, score in retrieved_list}
        vsmResults.append(ids)

    return vsmResults

vsm_results = run_vsm(iv.vs_inv_index, doc_count, queryTokens)
print("VSM Results: ", vsm_results)
#%% md
- Metrics

The precision and recall are calculated for each model. Although from the
Vector Space Model only the k(=100) most relevant documents are retrieved,
the Boolean Model retrieves all the significant documents, sence there is
no metric of significance, like cosine similarity. To better compare the
two models, only the first 100 retrieved documents -which in Boolean do not
represent a higher significance- are used in the computation of the
precision and recall.
#%%
def precision_recall_calculation(results, relevant_docs,k):
```

```
    all_precisions = []
    all_recalls = []

    for retrieved, relevant in zip(results, relevant_docs):
        top_k_results = list(retrieved)[:k]
        relevant_ids = []

        for j, result in enumerate(top_k_results):
            if result in set(relevant):
                relevant_ids.append(j)



        precisions = []
        recalls = []
        for num_relevant_id, relevant_id in enumerate(relevant_ids,
start=1):
            precision = num_relevant_id / (relevant_id + 1)
            recall = num_relevant_id / (len(relevant))
            precisions.append(precision)
            recalls.append(recall)

        all_precisions.append(precisions)
        all_recalls.append(recalls)

    return all_recalls, all_precisions

bool_recalls, bool_precisions =
precision_recall_calculation(booleanResults, loadedRelevantDocs,100)
vsm_recalls, vsm_precisions = precision_recall_calculation(vsm_results,
loadedRelevantDocs, 100)
print(bool_recalls, bool_precisions)
print(vsm_recalls, vsm_precisions)
#%%
def plot_precision_recall(precisions, recalls, method_name, num_queries):
    for i in range(num_queries):
        plt.figure(figsize=(8, 6))
        plt.plot(recalls[i], precisions[i], marker='o', label=f"Query {i +
1}")
        plt.xlabel("Recall")
        plt.ylabel("Precision")
        plt.title(f"Precision-Recall Curve for {method_name} - Query {i +
1}")
        plt.legend()
        plt.grid()
        plt.show()

plot_precision_recall(bool_recalls, bool_precisions, "Boolean", 20)
plot_precision_recall(vsm_recalls, vsm_precisions, "Boolean", 20)
```

Vector_Space_Model.py

```
import math


class VectorSpaceModel:

   def __init__(self, inverted_index, doc_count):


       self.inverted_index = inverted_index

       self.doc_count = doc_count

       self.documents = {}  # {doc_id: {term: tfidf_value}}




   def doc_tfidf(self):


       vs_index = self.inverted_index


       doc_term_freqs = {}


       for term, info in vs_index.items():

           for doc_id, freq in info["TF"].items():

               if doc_id not in doc_term_freqs:

                   doc_term_freqs[doc_id] = {}

               doc_term_freqs[doc_id][term] = freq


       #  TF-IDF for each term in every doc

       for doc_id, freqs in doc_term_freqs.items():

           max_freq = max(freqs.values())  # highest r(aw) frequency in
this doc
```

```
            self.documents[doc_id] = {}


            for term, freq in freqs.items():
                # tf=(freq / max freq)
                tf = freq / max_freq


                # idf=log(total_docs / DF)
                df = vs_index[term]["DF"]
                if df == 0 or self.doc_count == 0:
                    idf = 0
                else:
                    idf = math.log10(self.doc_count / df)


                self.documents[doc_id][term] = tf * idf


    def query_tfidf(self, query_info):


        stemmed_tokens = query_info["stemmed_tokens"]
        term_frequencies = query_info["tf"]
        total_terms = query_info["total_terms"]


        #  TF-IDF for each query term
        query_tfidf = {}
        for term in stemmed_tokens:
            tf = term_frequencies[term] / total_terms if total_terms > 0
else 0


            #  (IDF) query doesnt have DF so we use DF from the collection
of docs
```

```
            df = self.inverted_index.get(term, {}).get("DF", 0)
            idf = math.log10(self.doc_count / df) if df > 0 else 0

            # TF-IDF
            query_tfidf[term] = tf * idf

        return query_tfidf

    def cosine_similarity(self, doc_vector, query_vector):

        # cosine similarity between q and d

        numerator = 0.0
        for term, wq in query_vector.items():
            wd = doc_vector.get(term, 0.0)
            numerator += wd * wq

        # norms
        doc_norm = math.sqrt(sum(val ** 2 for val in doc_vector.values()))
        query_norm = math.sqrt(sum(val ** 2 for val in
query_vector.values()))

        if doc_norm == 0 or query_norm == 0:
            return 0.0

        return numerator / (doc_norm * query_norm)

    def search_tokens(self, query_data, top_k=10):
```

```
        # query vectors
        query_vec = self.query_tfidf(query_data)


        #  similarity for each doc
        scores = []
        for doc_id, doc_vector in self.documents.items():
            sim = self.cosine_similarity(doc_vector, query_vec)
            doc_id_stripped = doc_id.lstrip("0") or "0"
            #strips zeros(not the all 0 case)
            scores.append((doc_id_stripped, sim))


        scores.sort(key=lambda x: x[1], reverse=True)


        if top_k is not None and top_k < len(scores):
            scores = scores[:top_k]


        # results
        print("---Descending ranking by similarity---")
        for  doc_id_stripped, score in scores:
            print(f" {doc_id_stripped} | Score: {score:.4f}")


        return scores
```

Boolean_Model.py

```
class BooleanModel:
```

```
    def __init__(self, boolean_inv_index):
        self.boolean_inv_index = boolean_inv_index
```

```
    def process_query(self, query):

        tokens1 = query.lower().split()

        test = ""



        for token in tokens1:

            if token == "and":

                test += " & "

            elif token == "or":

                test += " | "

            elif token == "not":

                all_docs = set(doc_id for docs in
self.boolean_inv_index.values() for doc_id in docs)

                test += f"(all_docs - "

            else:

                matching_docs = self.boolean_inv_index.get(token, set())

                test += f"{matching_docs}"



        test = test.replace("all_docs - ", "(all_docs - ") + ")" if
"all_docs - " in test else test

        result_docs = eval(test)



        return result_docs
```

Inverted_Index.py

```
class InvertedIndex:
```

```
    def __init__(self):

        self.vs_inv_index = {}

        self.boolean_inv_index = {}
```

```
    def add_word_vs(self, word, doc_id):
        if word not in self.vs_inv_index:
            # Initialize word entry with document occurrence
            self.vs_inv_index[word] = {"DF": 1, "TF": {doc_id: 1}}
        else:
            if doc_id not in self.vs_inv_index[word]["TF"]:
                self.vs_inv_index[word]["DF"] += 1
                self.vs_inv_index[word]["TF"][doc_id] = 1
            else:
                self.vs_inv_index[word]["TF"][doc_id] += 1


    def add_word_boolean(self, word, doc_id):
        if word not in self.boolean_inv_index:
            self.boolean_inv_index[word] = {doc_id}
        else:
                self.boolean_inv_index[word].add(doc_id)


    def add_word(self, word, doc_id):
        self.add_word_vs(word, doc_id)
        self.add_word_boolean(word, doc_id)


    def print(self):
        print("vs_inv_index: ", self.vs_inv_index)
        print("boolean_inv_index: ", self.boolean_inv_index)
```