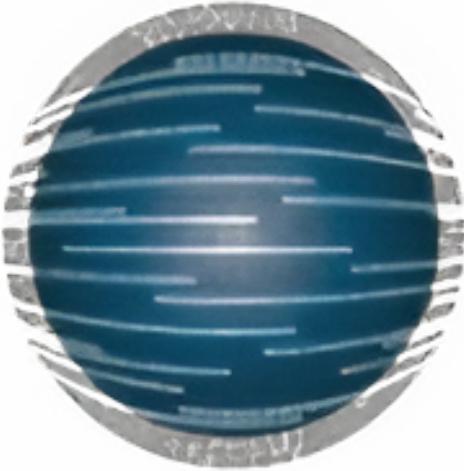


ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ, ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ  
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ



## Ανάκτηση Πληροφορίας

Χειμερινό Εξάμηνο

# Εργαστηριακή Άσκηση

Καθηγητές: Μακρής Χρήστος

ΑΜ	Επώνυμο	Όνομα	Έτος
1084567	Βιλιώτης	Αχιλλέας	4 <sup>ο</sup>
1084516	Μακρής	Ορέστης-Αντώνης	4 <sup>ο</sup>

# Περιεχόμενα

<b>Κεφάλαιο 1: Εισαγωγή</b>	<b>3</b>
1.a Παραδοτέα . . . . .	3
1.b Δομή αρχείων . . . . .	3
1.c Χρήση . . . . .	3
1.c.i VSM . . . . .	3
1.c.ii ColBERT . . . . .	4
<b>Κεφάλαιο 2: Υλοποίηση Vector Space Model</b>	<b>5</b>
2.a Εισαγωγή . . . . .	5
2.b Διαθέσιμες κλάσεις και μέθοδοι . . . . .	5
2.c Inverted Index . . . . .	5
2.d Vector Space Model . . . . .	5
<b>Κεφάλαιο 3: Υλοποίηση colBERT</b>	<b>7</b>
3.a Εισαγωγή . . . . .	7
3.b Διαθέσιμες κλάσεις και μέθοδοι . . . . .	7
<b>Κεφάλαιο 4: Προεπεξεργασία Δεδομένων και Αποτελέσματα</b>	<b>10</b>
4.a Προεπεξεργασία για το VSM . . . . .	10
4.b Προεπεξεργασία για το colBERT . . . . .	10
4.c Μετρικές και Αποτελέσματα . . . . .	10
4.d Συμπεράσματα . . . . .	13
4.e Προτάσεις Επέκτασης της Εργασίας: περαιτέρω έρευνα . . . . .	14
<b>Κεφάλαιο 5: Παράρτημα</b>	<b>15</b>
5.a Config.py Python File Code . . . . .	15
5.b fun.PY library Code . . . . .	15
5.c Main VCM Reverse Indexing Code . . . . .	27
5.d ColBERT Code . . . . .	28
5.e Metrics Visualization Code . . . . .	38
<b>Κεφάλαιο 6: Βιβλιογραφία</b>	<b>42</b>



## Κεφάλαιο 1: Εισαγωγή

Στην τεχνική αναφορά που ακολουθεί, παρουσιάζεται η εργασία πάνω στην υλοποίηση ενός Vector Space Μοντέλου σε python, με έναν δικό μας συνδυασμό TF-IDF βάσει του δωθέντος άρθρου [5] καθώς και η εκμετάλλευση του ColBERT μοντέλου και η σύγκριση μεταξύ τους.

Στοιχεία επικοινωνίας email:

Βιλλιώτης Αχιλλέας : up1084567@ac.upatras.gr

Μακρής Ορέστης-Αντώνης : up1084516@ac.upatras.gr

### 1.a Παραδοτέα

Για την εργαστηριακή άσκηση του μαθήματος Ανάκτηση Πληροφορίας για το χειμερινό εξάμηνο 2023 – 2024 έχουν ολοκληρωθεί όλα τα ζητούμενα ερωτήματα. Τα παραδοτέα περιλαμβάνουν όλα τα αρχεία του κώδικα που αναπτύχθηκαν στα πλαίσια τις άσκησης καθώς και όλα τα αρχεία outputs που παράχθηκαν από τα αποτελέσματα τις λειτουργίας των δύο μηχανών αναζήτησης csv και γραφικές παραστάσεις απαραίτητα για την σύγκριση τους .

### 1.b Δομή αρχείων

Ο root φάκελος περιέχει τα εξής αρχεία.

- README.md: Περιλαμβάνει απαραίτητες πληροφορίες εκτέλεσης του κώδικα.
- data/: Στον φάκελο αυτό περιλαμβάνονται όλα τα αρχεία που είναι απαραίτητα για την αξιολόγηση των μηχανών αναζήτησης που δόθηκαν από την εκφώνηση του Project, καθώς και τα επιπλέον αρχεία που παραδόθηκαν από το K. Καλογερόπουλο που περιλαμβάνουν 100 queries.
- docs/: περιλαμβάνει την συλλογή που έχει δοθεί Cystic Fibrosis (C.F) και περιλαμβάνει 1209 κείμενα.
- source/: Περιλαμβάνει τους κώδικες που αναπτύχθηκαν στα πλαίσια της εργασίας. Όλα τα αρχεία, πέρα του notebook, αφορούν το Vector Space Μοντέλο.
- paper\_analysis: Σύντομη παρουσίαση του σκεπτικού πίσω από την επιλογή των συντελεστών στο VSM.
- metrics\_visualization{20, 100}/: Περιέχουν εικόνες των μετρικών.
- outputs/: Φάκελος προορισμού για τα αποτελέσματα των μοντέλων.

### 1.c Χρήση

#### 1.c.i VSM

Προσοχή! Το VSM πρέπει να καλείται από τον root φάκελο του πρότζεκτ!

Παράδειγμα χρήσης:

- python source/main.py n f c n f x
- python source/main.py t f c n f x
- python source/main.py t x c n f x



### 1.c.ii ColBERT

Για την εκτέλεση του ColBert απαιτείται η χρήση Google Colaboratory με gpu Connection , και είναι απαραίτητο να δοθούν στον κώδικα τα απαριτητά File Paths των αρχείων για την ανάγνωση των δεδομένων. Για το Indexing των docs απαιτούνται περίπου 5-8 λεπτα.



## Κεφάλαιο 2: Υλοποίηση Vector Space Model

### 2.a Εισαγωγή

Η υλοποίηση έγινε εξ' ολοκλήρου σε Python3, χωρίς εξωτερικές βιβλιοθήκες. Σε όλο το πλάτος της υλοποίησης ακολουθήθηκε το "Google Python Style Guide" [2] ώστε να δημιουργηθεί μια πλήρης βιβλιοθήκη η οποία μπορεί να χρησιμοποιηθεί εύκολα και ανεξάρτητα από οποιονδήποτε τρίτο. Επίσης έγινε χρήση τεχνικών list-comprehension [4] καθώς παρατηρήθηκε σημαντική αύξηση ( $>30\%$ ) στην απόδοση σε σχέση με βρόγχους η map-reduce τεχνικές.

Η βιβλιοθήκη αποτελείται από δύο αρχεία, τα `fun.py` και `config.py`. Το πρώτο περιέχει όλες τις συναρτήσεις και μεθόδους, εξωτερικές και εσωτερικές που απαιτούνται για τη χρήση του VSM, ενώ το `config` περιέχει μερικές ρυθμίσεις όσον αφορά την συμπεριφορά του μοντέλου και το logging.

Η βιβλιοθήκη περιέχει 4 διαφορετικούς τρόπους υπολογισμών των βαρών TF-IDF καθώς και δυνατότητα για υπολογισμό μετρικών για ερωτήματα.

### 2.b Διαθέσιμες κλάσεις και μέθοδοι

Η βιβλιοθήκη παρέχει στον χρήστη 2 κλάσεις, την `InvertedIndex` και `VectorSpaceModel`.

Κατά την δημιουργία του `InvertedIndex` αντικειμένου, εισάγεται ο φάκελος των εγγράφων. Στην συνέχεια παρέχεται η μέθοδος `create_index` η οποία δημιουργεί το ανεστραμμένο αρχείο.

Κατά τη δημιουργία του `VectorSpaceModel` αντικειμένου, χρειάζεται να δοθούν το ανεστραμμένο αρχείο που δημιουργήθηκε πριν καθώς και οι παράμετροι που ορίζουν τον τρόπο υπολογισμού των TF-IDF για τα έγγραφα και τα ερωτήματα. Παρέχονται οι μέθοδοι `create_document_term_matrix` η οποία καθιστά το VSM λειτουργικό και `query` η οποία χρησιμοποιείται ώστε να ερωτηθεί το μοντέλο για κάποιο ερώτημα και να επιστρέψει την κατάταξη των εγγράφων.

Τέλος υπάρχουν δύο συναρτήσεις, οι `query_ranking` και `query_metrics`, οι οποίες χρησιμοποιούνται με έντελο το `vsm` και παρέχουν στον χρήστη μορφοποιημένη έξοδο για τις μετρικές και τα σχετικά έγγραφα, ανάλογα τις ρυθμίσεις του χρήστη configuration αρχείο.

### 2.c Inverted Index

Η υλοποίηση του Inverted Index είναι απλή, για κάθε έγγραφο στον φάκελο που δόθηκε ως όρισμα κατά την αρχικοποίηση του αντικειμένου, κάθε λέξη προστίθεται στο ανεστραμμένο αρχείο μέσω της εσωτερικής μεθόδου `_add_occurrence`, η οποία χειρίζεται το εμφωλευμένο λεξικό `_index`.

Το λεξικό έχει την μορφή λεξικού με κλειδί την λέξη και τιμή ένα δεύτερο λεξικό που έχει ως κλειδί το κείμενο και τιμή μία λίστα από ακεραίους που δείχνουν την θέση στο κείμενο. Η λίστα δεν χρησιμοποιήθηκε στο πρότζεκτ, όμως χρατήθηκε για λόγους πληρότητας, ενώ δεν χρησιμοποιήθηκε κάποια κωδικοποίηση της θέσης στο κείμενο επειδή είχαμε μικρά σε μήκος κείμενα και η Python έχει ήδη αστικοποιημένα αντικείμενα για τους ακέραιους [-8,256].

### 2.d Vector Space Model

Μετά την αρχικοποίηση του αντικειμένου `VectorSpaceModel`, χρειάζεται να δημιουργηθεί το document-term matrix το οποίο χρησιμοποιείται για τα ερωτήματα. Εσωτερικά της μεθόδους αξιοποιείται η μέθοδος `_calculate_tf_idf` η οποία καλεί με την σειρά την κατάλληλη μέθοδο υπολογισμού των TF-IDF, βάσει των παραμέτρων του μοντέλου.



Τπάρχουν συνολικά 4 υλοποιημένες μέθοδοι που μπορούν να χρησιμοποιηθούν ελεύθερα από τον χρήστη για τον υπολογισμό των βαρών TF-IDF, οι `_t_x_c_calculation`, `_t_f_c_calculation`, `_n_f_x_calculation` και `_n_f_c_calculation`.

Μέσω αυτών, επιλέχθηκαν 2+1 τρόποι υλοποίησης του μοντέλου, βάσει του δοθέντος paper κατά την εκφώνηση της άσκησης [5]. Δύο από αυτούς αναφέρονται ρητά στο paper, ενώ ο τρίτος προέκυψε από ανάλυση των ευρημάτων στο τέλος αυτού.

Οι συνδυασμοί `tfc-nfx` και `txc-nfx` επιλέχθηκαν καθώς λειτουργούσαν καλά σε γενική βάση.

Ο `tfx-tfx` απορρίφθηκε καθώς δεν έπαιζε καλά για μικρού μήκους queries.

Ο `nxx-bpx` ήταν τρίτος στη σειρά προς επιλογή όμως εφόσον ζητούμενο ήταν ήδη δύο μόνο συνδυασμοί, αποφασίσαμε να επιλέξουμε την συνδυαστική υλοποίηση. Οι υπόλοιποι δεν επιλέχθηκαν καθώς ορίστηκαν κατώτεροι από την έρευνα.

Για την συνδυαστική, επιλέχθηκε ο `nfc-nfx`, καθώς έχουμε μικρού μήκος ερώτημα ο παράγοντας `n`, ο παράγοντας `f` προτείνεται γενικά όπως και ο `x` στην κανονικοποίηση. Για τα έγγραφα λόγω των τεχνικών όρων στα κείμενα, επιλέχθηκε ο παράγοντας `n`, ο παράγοντας `f` καθώς προτείνεται και επειδή υπάρχουν μεγάλες διαβαθμίσεις στα μήκη των κειμένων επιλέχθηκε κανονικοποίηση.

Σε περίπτωση επιλογής συνδυασμού παραμέτρων που δεν έχει υλοποιηθεί εμφανίζεται μήνυμα κατά την δημιουργία του αντικειμένου. Επίσης κατάλληλο μήνυμα εμφανίζεται εάν ο χρήστης επιλέξει βάρη τα οποία δεν προβλέπονται/έχουν ελεγχθεί.



## Κεφάλαιο 3: Υλοποίηση colBERT

### 3.a Εισαγωγή

Η υλοποίηση έγινε εξ' ολοκλήρου σε Python3.7 ++ στο περιβάλλον ανάπτυξής Google Colaboratory [1] και με την χρήση της default available gpu T4 NVIDIA, χωρίς εξωτερικές βιβλιοθήκες, πέρα από την χρήση της απαραίτητη της βιβλιοθήκης ColBERT (v2) υλοποίησης του Stanford [7][6] [3]. Η οποία μας προσφέρει έτοιμα τα βασικά components του ColBERT indexer και Searcher. Σε όλο το πλάτος της υλοποίησης ακολουθήθηκε το "Google Python Style Guide" [2] ώστε να δημιουργηθεί μια εναρμόνιση με τον κώδικα που αναπτύχθηκε για το vsm / reverse index ερώτημα πρώτο δεύτερο της εργασίας, πράγμα που καθιστά πιο κατανοητό των κώδικα από οποιονδήποτε τρίτο.

Το περιβάλλον ανάπτυξης google Colaboratory, επιλέχθηκε για λόγους πρακτικότητας και ευκολίας. Οι βασικοί λόγοι περιλαμβάνουν πρόσβαση σε δωρεάν GPU (τα ατομικά υπολογιστικά συστήματα στα οποία αναπτύχθηκε η εργασία δεν διαθέτουν πρόσβαση σε cuda enabled gpu) και η εκτέλεση της έκδοσης της βιβλιοθήκης για CPU-only environments σε σύστημα windows είναι αρκετά απαιτητική. Επίσης εκτέλεση του indexing παίρνει σαφώς μεγαλύτερο χρόνο σε cpu από gpu. [6] [3]

Ο κώδικας αποτελείται από 5 κομμάτια πρώτο αρχικοποίησή του πειράματος στο περιβάλλον του Google Colaboratory (εγκατάσταση των απαραίτητων βιβλιοθηκών), δεύτερο την κλάση FileHandlerData η κλάση αυτή είναι παρέχει στο πρόγραμμα όλες της απαραίτητες συνάρτησης για την ανάγνωσή των αρχείων που εμπεριέχουν τα δεδομένα και την παραγωγή του αρχείου εξόδου csv. Τρίτο την κλάση ColBERTSearchEngine αρχικοποιείτε η μηχανή αναζήτησης indexing και παρέχει στον χρήστη την δεινότητα αναζήτησης στην μηχανή, τέταρτο την κλάση EvaluationMetrics περιεχει όλες τις μετρικές που χρησιμοποιήσαμε για την αξιολόγηση της μηχανής αναζήτησης, πέμπτο συνάρτηση main.

### 3.b Διαθέσιμες κλάσεις και μέθοδοι

Η κλάση FileHandlerData δημιουργήθηκε για τη διαχείριση αρχείων και την παροχή δεδομένων εκπαίδευσης και δοκιμής. Παρέχει μεθόδους για την ανάγνωση κειμένου από αρχεία, την ανάγνωση ερωτήσεων από αρχείο, την ανάγνωση βαθμολογιών σημαντικότητας για κάθε ερώτηση από αρχείο, καθώς και για την εξαγωγή μετριών απόδοσης της μηχανής αναζήτησης σε ένα αρχείο CSV.

Ας κάνουμε μια αναλυτική εξήγηση κάθε μεθόδου :

`__init__(self, folder_path):`

Ο κατασκευαστής (constructor) που αρχικοποιεί το αντικείμενο με τη διοθείσα διαδρομή του folder που περιέχει τα docs.

`read_dock_files(self):`

Αναγνωστική μέθοδος που διαβάζει τα κείμενα από τα αρχεία στον φάκελο και τα επιστρέφει ως λίστα.

`read_queries_from_file(self, file_path):`

Μέθοδος που διαβάζει ερωτήσεις από ένα αρχείο και τις επιστρέφει ως λίστα.

`read_relevance_scores_from_file(self, file_path):`

Μέθοδος που διαβάζει τις βαθμολογίες σημαντικότητας για κάθε ερώτηση από ένα αρχείο και τις επιστρέφει ως λεξικό.



```
export_metrics(self, query_metrics, csv_file_path):
```

Μέθοδος που εξάγει τις μετρικές απόδοσης σε ένα αρχείο CSV.

```
export_metrics:
```

Η μέθοδος εξάγει τα αποτελέσματα σε ένα αρχείο CSV με κεφαλίδες και αντίστοιχες τιμές για κάθε μετρική.

Η κλάση ColBERTSearchEngine χρησιμοποιείται για τη δημιουργία ενός συστήματος αναζήτησης με χρήση του μοντέλου ColBERT (V2).

noindent Αναλυτική εξήγηση της κλάσης:

noindent Attributes:

```
index_name (str):
```

Το όνομα του ευρετηρίου για την αναζήτηση.

```
dock_file_texts (list):
```

Λίστα που περιέχει τα κείμενα των εγγράφων για τη δημιουργία του ευρετηρίου.

Μέθοδοι:

```
__init__(self, index_name, dock_file_texts):
```

Ο κατασκευαστής (constructor) που αρχικοποιεί το αντικείμενο με το όνομα του ευρετηρίου και τα κείμενα των εγγράφων.

```
create_index(self, doc maxlen, nbits):
```

Μέθοδος που δημιουργεί ένα ευρετήριο χρησιμοποιώντας τα δοιθέντα κείμενα εγγράφων, το ευρετήριο χρησιμοποιείτε για την πιο αποδοτική αναζήτηση και αντιστοίχιση εγγράφων. [6]

```
search_queries(self, queries, _number_):
```

Μέθοδος που αναζητά για ερωτήσεις στο ευρετήριο και επιστρέφει τα αποτελέσματα της αναζήτησης, η μέθοδος εκτελεί αναζήτηση για queries στο ευρετήριο και επιστρέφει τα κορυφαία K αποτελέσματα. Τα αποτελέσματα επιστρέφονται σε ένα λεξικό όπου οι κλειδιά είναι οι δείκτες των ερωτήσεων και οι τιμές είναι λίστες που περιέχουν τα τρία στοιχεία (Query, Passage ID, Score ταιριάσματος , rank).

Η κλάση EvaluationMetrics παρέχει τις στατικές μεθόδους για τον υπολογισμό μετρικών απόδοσης και αξιολόγησης για το σύστημα ανάζησης πληροφοριών (information retrieval) βασισμένο στο ColBERT.

Μεθόδους:

```
calculate_confusion_matrix(retrieved_docs, relevant_docs):
```

Υπολογίζει τον πίνακα σύγχυσης για δυαδική κατηγοριοποίηση. Επιστρέφει τις τέσσερις βασικές μετρικές: True Positives (TP), False Positives (FP), False Negatives (FN), True Negatives (TN).

```
calculate_precision_recall(retrieved_docs, relevant_docs):
```



Τυπολογίζει την ακρίβεια και την ανάληση. Χρησιμοποιεί την calculate confusion matrix για του υπολογισμό των TP, FP, FN. Επιστρέφει την ακρίβεια και την ανάληση.

`calculate_f1_score(precision, recall):`

Τυπολογίζει το F1-score χρησιμοποιώντας την ακρίβεια και την ανάληση.

`calculate_ap(results_dict, relevance_scores):`

Τυπολογίζει το Average Precision (AP) για τα αποτελέσματα της αναζήτησης αξιοποιώντας τα δεδομένα σχετικότητας από το αρχείο relevance. Επιστρέφει ένα λεξικό με τα AP για κάθε ερώτηση.

`calculate_rr(query_results, relevance_scores):`

Τυπολογίζει το Reciprocal Rank (RR) για τα αποτελέσματα της αναζήτησης αξιοποιώντας τα δεδομένα σχετικότητας από το αρχείο relevance Επιστρέφει ένα λεξικό με τα RR για κάθε ερώτηση.

Η main αναλαμβάνει την εκτέλεση του κώδικα και περιλαμβάνει τα εξής βήματα:

Ανάγνωση Αρχείων: Καθορίζονται οι διαδρομές των φακέλων και των αρχείων που περιέχουν τα κείμενα των εγγράφων, τις ερωτήσεις και τις αξιολογήσεις των εγγράφων.

Ορίζονται παράμετροι για το ColBERT, όπως τον αριθμό των κορυφαίων αποτελεσμάτων αναζήτησης (k number), το μέγιστο μήκος εγγράφου (doc maxlen) και τον αριθμό των nbits για την κωδικοποίηση των διαστάσεων.

Δημιουργία Ευρετηρίου (Index):

Χρησιμοποιείται η κλάση ColBERTSearchEngine για τη δημιουργία του ευρετηρίου (index). Εκτελείται η αναζήτηση για κάθε ερώτηση και αποθηκεύονται τα αποτελέσματα.

Τυπολογισμός Μετρικών Αξιολόγησης: Χρησιμοποιείται η κλάση EvaluationMetrics για τον υπολογισμό των μετρικών αξιολόγησης (Precision, Recall, F1-score, Average Precision, Reciprocal Rank) για κάθε ερώτηση.

Τα αποτελέσματα αποθηκεύονται σε ένα CSV αρχείο με το όνομα

`"querymetricsoutput.csv"`.

Το σενάριο φαίνεται να λειτουργεί όπως αναμένεται, εκτελώντας τις απαραίτητες διαδικασίες για τον ColBERT και υπολογίζοντας τις μετρικές αξιολόγησης.



## Κεφάλαιο 4: Προεπεξεργασία Δεδομένων και Αποτελέσματα

Και για τα δύο μοντέλα, τα αποτελέσματα περιορίστηκαν στα πρώτα 400 κείμενα.

### 4.a Προεπεξεργασία για το VSM

Κάποια query διορθώθηκαν, διότι χρησιμοποιούσαν τον χαρακτήρα "-", ενώ στα κείμενα ο χαρακτήρας παραβλέποταν, και ανάποδα.

### 4.b Προεπεξεργασία για το colBERT

Δεν εφαρμόστηκαν τεχνικές προ επεξεργασίες δεδομένων πέρα από το ότι κατά ανάγνωση των docs files στα missing docs μπήκαν οι τιμές Nal στα αντίστοιχα index. Τεχνικές όπως Text Cleaning, Tokenization, Stopword Removal, Stemming or Lemmatization θα οδηγήσουν στην απώλεια χρήσιμης πληροφορίας χωρίς κάποια οστική βελτίωση στον χρόνο λειτουργίας των μηχανών αναζήτησης.

### 4.c Μετρικές και Αποτελέσματα

Για την αξιολόγηση και την σύγκριση των δύο διαφορετικών συστημάτων ανάκτησης πληροφορίας επιλέχθηκαν οι ακόλουθες μετρικές :

- Ακρίβεια (Precision): Η ακρίβεια είναι ο λόγος των πραγματικών θετικών προβλέψεων προς τον συνολικό αριθμό των θετικών προβλέψεων που κάνει το μοντέλο. Στο πλαίσιο ενός μηχανισμού αναζήτησης, μετράει πόσο ακριβές είναι το μοντέλο όταν προβλέπει ότι ένα στιγμιότυπο ανήκει σε μια συγκεκριμένη κατηγορία. Υψηλή ακρίβεια σημαίνει λιγότερα φευδή θετικά.

- Ανάκληση (Recall): Η ανάκληση είναι ο λόγος των πραγματικών θετικών προβλέψεων προς τον συνολικό αριθμό των πραγματικών θετικών περιστατικών. Στο πλαίσιο ενός μηχανισμού αναζήτησης, μετράει την ικανότητα του μοντέλου να εντοπίζει όλα τα πραγματικά θετικά παραδείγματα σε μια συγκεκριμένη κατηγορία. Υψηλή ανάκληση σημαίνει λιγότερα φευδή αρνητικά.

1-σκορ (F1-score): Το F1-σκορ είναι ο αρμονικός μέσος της ακρίβειας και της ανάκλησης. Παρέχει έναν ισορροπημένο δείκτη μεταξύ των δύο μετρικών.

- Πραγματικά Θετικά (True Positives - TP): Ο αριθμός των περιπτώσεων όπου ο μηχανισμός αναζήτησης αναγνώρισε σωστά σχετικά έγγραφα.
- Ψευδή Θετικά (False Positives - FP): Ο αριθμός των περιπτώσεων όπου ο μηχανισμός αναζήτησης αναγνώρισε λανθασμένα μη σχετικά έγγραφα ως σχετικά.
- Μέση Ακρίβεια Περιοχής (Mean Average Precision - MAP): Η μέση ακρίβεια περιοχής (AP) είναι ένα μέτρο που λαμβάνει υπόψη την ακρίβεια σε διάφορα επίπεδα ανάκλησης. Το MAP είναι ο μέσος όρος του AP σε κάθε query.
- Μέσος Αντίστροφος Ρυθμός Κατάταξης (Mean Reciprocal Rank - MRR): Ο MRR είναι ένα μέτρο που λαμβάνει υπόψη τη θέση του πρώτου σωστού αποτελέσματος. Χρησιμοποιήθηκε για την αξιολόγηση την αποτελεσματικότητα των δύο διαφορετικών συστημάτων στο ranking.

Συνολικά, ο παρεχόμενος κώδικας εκτυπώνει ένα σύνολο σφαιρικών μετρικών αξιολόγησης για να αξιολογηθεί πιο ορθά η απόδοση ενός μηχανισμού αναζήτησης.



Μετρική	VSM (nfc nfx)	VSM (tfc nfx)	VSM (txc nfx)
Mean Precision	0.0431	0.0391	0.0335
Mean Recall	0.4004	0.3673	0.2970
Mean F1-score	0.0722	0.0653	0.0554
Mean True Positives	17.25	15.65	13.4
Mean False Positives	381.75	383.35	385.6
Mean Average Precision (MAP)	0.0661	0.0582	0.0436
Mean Reciprocal Rank (MRR)	0.2124	0.1444	0.1134

Table 1: Σύγκριση Μετρικών για **20 Queries** διαφορετικών υλοποιήσεων **VSM**

Μετρική	VSM (nfc nfx)	VSM (tfc nfx)	VSM (txc nfx)
Mean Precision	0.0485	0.0453	0.0366
Mean Recall	0.4059	0.4013	0.3173
Mean F1-score	0.0800	0.0748	0.0600
Mean True Positives	19.43	18.15	14.68
Mean False Positives	380.57	381.85	385.32
Mean Average Precision (MAP)	0.0845	0.0813	0.0570
Mean Reciprocal Rank (MRR)	0.3342	0.2922	0.2314

Table 2: Σύγκριση Μετρικών για **100 Queries** διαφορετικών υλοποιήσεων **VSM**

Μετρική	ColBert	VSM (nfc nfx)
Mean Precision	0.0685	0.0431
Mean Recall	0.7125	0.4004
Mean F1-score	0.1180	0.0722
Mean True Positives	28.0	17.25
Mean False Positives	372.0	381.75
Mean Average Precision (MAP)	0.2290	0.0661
Mean Reciprocal Rank (MRR)	0.8015	0.2124

Table 3: Σύγκριση Μετρικών για 20 Ερωτημάτων: ColBert έναντι VSM (nfc nfx)

Metric	VSM (nfc nfx)	ColBert
Mean Precision	0.0485	0.0771
Mean Recall	0.4059	0.6901
Mean F1-score	0.0800	0.1279
Mean True Positives	19.43	30.79
Mean False Positives	380.57	368.11
Mean Average Precision (MAP)	0.0845	0.2362
Mean Reciprocal Rank (MRR)	0.3342	0.7912

Table 4: Comparison of Metrics for Search Engine 100 Queries: VSM (nfc nfx) vs ColBert

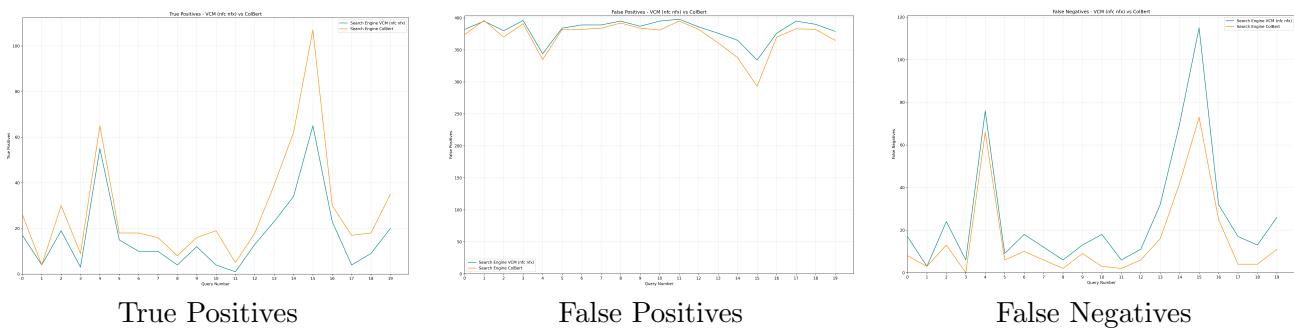


Table 5: Comparison graphs of Metrics for 100 Queries: VSM (nfc nfx) Blue vs ColBERT Orange

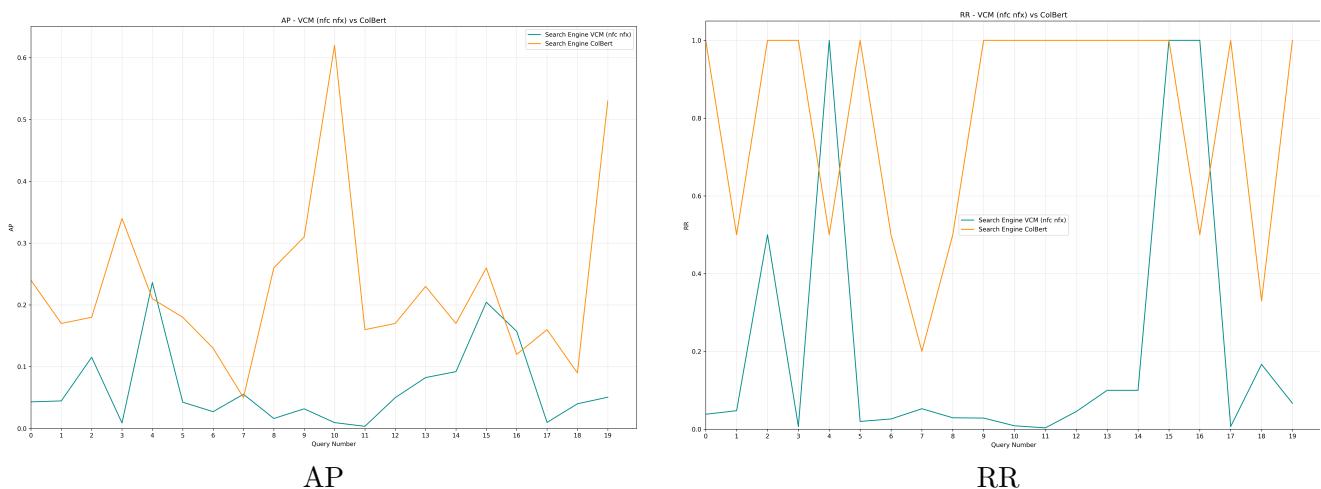


Table 6: Comparison graphs of Metrics for 100 Queries: VSM (nfc nfx) Blue vs ColBERT Orange

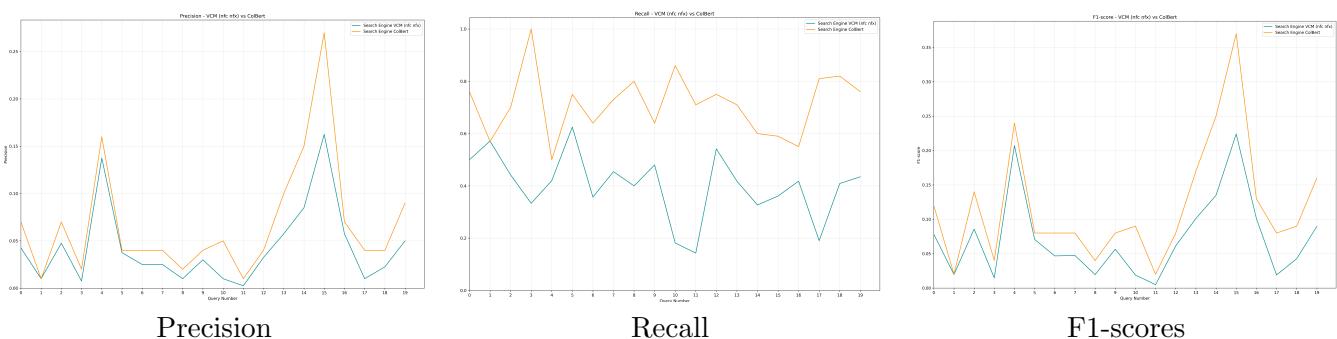


Table 7: Comparison graphs of Metrics for 100 Queries: VSM (nfc nfx) Blue vs ColBERT Orange

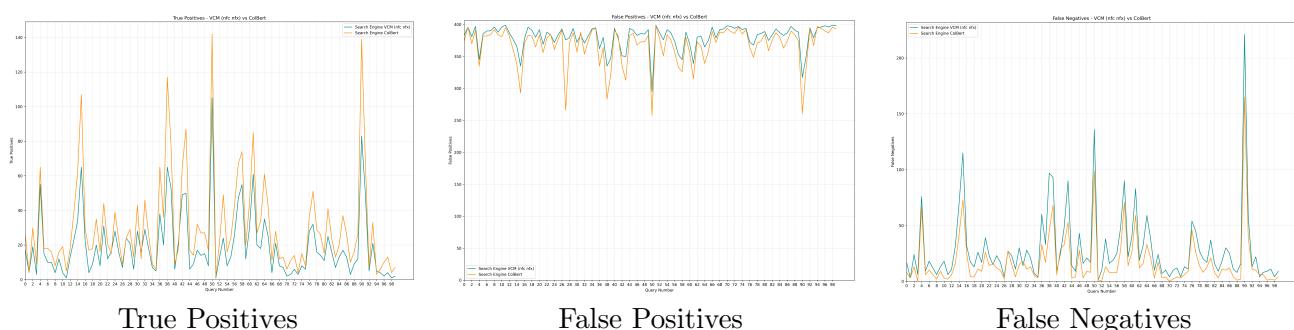


Table 8: Comparison graphs of Metrics for 20 Queries: VSM (nfc nfx) Blue vs ColBERT Orange

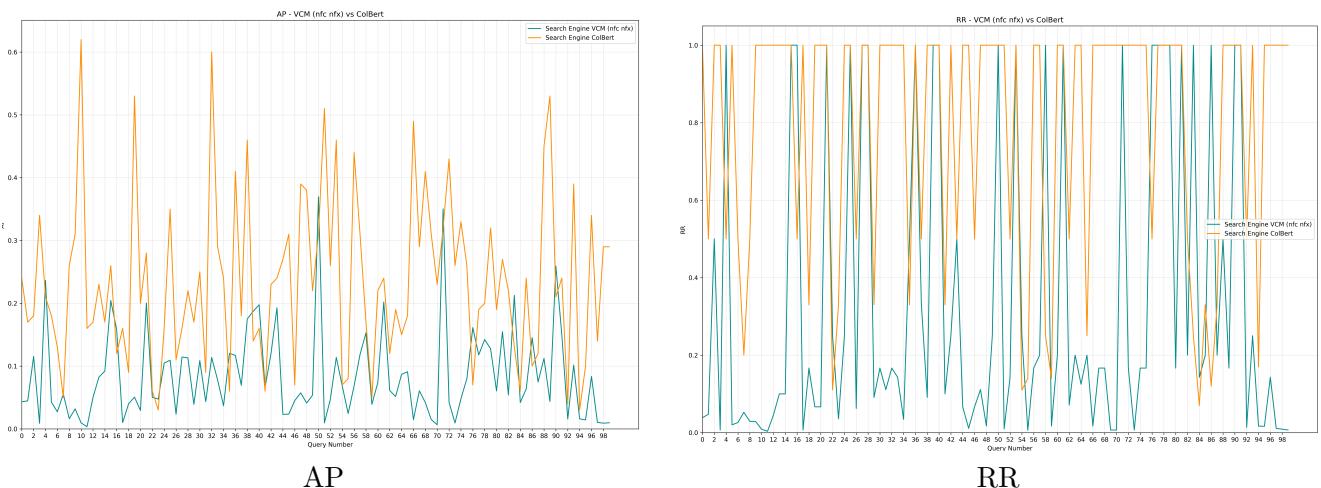


Table 9: Comparison graphs of Metrics for 20 Queries: VSM (nfc nfx) Blue vs ColBert Orange

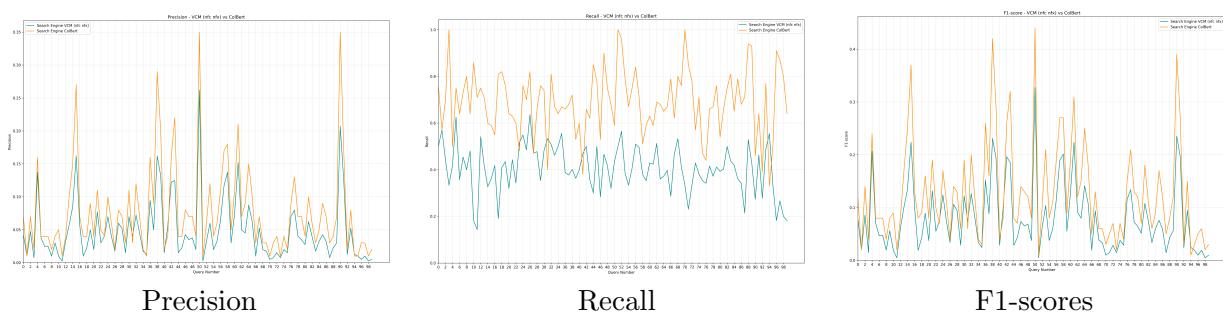


Table 10: Comparison graphs of Metrics for 20 Queries: VSM (nfc nfx) Blue vs ColBert Orange

#### 4.d Συμπεράσματα

Παρατηρούμε πως κατά μέσο όρο, το ColBERT υπερτερεί του VSM μας, όσον αφορά την ανάκτηση των 400 καλύτερων κειμένων. Επίσης ο χαμήλος δείκτης RR, φανερώνει πως η χρήση του σαν μηχανή αναζήτησης λίγω αποτελεσμάτων επίσης δεν θα ήταν αποτελεσματική. Υπήρξαν κάποια query για τα οποία το VSM "έτρεξε" καλύτερα, αλλά είναι αναμενόμενο και προφανώς αποτελούν την μειοψηφία.

Επίσης εξάγουμε το συμπέρασμα ότι το cobert θα αποτελούσε ένα καλό rerankers για ένα σύστημα ανάκτησής πληροφορίας όπως το vcm ή BM25 καθώς εκφράζει την μεγαλύτερή βελτίωση ως προς την μετρική του RR για κάθε Querry.

Η κακή απόδοση του vcm σε ένα βαθμό είναι αναμενομένη διότι γνωρίζουμε από τα πειράματα των Salton, Buckley τα οποία αποδεικνύουν ότι, το μοντέλο διανυσματικού χώρου είναι καλύτερο σε απόδοση από άλλα μοντέλα. Ωστόσο στο πείραμά της εργασίας ανακτούμε πληροφορία σε ένα πολύ τεχνικό iatricό dataset , το οποίο προφανώς δεν γενικεύει.

Βέβαια, καταφέραμε να βρούμε έναν πιο αποτελεσματικό τρόπο υλοποίησης του VSM πάνω στα δεδομένα μας, σε σχέση με την βιβλιογραφία [5] . Συγκεκριμένα, τα βάρυ tf-idf για document-query "nfc-nfx", ήταν πολύ πιο αποτελεσματικά, από τα δύο προτεινόμενα του άρθρου για γενική χρήση, "tfx-nfx" και "txc-nfx".



#### 4.e Προτάσεις Επέκτασης της Εργασίας: περαιτέρω έρευνα

Εφαρμογή ensembling τεχνικών μηχανικής μάθησης για την επιλογή του πιο optimal μοντέλου στο οποίο θα θέσουμε το query. Αρχετά δύσκολο να εφαρμοστεί στη δοσμένη συλλογή, αρκετά τεχνικό και λίγα queries για εκπαίδευση του ensemble.

Χρήση ColBERT για reranking στα αποτελέσματα ενός ερωτήματος τα οποία μας δίνει ένα μοντέλο όπως το BM25. (ColBERT είναι πολύ αποτελεσματικό σε μικρό αριθμό δεδομένων).

Τεχνικές μηχανικής μάθησης μπορούν να εφαρμοστούν για την πιο εξεζητημένη εύρεση τιμών  $g(d) =$  μέτρο ποιότητας σε τεχνικές Στάικων βαθμολογίων ποιότητας και διάταξης



## Κεφάλαιο 5: Παράρτημα

### 5.a Config.py Python File Code

```
1 global PRINT_OUTPUTS
2 global EXPORT_OUTPUTS
3 global LIMIT_RESULTS
4
5 PRINT_OUTPUTS = False
6 EXPORT_OUTPUTS = True
7 LIMIT_RESULTS = 401
8 INPUT_SIZE = 20
```

Listing 1: config.py

### 5.b fun.PY library Code

```
1 """Module containing everything needed for the implementation project
2 of course CEID1037 - Information Retrieval 2023-2024
3
4 Typical usage example:
5     inv_index = InvertedIndex("docs/")
6     inv_index.create_index()
7
8     vsm = VectorSpaceModel(inv_index, 'n', 'f', 'c', 'n', 'f', 'x')
9     vsm.create_document_term_matrix()
10
11    query = ("What are the effects of calcium on the physical properties of
12
13        "mucus from CF patients")
14    query = query.upper().split()
15    relevant = ("00139 00151 00166 00311 00370 00392 00439 00440 00441 00454
16
17        "
18
19        "00461 00502 00503 00505 00520 00522 00526 00527 00533 00593 00619 00737
20
21        "
22
23        "00742 00789 00827 00835 00861 00875 00891 00921 00922 01175 01185
24
25        01222")
26    relevant = relevant.split()
27
28    similarity = vsm.query(query)
29    pairs = zip(vsm.inverted_index.document_list, similarity)
30
31    for i, doc in enumerate(sorted_pairs):
32        if doc[0] in relevant:
33            print("%s (%f) in position %d" % (doc[0], doc[1], len(sorted_pairs)-
34                i))
35
36"""
37
38 from config import PRINT_OUTPUTS, EXPORT_OUTPUTS, LIMIT_RESULTS
39
40 from math import log10, sqrt
```



```
29 import os
30 from typing import Literal
31
32
33 class InvertedIndex:
34     """Class for creating and accessing an inverted index
35
36     Allows creating an inverted index in bulk via the createIndex() method,
37     which
38     utilizes the documentDirectory given during initialization. Uses
39     defaultdict
40     with default Nonetype instead of normal dictionaries for cleaner code.
41
42     Properties (read-only):
43         index: Dictionary containing the inverted index. The index is
44         structured as follows:
45             Each value is itself a dictionary, with the key being the
46             document
47                 name and the value being the list of positions in the document
48             where
49                 the word occurs.
50
51             document_list: List containing the names of all documents.
52             document_directory: Path to the directory containing the documents.
53
54     """
55
56     def __init__(self, document_directory: str):
57         """Initializes the InvertedIndex object
58
59         Args:
60             document_directory: Path to the directory containing the
61             documents,
62                 can include or omit the trailing slash.
63
64         """
65
66         self._index: dict[str, dict[str, list[int]]] = {}
67         self._document_list: list[str] = []
68         self._document_directory: str = document_directory
69
70     @property
71     def index(self) -> dict[str, dict[str, list[int]]]:
72         return self._index
73
74     @property
75     def document_list(self) -> list[str]:
76         return self._document_list
77
78     @property
79     def document_directory(self) -> str:
80         return self._document_directory
81
82     def _add_occurrence(self, word: str, document_name: str, position_in_doc
83 : int):
```



```
72     """ Adds a word to the inverted index
73
74     Args:
75         word: The word to add to the index
76         document_name: The name of the document where the word occurs
77         position_in_doc: The position of the word in the document
78     """
79
80     # Create data structures if they do not exist
81     if word not in self._index:
82         self._index[word] = {}
83
84     if document_name not in self._index[word]:
85         self._index[word][document_name] = []
86
87     self._index[word][document_name].append(position_in_doc)
88
89     def create_index(self):
90         """Method to automatically construct the inverted index from the
91         documents
92             in the documentDirectory given during initialization.
93         """
94
95         # Go through all documents, add them to the document list and add
96         # all words to the index
97         # TODO: order is not at fault, sorted gives same results
98         file_names = sorted(os.listdir(self._document_directory))
99         for file_name in file_names:
100             doc_dir = self._document_directory
101             file = open(os.path.join(doc_dir, file_name))
102             self._document_list.append(file_name)
103
104             position_in_doc = 0
105             for word in file:
106                 # assume 1 word/line
107                 word_to_add = word.split()          # Split from newline char
108                 if not word_to_add:              # Skip if empty row
109                     position_in_doc += 1
110                     continue
111
112
113     class VectorSpaceModel:
114         """Class for creating and querying a vector space model
115
116             Allows creating a document term matrix in bulk via the
117             create_document_term_matrix()
118             method, which utilizes an inverted index object. The document term
```



```
matrix is
118    stored as a list of lists, where each list corresponds to a document and
119    each
120    element in the list corresponds to the weight of a word in the document.
121    The
122    weights are calculated using the tf-idf formula, with the tf and idf
123    factors
124    being determined by the parameters given during initialization.

125
126    Properties (read-only):
127        document_term_matrix: The document term matrix, as a list of lists.
128        inverted_index: The inverted index object used to create the
129        document term matrix
130        doc_tf: The tf factor used for the document term matrix
131        doc_idf: The idf factor used for the document term matrix
132        doc_normalization: The normalization factor used for the document
133        term matrix
134        query_tf: The tf factor used for queries
135        query_idf: The idf factor used for queries
136        query_normalization: The normalization factor used for queries
137
138    """
139
140    def __init__(
141        self,
142        inverted_index: InvertedIndex,
143        doc_tf: Literal['b', 't', 'n'],
144        doc_idf: Literal['x', 'f', 'p'],
145        doc_normalization: Literal['x', 'c'],
146        query_tf: Literal['b', 't', 'n'],
147        query_idf: Literal['x', 'f', 'p'],
148        query_normalization: Literal['x', 'c']
149    ):
150
151        self._document_term_matrix: list[list[float]] = []
152        self._inverted_index = inverted_index
153        self._doc_tf: Literal['b', 't', 'n'] = doc_tf
154        self._doc_idf: Literal['x', 'f', 'p'] = doc_idf
155        self._doc_normalization: Literal['x', 'c'] = doc_normalization
156        self._query_tf: Literal['b', 't', 'n'] = query_tf
157        self._query_idf: Literal['x', 'f', 'p'] = query_idf
158        self._query_normalization: Literal['x', 'c'] = query_normalization
159
160        # Check if combinations are valid
161        valid_factors = [['n', 'f', 'x'],
162                         ['n', 'f', 'c'],
163                         ['t', 'f', 'c'],
164                         ['t', 'x', 'c']]
```



```
162     doc_factors = [doc_tf, doc_idf, doc_normalization]
163     query_factors = [query_tf, query_idf, query_normalization]
164     if (doc_factors not in valid_factors or
165         query_factors not in valid_factors):
166         print("TF-IDF combinations are not valid, check the
167 documentation!")
168         return None
169
170     if doc_factors + query_factors not in suggested_combinations:
171         print("Warning: document + query factor combination is not
172 suggested!")
173
174     valid_doc_combinations = [[['n', 'f', 'c'],
175                                ['t', 'f', 'c'],
176                                ['t', 'x', 'c']]]
177
178     valid_query_combinations = [[['n', 'f', 'x'],
179                                ['t', 'f', 'c'],
180                                ['t', 'x', 'c']]]
181
182 @property
183 def document_term_matrix(self) -> list[list[float]]:
184     return self._document_term_matrix
185
186 @property
187 def inverted_index(self) -> InvertedIndex:
188     return self._inverted_index
189
190 @property
191 def doc_tf(self) -> Literal['b', 't', 'n']:
192     return self._doc_tf
193
194 @property
195 def doc_idf(self) -> Literal['x', 'f', 'p']:
196     return self._doc_idf
197
198 @property
199 def doc_normalization(self) -> Literal['x', 'c']:
200     return self._doc_normalization
201
202 @property
203 def query_tf(self) -> Literal['b', 't', 'n']:
204     return self._query_tf
205
206 @property
207 def query_idf(self) -> Literal['x', 'f', 'p']:
208     return self._query_idf
209
210 @property
211 def query_normalization(self) -> Literal['x', 'c']:
```



```
210     return self._query_normalization
211
212     def _t_x_c_calculation(self, word_counts: list[int]) -> list[float]:
213         """TF-IDF calculation method using the formula:
214         w = tf * idf / sqrt(tf^2 * idf^2), where:
215         tf = tf
216         idf = 1
217
218     Args:
219         word_counts: A list of the number of times
220             each word occurs in the document
221
222     Returns:
223         A list of the weights of each word in the document
224     """
225
226     index = self._inverted_index.index
227     words = index.keys()
228     weights_sum_square = 0      # for normalization
229
230     # For normalization (idf = 1)
231     for i in range(len(words)):
232         weights_sum_square += word_counts[i] * word_counts[i]
233
234     # Normalization calculation
235     # TF = tf
236     norm_factor = 1/sqrt(weights_sum_square)
237     word_counts = [word_count * norm_factor
238                   for word_count in word_counts]
239
240     return word_counts
241
242     def _t_f_c_calculation(self, word_counts: list[int]) -> list[float]:
243         """TF-IDF calculation method using the formula:
244         w = tf * idf / sqrt(tf^2 * idf^2), where:
245         tf = tf
246         idf = log10(total_documents/documents_containing_word)
247
248     Args:
249         word_counts: A list of the number of times each word occurs in
250             the document
251
252     Returns:
253         A list of the weights of each word in the document
254     """
255
256     index = self._inverted_index.index
257     words = index.keys()
258     total_documents = len(self._inverted_index.document_list)
259     weights_sum_square = 0      # for normalization
260
261     # IDF calculation (can be applied since tf=tf)
```



```
259     for i, word in enumerate(words):
260         documents_containing_word = len(index[word])
261         word_counts[i] *= log10(total_documents /
262             documents_containing_word)
263         weights_sum_square += word_counts[i] * word_counts[i]      # for
264         norm
265
266         # Normalization calculation
267         # TF = tf
268         norm_factor = 1/sqrt(weights_sum_square)
269         word_counts = [word_count * norm_factor
270                         for word_count in word_counts]
271
272     return word_counts
273
274 def _n_f_x_calculation(self, word_counts: list[int]) -> list[float]:
275     """TF-IDF calculation method using the formula:
276     w = tf * idf, where:
277     tf = 0.5 + 0.5 * (tf / max_tf) [max_tf = largest tf in document]
278     idf = log10(total_documents/documents_containing_word)
279
280     Args:
281         word_counts: A list of the number of times each word occurs in
282             the document
283
284     Returns:
285         A list of the weights of each word in the document
286     """
287
288     index = self._inverted_index.index
289     words = index.keys()
290     total_documents = len(self._inverted_index.document_list)
291     max_word_count = 0      # for tf calculation
292     idf_factors = []
293
294     # IDF calculation (saves to array for later use)
295     for i, word in enumerate(words):
296         documents_containing_word = len(index[word])
297
298         # for finding largest tf
299         if word_counts[i] > max_word_count:
300             max_word_count = word_counts[i]
301
302         idf_factors.append(log10(total_documents /
303             documents_containing_word))
304
305         # TF calculation
306         for i in range(len(word_counts)):
307             word_counts[i] = 0.5 + 0.5 * (word_counts[i] / max_word_count)
308
309             # tf
310             word_counts[i] = word_counts[i] * idf_factors[i]      # idf
```



```
304         return word_counts
305
306
307     def _n_f_c_calculation(self, word_counts: list[int]) -> list[float]:
308         """TF-IDF calculation method using the formula:
309         w = tf * idf / sqrt(tf^2 * idf^2), where:
310         tf = 0.5 + 0.5 * (tf / max_tf) [max_tf = largest tf in document]
311         idf = log10(total_documents/documents_containing_word)
312
313         Args:
314             word_counts: A list of the number of times each word occurs in
315             the document
316
317         Returns:
318             A list of the weights of each word in the document
319             """
320
321         index = self._inverted_index.index
322         words = index.keys()
323         total_documents = len(self._inverted_index.document_list)
324         max_word_count = 0          # for tf calculation
325         weights_sum_square = 0      # for norm
326         idf_factors = []
327
328         # IDF & norm calculation (saves to array for later use)
329         for i, word in enumerate(words):
330             documents_containing_word = len(index[word])
331
332             # for finding largest tf
333             if word_counts[i] > max_word_count:
334                 max_word_count = word_counts[i]
335
336             idf_factors.append(log10(total_documents/
337             documents_containing_word))
338
339             # TF calculation and sum of squares for norm
340             for i in range(len(word_counts)):
341                 word_counts[i] = 0.5 + 0.5 * (word_counts[i] / max_word_count)
342             # tf
343             word_counts[i] = word_counts[i] * idf_factors[i]           # idf
344             weights_sum_square += word_counts[i] * word_counts[i]       # norm
345
346             # normalization
347             norm_factor = 1/sqrt(weights_sum_square)
348             word_counts = [word_count * norm_factor
349                           for word_count in word_counts]
350
351         return word_counts
352
353
354     def _t_f_x_calculation(self, word_counts: list[int]) -> list[float]:
```



```
350
351     index = self._inverted_index.index
352     words = index.keys()
353     total_documents = len(self._inverted_index.document_list)
354     max_word_count = 0      # for tf calculation
355     idf_factors = []
356
357
358
359     def _calculate_tf_idf(
360         self,
361         word_counts: list[int],
362         tf_mode: Literal['b', 't', 'n'],
363         idf_mode: Literal['x', 'f', 'p'],
364         normalization_mode: Literal['x', 'c']) -> list[float]:
365         """Wrapper method for the different tf-idf calculation methods
366         Args:
367             word_counts: A list of the number of times each word occurs in
368             the document
369             tf_mode: The tf factor to use
370             idf_mode: The idf factor to use
371             normalization_mode: The normalization factor to use
372
373         Returns:
374             A list of the weights of each word in the document
375         """
376
377         match [tf_mode, idf_mode, normalization_mode]:
378             case ['t', 'x', 'c']:
379                 return self._t_x_c_calculation(word_counts)
380             case ['t', 'f', 'c']:
381                 return self._t_f_c_calculation(word_counts)
382             case ['n', 'f', 'x']:
383                 return self._n_f_x_calculation(word_counts)
384             case ['n', 'f', 'c']:
385                 return self._n_f_c_calculation(word_counts)
386             case ['t', 'f', 'x']:
387                 return self._t_f_x_calculation(word_counts)
388             case _:
389                 return -1
390
391     def create_document_term_matrix(self):
392         """Method for creating the document term matrix
393         Uses the inverted index object given during initialization to create
394         the
395         document term matrix.
396         """
397         tf_mode = self._doc_tf
398         idf_mode = self.doc_idf
```



```
398     normalization_mode = self.doc_normalization
399     index = self._inverted_index.index
400     words = index.keys()
401     documents = self._inverted_index.document_list
402
403     # For each document, create its term_array and calculate it's weight
404     for i, document in enumerate(documents):
405         if PRINT_OUTPUTS:
406             if i % 100 == 0:
407                 print("Processing document " + str(i + 1) + " of " + str
408 (len(documents)))
409
410         word_counts = [len(index[word][document])
411                         if document in index[word] else 0
412                         for word in words]
413         tf_idf = self._calculate_tf_idf(word_counts,
414                                         tf_mode,
415                                         idf_mode,
416                                         normalization_mode)
417         self._document_term_matrix.append(tf_idf)
418
419     def _inner_product(
420         self,
421         list_1: list[float],
422         list_2: list[float]) -> float:
423         """Method for calculating the inner product of two lists
424         Args:
425             list_1: The first list
426             list_2: The second list
427
428         Returns:
429             The inner product of the two lists
430         """
431
432         result = 0
433         for i in range(len(list_1)):
434             result += list_1[i] * list_2[i]
435
436         return result
437
438     def query(self, text: list[str]) -> list[tuple[str, float]]:
439         """Method for querying the document term matrix
440         Args:
441             text: The query text, as a list of words
442
443         Returns:
444             A list of tuples, where each tuple contains the name of a
445             document and
```



```
446     idf_mode = self._query_idf
447     normalization_mode = self._query_normalization
448
449     # create empty dict from corpus words and add occurrences from query
450     word_counts = {}
451
452     for key in self._inverted_index.index.keys():
453         word_counts[key] = 0
454
455     for word in text:
456         if word in word_counts:      # Skip if queried word is not in
457             word_counts[word] += 1
458
459     tf_idf = self._calculate_tf_idf(list(word_counts.values()),
460                                     tf_mode,
461                                     idf_mode,
462                                     normalization_mode)
463
464     similarity = [self._inner_product(tf_idf, document)
465                   for document in self._document_term_matrix]
466
467     pairs = zip(self.inverted_index.document_list, similarity)
468     sorted_pairs = sorted(pairs, key=lambda x: x[1], reverse=True)
469
470     return sorted_pairs[0:LIMIT_RESULTS-1]
471
472
473 def query_ranking(
474     vsm: VectorSpaceModel,
475     query: list[str],
476     relevant: list[str]):
477     similarity = vsm.query(query)
478
479     for i, doc in enumerate(similarity):
480         if doc[0] in relevant:
481             if PRINT_OUTPUTS:
482                 print("%s (%f) in position %d" % (doc[0], doc[1], i + 1))
483
484             if EXPORT_OUTPUTS:
485                 with open(os.path.join("output"), "a") as output_file:
486                     output_file.write("%s (%f) in position %d" % (doc[0],
487 doc[1], i + 1))
488
489
490 def query_metrics(
491     vsm: VectorSpaceModel,
492     query: list[str],
493     relevant: list[str]):
```



```
494     similarity = vsm.query(query)
495
496     # score variables
497     average_precision = 0
498     true_positive = 0
499     false_negative = 0
500     false_positive = 0
501
502     # setup variables
503     first_relevant_seen = False
504     relevant_seen = 0
505
506
507     for i, doc in enumerate(similarity):
508         if doc[0] in relevant:
509             # setup variables
510             if not first_relevant_seen:
511                 first_relevant_seen = True
512                 reciprocal_rank = 1 / (i + 1)
513
514             relevant_seen += 1
515
516             # score variables
517             average_precision += relevant_seen / (i+1)
518             true_positive += 1 # Document is relevant and retrieved
519         else:
520             false_positive += 1 # Document is retrieved but not relevant
521
522     # Calculate False Negative
523     false_negative = len(relevant) - true_positive
524
525     # score calculations
526     average_precision /= relevant_seen
527     precision = relevant_seen / LIMIT_RESULTS
528     recall = relevant_seen / len(relevant)
529     f1_score = 2 * (precision * recall) / (precision + recall)
530
531     if PRINT_OUTPUTS:
532         print("Precision=% .4f" % precision)
533         print("Recall=% .4f" % recall)
534         print("F1 Score=% .4f" % f1_score)
535         print("True Positive=%d" % true_positive)
536         print("False Positive=%d" % false_positive)
537         print("False Negative=%d" % false_negative)
538         print("Average Precision=% .4f" % average_precision)
539         print("Reciprocal Rank=% .4f" % reciprocal_rank)
540
541     if EXPORT_OUTPUTS:
542         with open(os.path.join("output"), "a") as output_file:
543             output_file.write(str(precision) +
```



```
544             ', ' +
545             str(recall) +
546             ', ' +
547             str(f1_score) +
548             ', ' +
549             str(true_positive) +
550             ', ' +
551             str(false_positive) +
552             ', ' +
553             str(false_negative) +
554             ', ' +
555             str(average_precision) +
556             ', ' +
557             str(reciprocal_rank) +
558             '\n')
559     output_file.close()
560
561
562 if __name__ == "__main__":
563     print("INCORRECT USAGE, RUN MAIN USING main.py FROM ROOT DIRECTORY")
```

Listing 2: fun.py

### 5.c Main VCM Reverse Indexing Code

```
1 import fun
2 from config import INPUT_SIZE
3 import os
4 import sys
5
6 if __name__ == "__main__":
7     inverted_index = fun.InvertedIndex("docs/")
8     inverted_index.create_index()
9
10    if len(sys.argv) == 7:
11        vsm = fun.VectorSpaceModel(inverted_index,
12                                    sys.argv[1],
13                                    sys.argv[2],
14                                    sys.argv[3],
15                                    sys.argv[4],
16                                    sys.argv[5],
17                                    sys.argv[6])
18    else:
19        print("No command line arguments given. Using default values.")
20        vsm = fun.VectorSpaceModel(inverted_index, 'n', 'f', 'c', 'n', 'f',
21                                   'x')
22
23        vsm.create_document_term_matrix()
24
25        queries_file = open(os.path.join("data", "Queries.txt"))
```



```
25     relevant_file = open(os.path.join("data", "Relevant.txt"))
26
27     # Write tf, idf and normalization factors to file
28     with open(os.path.join("output"), "a") as output_file:
29         output_file.write(vsm.doc_tf +
30                            vsm.doc_idf +
31                            vsm.doc_normalization +
32                            ' ' +
33                            vsm.query_tf +
34                            vsm.query_idf +
35                            vsm.query_normalization +
36                            '\n')
37
38     output_file.close()
39
40
41     for i in range(INPUT_SIZE):
42         query = queries_file.readline().replace("-", " ")
43         relevant = relevant_file.readline().split()
44         relevant = ["0" * (5 - len(doc_id)) + doc_id
45                     for doc_id in relevant]
46
47         print(str(i + 1) + "." + query)
48         fun.query_metrics(vsm, query, relevant)
49         print("=====")
```

Listing 3: main.py

## 5.d ColBERT Code

```
1 # Import necessary modules
2 import os
3 import sys
4 import csv
5
6 # Check if ColBERT repository exists; if not, clone it
7 if not os.path.exists('ColBERT/'):
8     os.system('git clone https://github.com/stanford-futuredata/ColBERT.git')
9 else:
10     # If ColBERT repository exists, pull the latest changes
11     os.system('git -C ColBERT/ pull')
12
13 # Add ColBERT directory to sys.path to enable imports
14 sys.path.insert(0, 'ColBERT/')
15
16 # Install required packages using pip
17 !pip install -e ColBERT/['faiss-gpu','torch']
18
19 # Import all necessary ColBERTv2 modules
20 import colbert
21 from colbert import Indexer, Searcher
```



```
22 from colbert.infra import Run, RunConfig, ColBERTConfig
23 from colbert.data import Queries, Collection
24
25 """
26     A class for handling files and giving training and testing data.
27
28     Attributes:
29         folder_path (str): The path to the folder containing the files.
30 """
31
32 class FileHandlerData:
33     def __init__(self, folder_path):
34         """Initializes the FileHandlerData with the specified folder path.
35
36         Args:
37             folder_path (str): Path to the folder containing files.
38         """
39         self.folder_path = folder_path
40
41     def read_dock_files(self):
42         """Reads dock files from the specified folder.
43
44         Returns:
45             list: A list containing texts of dock files indexed by file
46             number.
47         """
48         dock_file_texts = []
49
50         try:
51             # Get the list of all dock files in the folder.
52             dock_files = os.listdir(self.folder_path)
53             dock_files.sort(key=int) # Sort files numerically
54
55             # Create a list to store the dock file texts with None values
56             # initially.
57             dock_file_texts = ["Null"] * (int(max(dock_files)) + 1) # Add 1
58             to accommodate dock files starting from index 1.
59
60             # Iterate over the dock files and read their texts.
61             for dock_file in dock_files:
62
63                 dock_file_index = int(dock_file) # Assuming the dock file
64                 name can be converted to an integer.
65
66                 with open(os.path.join(self.folder_path, dock_file), "r",
67                           encoding="utf-8") as f:
68
69                     dock_file_text = f.read()
70                     dock_file_texts[dock_file_index] = dock_file_text
71
72         except FileNotFoundError as e:
```



```
67         print(f"File '{e.filename}' not found.")
68     except Exception as e:
69         print(f"Error: {e}")
70
71     return dock_file_texts
72
73 def read_queries_from_file(self, file_path):
74     """Reads queries from a text file.
75
76     Args:
77         file_path (str): Path to the input file containing queries.
78
79     Returns:
80         list: A list containing queries read from the file.
81
82     Raises:
83         FileNotFoundError: If the specified file is not found.
84         Exception: If an error occurs during file reading.
85     """
86
87     queries = [] # Initialize an empty list to store queries
88     try:
89         with open(file_path, "r", encoding="utf-8") as file:
90             for line in file:
91                 query = line.strip("\n") # Remove leading/trailing
92                 whitespaces and newline characters
93                 queries.append(query) # Add the cleaned query to the
94                 list of queries
95             except FileNotFoundError:
96                 raise FileNotFoundError(f"File '{file_path}' not found.")
97             except Exception as e:
98                 raise Exception(f"Error: {e}") # Raise an exception if any
99                 other error occurs during file reading
100
101         return queries # Return the list of queries read from the file
102
103
104     def read_relevance_scores_from_file(self, file_path):
105         """Reads the relevance scores for each query from a file.
106
107         Args:
108             file_path (str): The path to the file containing the
109             relevance scores.
110
111         Returns:
112             dict: A dictionary where the keys are the line numbers and
113             the values are the relevance scores for each line.
114         """
115
116         relevance_scores = {} # Initialize an empty dictionary to store
117         relevance scores
118         line_number = 0 # Initialize line number counter
```



```
111
112     try:
113         with open(file_path, 'r') as f:
114             for line in f:
115                 scores = line.strip().split() # Split the line into
116                 individual scores
117                 relevance_scores[line_number] = scores # Add scores
118                 to the dictionary with line number as key
119                 line_number += 1 # Increment line number
120
121     except FileNotFoundError:
122         raise FileNotFoundError(f"File '{file_path}' not found.")
123     except Exception as e:
124         raise Exception(f"Error: {e}") # Raise an exception if any
125         other error occurs during file reading
126
127         return relevance_scores # Return the dictionary containing
128         relevance scores for each line
129
130     def export_metrics(self, query_metrics, csv_file_path):
131         """Writes evaluation metrics to a CSV file.
132
133         Args:
134             query_metrics (dict): A dictionary containing evaluation metrics
135             for each query.
136             csv_file_path (str, optional): The path to the CSV file.
137             Defaults to "query_metrics_output.csv".
138
139             """
140
141             # Open the CSV file in write mode
142             with open(csv_file_path, mode="w", newline='') as csv_file:
143                 # Create a CSV writer object
144                 csv_writer = csv.writer(csv_file)
145
146
147                 # Write the header row
148                 header = ["Precision", "Recall", "F1-score", "True Positives", "False Positives", "False Negatives", "AP", "RR"]
149                 csv_writer.writerow(header)
150
151
152                 # Iterate over query metrics and write each row to the CSV file
153                 for query_index, metrics in query_metrics.items():
154                     row = [
155                         f"{metrics['Precision']:.2f}",
156                         f"{metrics['Recall']:.2f}",
157                         f"{metrics['F1-score']:.2f}",
158                         metrics['True Positives'],
159                         metrics['False Positives'],
160                         metrics['False Negatives'],
161                         f"{metrics['AP']:.2f}",
162                         f"{metrics['RR']:.2f}"
163                     ]
164
165                     csv_writer.writerow(row)
```



```
154
155 class ColBERT_Search_Engine:
156     def __init__(self, index_name, dock_file_texts):
157         """Initializes the ColBERTSearchEngine with the specified index name
158         and dock file texts.
159
160         Args:
161             index_name (str): Name of the index for the search engine.
162             dock_file_texts (list): List containing texts of dock files for
163             indexing.
164
165             """
166
167             self.index_name = index_name
168             self.dock_file_texts = dock_file_texts
169
170
171             def create_index(self, doc maxlen, nbits , kmeans_niters):
172                 """Creates an index for the search engine using the provided dock
173                 file texts.
174
175                 Args:
176                     doc maxlen (int): Maximum length of a document for indexing.
177                     nbits (int): Number of bits for hash functions in the index.
178
179                 Returns:
180                     Indexer: An instance of the Indexer class representing the
181                     created index.
182
183                     """
184
185                     checkpoint = 'colbert-ir/colbertv2.0'
186
187
188                     with Run().context(RunConfig(nranks=1, experiment='notebook')):
189                         config = COLBERTConfig(doc maxlen=doc maxlen, nbits=nbits,
190                         kmeans_niters = kmeans_niters)
191
192                         indexer = Indexer(checkpoint=checkpoint, config=config)
193                         indexer.index(self.index_name, collection=self.dock_file_texts,
194                         overwrite=True)
195
196
197                     return indexer
198
199
200                     def search_queries(self, queries, _number_):
201                         """Searches for queries in the created index and returns the search
202                         results.
203
204                         Args:
205                             queries (list): List of queries to search for.
206                             _number_ (int): Number of top passages to retrieve for each
207                             query.
208
209                         Returns:
210                             dict: A dictionary where keys are query indices and values are
211                             lists of tuples
212                                 containing (query, passage_id, passage_score) for top
```



```
passages.

195      """
196
197     with Run().context(RunConfig(experiment='notebook')):
198         searcher = Searcher(self.index_name, collection=self.
199         dock_file_texts)
200
201
202     results_dict = {} # Initialize an empty dictionary to store the
203     results
204
205     i = 0 # Initialize an incrementing number
206
207
208     for query in queries:
209         print(f"Query: [{query}]")
210         results = searcher.search(query, k=_number_)
211         query_results = []
212         for passage_id, passage_rank, passage_score in zip(*results):
213             query_results.append((query, passage_id, passage_score,
214             passage_rank))
215         results_dict[i] = query_results
216         i += 1 # Increment the number for the next iteration
217
218
219     return results_dict
220
221 class EvaluationMetrics:
222
223     @staticmethod
224     def calculate_confusion_matrix(retrieved_docs, relevant_docs):
225         """Calculate confusion matrix for binary classification.
226
227         Args:
228             retrieved_docs (list): List of retrieved document IDs.
229             relevant_docs (list): List of relevant document IDs.
230
231         Returns:
232             tuple: True positives, false positives, false negatives, true
233             negatives.
234
235         """
236
237         true_positives = len(set(retrieved_docs) & set(relevant_docs))
238         false_positives = len(set(retrieved_docs) - set(relevant_docs))
239         false_negatives = len(set(relevant_docs) - set(retrieved_docs))
240         true_negatives = 0 # In information retrieval, TN is usually not
241         applicable
242
243
244         return true_positives, false_positives, false_negatives,
245         true_negatives
246
247
248     @staticmethod
249     def calculate_precision_recall(retrieved_docs, relevant_docs):
250         """Calculate precision and recall.
251
252         Args:
253             retrieved_docs (list): List of retrieved document IDs.
254             relevant_docs (list): List of relevant document IDs.
```



```
238
239     Returns:
240         tuple: Precision and recall.
241         """
242
243     # Calculate precision and recall
244     true_positives, false_positives, false_negatives, _ =
245     EvaluationMetrics.calculate_confusion_matrix(
246         retrieved_docs, relevant_docs)
247
248     precision = 0 if (true_positives + false_positives) == 0 else
249     true_positives / (true_positives + false_positives)
250     recall = 0 if (true_positives + false_negatives) == 0 else
251     true_positives / (true_positives + false_negatives)
252     return precision, recall
253
254     @staticmethod
255     def calculate_f1_score(precision, recall):
256         """Calculate F1-score.
257
258         Args:
259             precision (float): Precision value.
260             recall (float): Recall value.
261
262         Returns:
263             float: F1-score.
264         """
265
266     # Calculate F1-score:  $2 * (precision * recall) / (precision + recall)$ 
267
268     if precision + recall == 0:
269         return 0
270     return 2 * (precision * recall) / (precision + recall)
271
272     @staticmethod
273     def calculate_ap(results_dict, relevance_scores):
274         """Calculate Average Precision (AP) for the given results and
275         relevance scores.
276
277         Args:
278             results_dict (dict): A dictionary containing query indices and
279             corresponding retrieved documents.
280             relevance_scores (dict): A dictionary containing query indices
281             and lists of relevant document IDs.
282
283         Returns:
284             dict: A dictionary with query indices as keys and their
285             corresponding AP scores as values.
286         """
287
288     # Dictionary to store AP scores for each query
289     query_metrics = {}
```



```
280     for query_index, retrieved_docs in results_dict.items():
281         relevant_docs = set(int(doc_id) for doc_id in relevance_scores[
282             query_index])
283
283         # Calculate precision at each position and average precision
284         precision_at_k = [1 if doc_id in relevant_docs else 0 for _,,
285             doc_id, _, _ in retrieved_docs]
285         average_precision = sum(precision_at_k[:i + 1].count(1) / (i +
286             1) for i in range(len(precision_at_k)) if precision_at_k[i] == 1)
286         average_precision /= len(relevant_docs) if len(relevant_docs) >
287             0 else 1
288
288         query_metrics[query_index] = average_precision
289
289     return query_metrics
290
291
292     @staticmethod
293     def calculate_rr(query_results, relevance_scores):
294         """Calculate Reciprocal Rank (RR) for the given results and
295         relevance scores.
296
296         Args:
297             query_results (dict): A dictionary where each key is a query
298             index, and the corresponding value is a list of tuples
299             containing (passage_id, passage_score,
300             passage_rank).
301             relevance_scores (dict): A dictionary where each key is a query
302             index, and the corresponding value is a list
303             containing relevant passage IDs.
304
304         Returns:
305             dict: A dictionary where each key is a query index, and the
306             corresponding value is the Reciprocal Rank (RR) for that query.
307             """
307             rr_values = {}
308
309             for query_index, results_list in query_results.items():
310                 relevant_docs = set(int(doc_id) for doc_id in relevance_scores[
311                     query_index])
312
312                 # Find the rank of the first relevant document (Reciprocal Rank)
313                 reciprocal_rank = 0
314                 for i, (_, doc_id, _, _) in enumerate(results_list, start=1):
315                     if int(doc_id) in relevant_docs:
316                         reciprocal_rank = 1 / i
317                         break
318
318                 rr_values[query_index] = reciprocal_rank
319
319             return rr_values
```



```
320
321 def main():
322
323     # Specify the folder path containing dock files
324     folder_path_docs = "/content/drive/MyDrive/docs"
325
326     # Specify the file path containing the Queries
327     # file_path_queries = "/content/drive/MyDrive/data/Queries.txt"
328     file_path_queries = "/content/drive/MyDrive/data/Queries_20"
329
330     # Specify the file path containing the evaluation data relevant texts
331     # file_path_evaluation = "/content/drive/MyDrive/data/Relevant.txt"
332     file_path_evaluation = "/content/drive/MyDrive/data/Relevant_20"
333
334     # ColBERT Parameters
335     # Number of ranked texts results the search engine returns
336     k_number = 400
337     doc_maxlen=500 # truncate passages at 500 tokens
338     kmeans_niters=14 # kmeans_niters specifies the number of iterations of k
339     -means clustering.
340     nbits=4 # encode each dimension with 4 bits
341
342     # Initialize and use DockFileHandler to read dock files from the
343     # specified folder
344     dock_file_handler = FileHandlerData(folder_path_docs)
345     dock_file_texts = dock_file_handler.read_dock_files()
346     # print(f"Text of dock file with index 200:\n{dock_file_texts[200]}")
347
348     queries = dock_file_handler.read_queries_from_file(file_path_queries)
349     # print(queries[88])
350
351
352     relevance_scores = dock_file_handler.read_relevance_scores_from_file(
353         file_path_evaluation)
354     # print(f"re {relevance_scores[99]}")
355
356
357     indexer_name = 'Le_Indexer'
358     search_engine = ColBERT_Search_Engine(indexer_name, dock_file_texts)
359     indexer = search_engine.create_index(doc_maxlen, nbts, kmeans_niters)
360
361     results_dict = search_engine.search_queries(queries, k_number)
362
363     # comment out the following section if you want to print the search
364     # engine results each query
365     # Print the results
366     for query, results in results_dict.items():
367         print(f"Results for query: [{query+1}]")
368         for query,passage_id,passage_score,passage_rank in results:
369             print(f"Query: {query} Passage ID: {passage_id}, Score: {passage_score:.1f}, Rank [{passage_rank}]")
```



```
365     print("\n")
366
367
368     # Initialize an empty dictionary to store evaluation metrics for each
369     # query
370     query_metrics = {}
371
372     # Iterate through all queries
373     for query_index in range(0, len(queries)):
374         query_text = queries[query_index] # Get the query text for the
375         current query
376         retrieved_docs = results_dict[query_index]
377         relevant_docs = set(int(doc_id) for doc_id in relevance_scores[
378             query_index])
379
380         # Calculate precision, recall, F1-score, and confusion matrix for
381         # the current query
382         retrieved_doc_ids = set(doc_id for _, doc_id, _, _ in retrieved_docs
383     )
384         precision, recall = EvaluationMetrics.calculate_precision_recall(
385             retrieved_doc_ids, relevant_docs)
386         f1_score = EvaluationMetrics.calculate_f1_score(precision, recall)
387         true_positives, false_positives, false_negatives, _ =
388             EvaluationMetrics.calculate_confusion_matrix(
389                 retrieved_doc_ids, relevant_docs)
390
391         # Calculate MAP and MRR for the current query
392         ap_values = EvaluationMetrics.calculate_ap({query_index:
393             retrieved_docs},
394                                         {query_index:
395                 relevance_scores[query_index]})
396         rr_values = EvaluationMetrics.calculate_rr({query_index:
397             retrieved_docs},
398                                         {query_index:
399                 relevance_scores[query_index]})
400
401         # Store metrics in the dictionary including query text
402         query_metrics[query_index] = {
403             "Query Text": query_text, # Include query text in the
404             "dictionary"
405             "Precision": precision,
406             "Recall": recall,
407             "F1-score": f1_score,
408             "True Positives": true_positives,
409             "False Positives": false_positives,
410             "False Negatives": false_negatives,
411             "AP": ap_values[query_index],
412             "RR": rr_values[query_index]
413         }
414
415 }
```



```
403     dock_file_handler.export_metrics(query_metrics , "  
404         query_metrics_output.csv")  
405  
406 if __name__ == "__main__":  
    main()
```

Listing 4: ColBERT ceid ir Implementation.ipynb

## 5.e Metrics Visualization Code

```
1 import csv  
2 import os  
3 import matplotlib.pyplot as plt  
4 from config import PRINT_OUTPUTS , EXPORT_OUTPUTS  
5  
6 def read_csv(file_path):  
7     """  
8         Reads data from a CSV file.  
9  
10    Args:  
11        file_path (str): Path to the CSV file.  
12  
13    Returns:  
14        dict: A dictionary containing metric names as keys and lists of  
15        metric values as values.  
16        """  
17    data = {  
18        'Precision': [] ,  
19        'Recall': [] ,  
20        'F1-score': [] ,  
21        'True Positives': [] ,  
22        'False Positives': [] ,  
23        'False Negatives': [] ,  
24        'AP': [] ,  
25        'RR': []  
26    }  
27  
28    try:  
29        with open(file_path , newline='') as csvfile:  
30            reader = csv.DictReader(csvfile)  
31            for row in reader:  
32                for key , value in row.items():  
33                    data[key].append(float(value))  
34    except FileNotFoundError:  
35        print(f"Error: File not found at {file_path}")  
36    except Exception as e:  
37        print(f"Error reading file {file_path}: {e}")  
38  
39    return data
```



```
40 def plot_metric(metric, search_engine1, search_engine2):
41     """
42         Plots a specific metric for two search engines.
43
44     Args:
45         metric (str): The metric to plot.
46         search_engine1 (dict): Data for search engine 1.
47         search_engine2 (dict): Data for search engine 2.
48     """
49     # Create the metrics_visualization folder if it doesn't exist
50     folder_path = 'metrics_visualization_100'
51     os.makedirs(folder_path, exist_ok=True)
52
53     plt.figure(figsize=(18, 12)) # Set the size of the figure
54     x_ticks = list(range(0, len(search_engine1[metric]), 2))
55     plt.plot(search_engine1[metric], label='Search Engine VCM (nfc nfx)', color='darkcyan')
56     plt.plot(search_engine2[metric], label='Search Engine ColBERT', color='darkorange')
57     plt.xlabel('Query Number')
58     plt.ylabel(metric)
59     plt.title(f'{metric} - VCM (nfc nfx) vs ColBERT')
60     plt.grid(alpha=0.3)
61     plt.legend()
62     plt.xticks(x_ticks)
63     plt.ylim(0) # y-axis to start from zero
64     plt.xlim(0) # x-axis to start from zero
65     if EXPORT_OUTPUTS:
66         plt.savefig(os.path.join(folder_path, f'{metric}_comparison_100_nfc_nfx.png'), dpi=300)
67     if PRINT_OUTPUTS:
68         plt.show()
69
70 def plot_precision_recall_curve(precision, recall, engine_name):
71     """
72         Plots the precision-recall curve.
73
74     Args:
75         precision (list): Precision values.
76         recall (list): Recall values.
77         engine_name (str): Name of the search engine.
78     """
79     plt.figure(figsize=(8, 8))
80     plt.plot(recall, precision, color='darkblue', lw=2, label='Precision-Recall curve')
81     plt.xlabel('Recall')
82     plt.ylabel('Precision')
83     plt.title(f'Precision-Recall Curve - {engine_name}')
84     plt.legend()
85     plt.grid(alpha=0.3)
```



```
86     if EXPORT_OUTPUTS:
87         plt.savefig(os.path.join('metrics_visualization', f'precision_recall_curve_{engine_name.lower()}.png'), dpi=300)
88     if PRINT_OUTPUTS:
89         plt.show()
90
91 if __name__ == "__main__":
92
93     file_path1 = './output_nfc_nfx.csv'
94     file_path2 = './query_metrics_output_100.csv'
95
96     search_engine1_data = read_csv(file_path1)
97     search_engine2_data = read_csv(file_path2)
98
99     metrics = [
100         'Precision',
101         'Recall',
102         'F1-score',
103         'True Positives',
104         'False Positives',
105         'False Negatives',
106         'AP',
107         'RR'
108     ]
109
110     for metric in metrics:
111         plot_metric(metric, search_engine1_data, search_engine2_data)
112
113
114     # Calculate and print MAP and MRR
115     for i, (search_data, engine_name) in enumerate(zip([search_engine1_data,
116             search_engine2_data], ['VCM (nfc nfx)', 'ColBERT'])):
117         precision = sum(search_data['Precision']) / len(search_data['Precision'])
118         recall = sum(search_data['Recall']) / len(search_data['Recall'])
119         f1_score = sum(search_data['F1-score']) / len(search_data['F1-score'])
120         true_positives = sum(search_data['True Positives']) / len(search_data['True Positives'])
121         false_positives = sum(search_data['False Positives']) / len(search_data['False Positives'])
122         average_precision = sum(search_data['AP']) / len(search_data['AP'])
123         reciprocal_rank = sum(search_data['RR']) / len(search_data['RR'])
124
125         print(f"\nMetrics for Search Engine 100 Queries {engine_name}:")
126         print(f"Precision: {precision:.4f}")
127         print(f"Recall: {recall:.4f}")
128         print(f"F1-score: {f1_score:.4f}")
129         print(f"True Positives: {true_positives}")
130         print(f"False Positives: {false_positives}")
```



```
130     print(f"Mean Average Precision (MAP): {average_precision:.4f}")
131     print(f"Mean Reciprocal Rank (MRR): {reciprocal_rank:.4f}")
```

Listing 5: metrics visualization



## Κεφάλαιο 6: Βιβλιογραφία

- [1] google. colab.google. <https://colab.google/notebooks/>.
- [2] Google. Google python style guide. <https://google.github.io/styleguide/pyguide.html>.
- [3] Jon Saad-Falcon Christopher Potts Keshav Santhanam, Omar Khattab and Matei Zaharia. Colbertv2: Effective and efficient retrieval via lightweight late interaction. (*SIGIR'20, TACL'21, NeurIPS'21, NAACL'22, CIKM'22*, page 5, 2021).
- [4] Python. 5.data structures - list comprehensions. <https://docs.python.org/3.9/tutorial/datastructures.html#list-comprehensions>.
- [5] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing Management*, 24(5):513–523, 1988.
- [6] Gerard Salton and Christopher Buckley. Colbert: Efficient and effective passage search via contextualized late interaction over bert. *SIGIR*, 0(0):0, 2020.
- [7] stanford. Colbert (v2). <https://github.com/stanford-futuredata/ColBERT?tab=readme-ov-file>.