Τμήμα Μηχανικών Η/Υ και Πληροφορικής
Πανεπιστήμιο Πατρών
Πολυτεχνική Σχολή

Τομέας Λογικού των Υπολογιστών
Διδάσκων: Χρήστος Μακρής
Ακαδημαϊκό Έτος: 2023 – 2024
Ημ / νία Παράδοσης: 28 / 01 / 2024

Ανάκτηση Πληροφορίας

Επιλεγόμενο Μάθημα – CEID_NE5597

Εργαστηριακή Άσκηση
Χειμερινό Εξάμηνο 2023

Χρυσαυγή Πατέλη | 1084513 | up1084513@ac.upatras.gr
Μηλτιάδης Μαντές | 1084661 | up1084661@ac.upatras.gr

ΠΕΡΙΕΧΟΜΕΝΑ

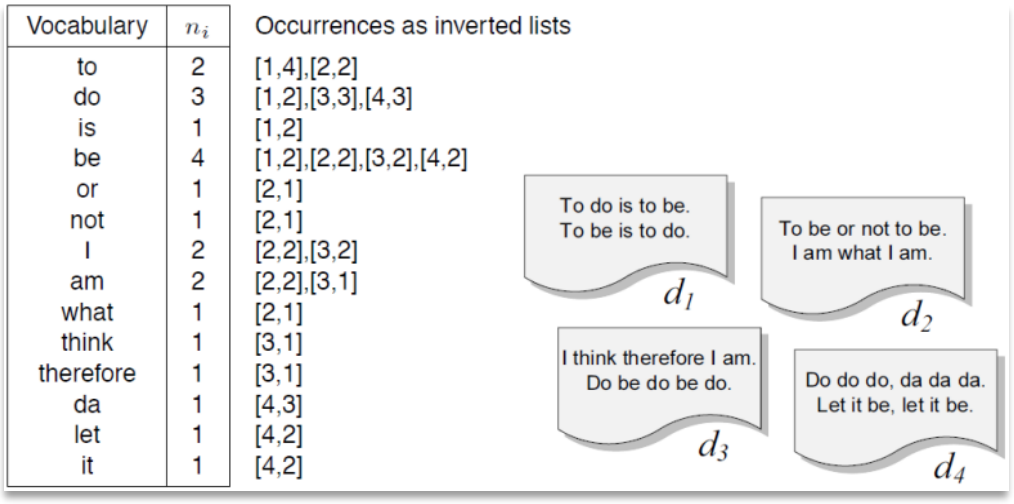
ΕΙΣΑΓΩΓΗ

1.1 Ερώτημα 1

Στο συγκεκριμένο ερώτημα μας ζητείται να υλοποιήσουμε την δομή του **Ανεστραμμένου Ευρετηρίου** (Inverted Index) πάνω στη συλλογή *Cystic Fibrosis*.

Το Ανεστραμμένο Ευρετήριο είναι μια δομή δεδομένων, η οποία χρησιμοποιείται σε συστήματα ανάκτησης πληροφοριών, μηχανές αναζήτησης και γενικά σε συστήματα που είναι απαραίτητη η αναζήτηση κειμένων. Στόχος του είναι η γρήγορη αναζήτηση και εύρεση όλων των εγγράφων που περιέχουν έναν συγκεκριμένο όρο ή φράση. Οι όροι αυτοί καλούνται και όροι δεικτοδότησης.

Το Αναστραμμένο Ευρετήριο αποτελείται από δυο μέρη, το λεξικό που περιέχει τους όρους δεικτοδότησης από όλη την συλλογή και για κάθε όρο του λεξικού τις λίστες εμφανίσεων, οι οποίες περιέχουν πληροφορίες για την εμφάνιση των λέξεων στα κείμενα. Οι λίστες αυτές ονομάζονται ανεστραμμένες λίστες. Η πιο απλή μορφή τέτοιας λίστας είναι να περιέχει το πλήθος των εγγράφων που περιέχεται κάθε όρος καθώς και τα IDs των εγγράφων. Υπάρχουν ωστόσο και άλλες εκδοχές, όπου στην ανεστραμμένη λίστα μαζί με το ID των κειμένων που περιέχουν την λέξη υπάρχουν και άλλες πληροφορίες σχετικά με την λέξη. Για παράδειγμα, στην αναζήτηση πληροφορίας στο διαδίκτυο μαζί με τα URLs (ή website IDs) αποθηκεύονται και τα μεταδεδομένα της ιστοσελίδας για καλύτερη αναπαράσταση του τελικού αποτελέσματος ανάκτησης στον χρήστη. Στην εκδοχή που χρησιμοποιήσαμε εμείς περιέχεται ως επιπλέον πληροφορία η συχνότητα της κάθε λέξης μέσα στο συγκεκριμένο κείμενο εκτός από το ID του κειμένου, όπως φαίνεται και στην παρακάτω εικόνα.



Εικόνα 1: Προσομοίωση Ανεστραμμένου Ευρετηρίου

Το πλεονέκτημα του **Ανεστραμμένου Ευρετηρίου** είναι ότι επιτρέπει γρήγορες, αποδοτικές αναζητήσεις ερωτημάτων και ανάλογα την μορφή μπορεί να απαντήσει σε διάφορα είδη ερωτημάτων (ερωτήματα πολλαπλών λέξεων, μονή λέξης, τομή λίστας, σύνθετα ερωτήματα, ερωτήματα φράσης και εγγύτητας, boolean). Ωστόσο, διαθέτουν και μειονεκτήματα, καθώς έχουν υψηλό κόστος συντήρησης, αποθήκευσης

και απαιτούν πολύ χώρο στην μνήμη για την αποθήκευση και την ενημέρωσή τους. Γι' αυτό και συχνά εφαρμόζονται τεχνικές συμπίεσης (κωδικοποίηση δέλτα, κωδικοποίηση γάμμα, κωδικοποίηση μεταβλητών byte) πάνω στο **Ανεστραμμένο Ευρετήριο**, για περαιτέρω μείωση του απαιτούμενου χώρου. Επιπλέον, είναι πιο αποδοτικό για τους ίδιους λόγους το κάθε κείμενο (ή σελίδα) να αναπαρίσταται με ένα μοναδικό αναγνωριστικό τύπου ακεραίου και όχι με βάση τον τίτλο, το URL ή γενικά αλφαριθμητικά, καθώς καταλαμβάνουν μεγαλύτερο χώρο στη κύρια μνήμη και είναι πιο δύσκολο να εφαρμοστούν σε αυτά bitwise λογικοί τελεστές, όπως AND, OR και NOT.

1.2 Ερώτημα 2

Στο συγκεκριμένο ερώτημα μας ζητείται να υλοποιήσουμε το **Μοντέλο Διανυσματικού Χώρου** (Vector Space Model) πάνω στο Ανεστραμμένο Ευρετήριο που εξάγαμε στο προηγούμενο ερώτημα.

Το **Μοντέλο Διανυσματικού Χώρου** χρησιμοποιεί την έννοια των διανυσμάτων στον πολυδιάστατο χώρο προκειμένου να αναπαραστήσει αλγεβρικά την φυσική γλώσσα των κειμένων της συλλογής εγγράφων. Πιο συγκεκριμένα, το σκεπτικό πίσω από αυτό το σύστημα ανάκτησης πληροφορίας έγκειται στην ευκολία εύρεσης κοινών όρων (terms) ανάμεσα σε κείμενα και ερωτήματα χρήστη (queries) αν τα ζυγισμένα διανύσματα που τα αναπαριστούν τοποθετηθούν σε διανυσματικό χώρο *V*.

Σε μια πιο γενική περιγραφή του μοντέλου ισχύουν τα εξής:

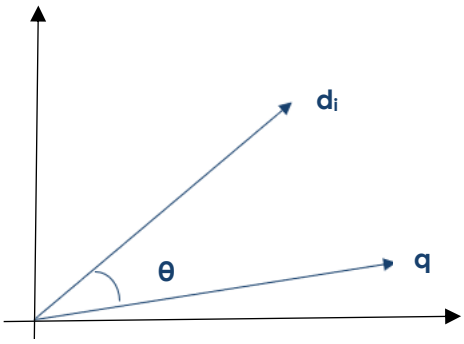
1. Έστω $\{t_1, t_2, \ldots, t_n\}$ οι όροι ενός κειμένου d_i, $i = 1, 2, \ldots, m$ από την συλλογή μας. Υποθέτοντας χωρίς βλάβη γενικότητας ότι για κάθε διακριτό όρο t_i υπάρχει στο χώρο ένα μοναδιαίο ορθοκανονικό διάνυσμα $\mathbf{t_i}$ στο οποίο αντιστοιχίζεται, τότε ο διανυσματικός χώρος *V* θα είναι το span$\{\mathbf{t_i}\}$ και έτσι το διάνυσμα $\mathbf{d_i}$ του κειμένου μπορεί να δοθεί ως γραμμικός συνδυασμός των $\mathbf{t_i}$.

$$\mathbf{d_i} = \sum_{k=1}^{n} a_{k,i}\mathbf{t_k}$$

2. Έστω $\{T_1, T_2, \ldots, T_n\}$ οι όροι ενός ερωτήματος, οι οποίοι σχετίζονται με το σύνολο $\{t_1, t_2, \ldots, t_n\}$. Για κάθε ερώτημα q όμοια προκύπτει ότι το αντίστοιχο του διάνυσμα **q** του διανυσματικού χώρου *V* θα είναι πάλι γραμμικός συνδυασμός των διανυσμάτων $\mathbf{t_i}$.

$$\mathbf{q} = \sum_{k=1}^{n} q_{k,i}\mathbf{t_k}$$

Κάθε διακριτός όρος σε κάθε περίπτωση αντιπροσωπεύει μια διάσταση του χώρου μας. Συνεπώς, στόχος μας είναι στον n – διάστατο χώρο *V* για κάθε διάνυσμα **q** να υπολογίσουμε την απόστασή του από κάθε διάνυσμα $\mathbf{d_i}$. Για να γίνει αυτό επιδιώκουμε να προσεγγίσουμε τη γωνία που σχηματίζεται ανάμεσά τους και πιο συγκεκριμένα το συνημίτονό της, cos(θ), το οποίο δίνεται από το εσωτερικό τους γινόμενο.



Εικόνα 2: Γραφική Αναπαράσταση **VSM** σε 2 – d χώρο

$$\cos(\theta) = \frac{\|\mathbf{d_i}\|\|\mathbf{q}\|}{\mathbf{d_i}\cdot\mathbf{q}}$$

Στο τέλος, χρησιμοποιούμε ως κριτήριο ταξινόμησης τη τιμή του συνημίτονου που προκύπτει προκειμένου να διατάξουμε τα κείμενα μας με βάση τη σχετικότητα τους ως προς κάθε ερώτημα χρήστη.

Ειδικότερα τώρα, στη δικιά μας υλοποίηση επιλέγουμε για απλοποίηση τα διανύσματα $\mathbf{d_i}$, $\mathbf{q_j}$ να έχουν ως διάσταση το πλήθος των διακριτών όρων του Ανεστραμμένου

Ευρετηρίου και κάθε γραμμή του διανύσματος $\mathbf{d_i}$ και $\mathbf{q_j}$ να περιέχει ένα βάρος για τον εκάστοτε όρο ($d_{i,k}$ και $q_{j,k}$ αντίστοιχα).

Τώρα, για την καλύτερη εφαρμογή του **Vector Space Model** είναι απαραίτητη η αξιοποίηση της τεχνικής του *term – weighting*, καθώς οι διαφορετικοί όροι μπορεί να έχουν διαφορετική σημασία και άρα και επίδραση στα συμφραζόμενα του κειμένου. Αρχικά, όσον αφορά το local term – weighting, επιλέγουμε μια από τις ακόλουθες τεχνικές υπολογισμού του term frequency (TF), καθώς μας αφορά μόνο η συχνότητα με την οποία εμφανίζονται οι όροι σε κάθε κείμενο ανεξάρτητα, την οποία και έχουμε αποθηκευμένη στο ανεστραμμένο αρχείο.

Τρόπος Υπολογισμου	TF_{ij}
Δυαδικό	$\{0, 1\}$
Απλή συχνότητα εμφάνισης	F_{ij}
Απλή λογαριθμικη κανονικοποίηση	$1 + log\ F_{ij}$
Διπλή 0,5 κανονικοποίηση	$0.5 + 0.5 \cdot \frac{F_{ij}}{MAX(F_{ij})}$
Διπλή K κανονικοποίηση	$K + (1 - K) \cdot \frac{F_{ij}}{MAX(F_{ij})}$

Πίνακας 1: Τρόποι Υπολογισμού Term Frequency

Η πρώτη μας σκέψη είναι να επιλέξουμε την *Απλή Συχνότητα Εμφάνισης*, η οποία υπολογίζει πόσες φορές εμφανίζεται ένας όρος στο κείμενο. Άλλωστε, όσο περισσότερες φορές εμφανίζεται ένας όρος στο κείμενο, τόσο πιο πιθανό είναι το κείμενο αυτό να είναι σχετικό με το ερώτημά μας. Ωστόσο, το μειονέκτημα αυτού του σκεπτικού είναι κυρίως ότι ευνοεί υπερβολικά τους όρους που εμφανίζονται με αυξημένη συχνότητα, καθώς η μεγάλη συχνότητα εμφάνισης δεν μας εξασφαλίζει υποχρεωτικά ότι οι όροι αυτοί θα είναι και πιο σημαντικοί από κάποιους όρους που εμφανίζονται πιο σπάνια. Επίσης, η χρήση δυαδικών βαρών είναι και αυτή αρκετά περιοριστική. Γι' αυτό καταλήγουμε στην *Απλή Λογαριθμική Κανονικοποίηση*, η οποία προσπαθεί να αντισταθμίσει την επίδραση της *Απλής Συχνότητας Εμφάνισης* μειώνοντας μέσω του λογαρίθμου την επίδραση πολύ μεγάλων διαφορών στις συχνότητες εμφάνισης, έτσι ώστε να έχουμε δικαιότερη απόδοση βαρών στο τελικό διάνυσμα $\mathbf{d_i}$.

Έπειτα, όσον αφορά το global term – weighting, θέλουμε να εξετάσουμε πώς η κατανομή και η συχνότητα εμφάνισης ενός όρου σε όλα τα κείμενα της συλλογής επηρεάζει το πόσο σημαντικός είναι. Για το λόγο αυτό επιλέγουμε την αποδοτικότερη από τις πιο κάτω τεχνικές για τον υπολογισμό του Inverted Document Frequency (IDF):

Πίνακας 2: Τρόποι Υπολογισμού Inverted Document Frequency

Τρόπος Υπολογισμου	IDF_i
Μοναδιαίο	1
Απλη ανάστροφη συχνότητα εμφάνισης	$log(\frac{N}{n_i})$
Απλή λογαριθμική κανονικοποίηση	$log(1 + \frac{N}{n_i})$
Ανάστροφη κανονικοποίηση περισσότερων εμφανίσεων	$log(1 + \frac{MAX(n_i)}{n_i})$

Εδώ επιλέγουμε την τεχνική της Απλής Ανάστροφης Συχνότητας Εμφάνισης, η οποία «λογαριθμίζει» την αναλογία των κειμένων μέσα στη συλλογή μας που περιέχουν ένα συγκεκριμένο όρο. Η τιμή της αυξάνεται όσο το πλήθος των κειμένων στο οποίο εντοπίζεται αυτός ο όρος μειώνεται και έτσι ψηλή τιμή σημαίνει συνήθως ότι ένας όρος εμφανίζεται στο συγκεκριμένο έγγραφο με μεγαλύτερη συχνότητα από το συνηθισμένο.

Έχοντας ολοκληρώσει αυτή τη διαδικασία, πολλαπλασιάζουμε κάθε συντεταγμένη των διανυσμάτων μας με τα τελικά βάρη και τα διανύσματα που προκύπτουν θα είναι:

d_i = [d_{i1}, d_{i2}, d_{i3}, …, d_{ik}] με d_{ik} να συμβολίζει το βάρος του k – όρου στο έγγραφο i
q_j = [q_{j1}, q_{j2}, q_{j3}, …, q_{jk}] με q_{jk} να συμβολίζει το βάρος του k – όρου στο ερώτημα j

και συνεπώς το συνημίτονο της γωνίας που θα σχηματίζουν θα δίνεται από τη σχέση:

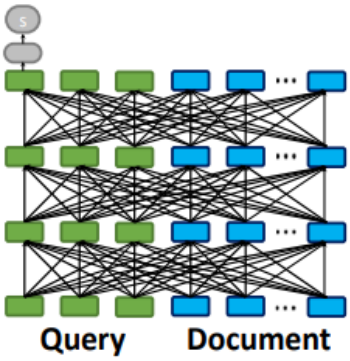
$$\cos(\mathbf{d_i}, \mathbf{q_j}) = \frac{\sum_{k=1}^{n} d_{ik} q_{jk}}{\sqrt{\sum_{l=1}^{n} d_{il}^2 \sum_{k=1}^{n} q_{jk}^2}}$$

Σημειώνουμε εδώ ότι τα βάρη των όρων που δεν εντοπίζονται μέσα σε κάποιο κείμενο θα παραμείνουν 0 λόγω του πολλαπλασιασμού με τα βάρη tf-idf όπως προείπαμε.

Μια συνηθισμένη τεχνική αποτελεί επίσης η κανονικοποίηση του μήκους των διανυσμάτων **d_i**, **q_i** (length – normalization) διαιρώντας κάθε συντεταγμένη τους με την L2 νόρμα τους, προκειμένου και να γίνουν και αυτά μοναδιαία όπως αυτά που ορίζουν τη βάση του χώρου *V* στον οποίον ανήκουν. Χάριν απλότητας επιλέγουμε να παραλείψουμε αυτό το βήμα, καθώς η χρησιμότητά του είναι κυρίως να κάνει τα βάρη των πολύ μικρών ή πολύ μεγάλων κειμένων ακόμα πιο συγκρίσιμα.

Όσον αφορά τα πλεονεκτήματα του **Vector Space Model**, ένα σημαντικό σημείο είναι ότι αντί να προσπαθήσει να προσδιορίσει αν ένα κείμενο είναι ή δεν είναι σχετικό, διατάσσει τα κείμενα με βάση τον βαθμό ομοιότητάς τους προς το ερώτημα, έτσι ώστε ένα κείμενο να μπορεί να ανακτηθεί ακόμα και αν ταιριάζει κατά προσέγγιση με το ερώτημα. Ένα μειονέκτημα του ωστόσο είναι ότι οι όροι δεικτοδότησης θεωρούνται ανεξάρτητοι μεταξύ τους. Έτσι, δεν αντιμετωπίζεται επαρκώς το πρόβλημα της Συνωνυμίας και της Πολυσημίας, καθώς δεν λαμβάνονται υπόψη οι συσχετίσεις ανάμεσα στους όρους ή η σημασιολογία τους σε όλο το περιεχόμενο του κειμένου. Σε γενικές γραμμές, το **Vector Space Model**, παρά την απλότητα υλοποίησής του είναι ένα αρκετά αξιόπιστο μοντέλο. Το οποίο μα επιστρέφει αποτελέσματα που είναι δύσκολο να βελτιωθούν χωρίς επέκταση του ερωτήματος ή εφαρμογή ανάδρασης χρήστη (feedback).

1.3 Ερώτημα 3

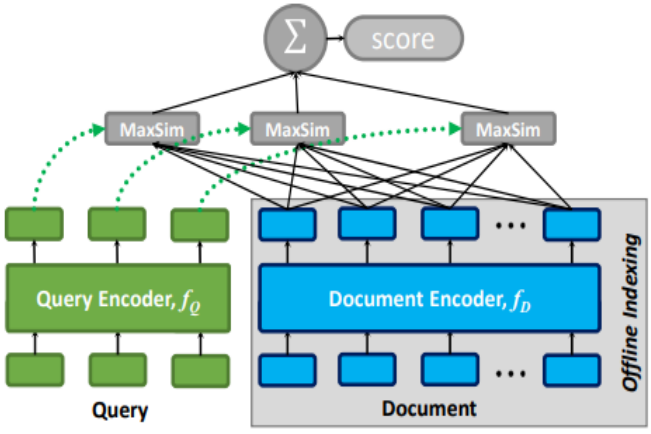


Μια πολύ κοινή τακτική που χρησιμοποιείται στην κατάταξη κειμένων με βάση την ομοιότητά τους είναι η χρήση αρχιτεκτονικών που αξιοποιούν νευρωνικά δίκτυα, οι οποίες είναι γνωστές και ως transformers. Η πιο γνωστή περίπτωση τέτοιου transformer είναι το **BERT**, το οποίο σε συνδυασμό με τη self – supervised προεκπαίδευση (pretraining) του νευρωνικού δικτύου αποτελεί τη βάση για την λειτουργία του Μοντέλου **ColBERT** που μελετάμε στη συγκεκριμένη εργασία.

Εικόνα 3: Υλοποίηση BERT

Πιο συγκεκριμένα, το **ColBERT** εισάγει για πρώτη φορά την έννοια των αρχιτεκτονικών καθυστερημένης αλληλεπίδρασης (late interaction) ανάμεσα στα έγγραφα και τα ερωτήματα, όπου πρώτα κωδικοποιείται το κείμενο και το ερώτημα σε δύο ανεξάρτητα το ένα από το άλλο σύνολα από contextual embeddings E_D και E_Q με τη χρήση **BERT**s και στη συνέχεια μοντελοποιεί αποτελεσματικά και λεπτομερώς τις πολύ μεγάλες ομοιότητες που προκύπτουν. Η μοντελοποίηση γίνεται με πολύ γρήγορους υπολογισμούς και στα δύο σετ από embeddings χωρίς να είναι έτσι απαραίτητος ο προσδιορισμός όλων των πιθανών υποψηφίων για την δημιουργία της τελικής κατάταξης σχετικότητας. Με την εισαγωγή της καθυστέρησης προσφέρεται η δυνατότητα ακόμα πιο ακριβούς υπολογισμού των τιμών ομοιότητας ανάμεσα στα κείμενα και τα ερωτήματα, η οποία δίνεται ως το άθροισμα των τελεστών MaxSim από κάθε embedding. Γενικά, το **ColBERT** απομονώνει τους υπολογισμούς ανάμεσα στα κείμενα και τα ερωτήματα επιτρέποντας έτσι τον εκ των προτέρων υπολογισμό των αναπαραστάσεων των εγγράφων offline. Έτσι, η δεικτοδότηση των εγγράφων μέσα στον κωδικοποιητή γίνεται αποδοτικότερα. Τέλος, ο υπολογισμός των τελεστών MaxSim γίνεται με την ακόλουθη διαδικασία: Αρχικά, τα τελικά emdeddings κανονικοποιούνται ώστε καθένα από αυτά να έχει μοναδιαία L2 νόρμα. Έπειτα, βρίσκουμε τα εσωτερικά γινόμενα των δύο emeddings, τα οποία θα ταυτίζονται με το cosine similarity εφόσον τα embeddings έχουν νόρμα 1. Τέλος, από όλες τις τιμές κρατάμε τη μέγιστη τιμή cosine similarity κάθε διανύσματος ερωτήματος **q** στο E_Q με όλα τα διανύσματα **d** στο E_D και έπειτα συνδυάζουμε τα αποτελέσματα αυτά σε μία τιμή score μέσω του τελεστή αθροίσματος (Σ).

Όλη αυτή η διαδικασία φαίνεται στο διπλανό σχήμα:



Εικόνα 4: Υλοποίηση **ColBERT**

Σε μια γενική περιγραφή οι κωδικοποιητές f_Q και f_D λειτουργούν ως εξής:

1. **Document Encoder f_D**
   Αρχικά, κάθε έγγραφο d τμηματοποιείται σε διακριτά tokens d_1, d_2, …, d_m στα οποία προστίθενται ένα token εκκίνησης [CLS] του **BERT** ακολουθούμενο από ένα token [D] που δηλώνει τη θέση του εκάστοτε εγγράφου στην ακολουθία. Έπειτα, περνάμε σαν είσοδο στον κωδικοποιητή την ακολουθία που προκύπτει μετά τις πιο πάνω προσθήκες και αυτός αναλαμβάνει να την φιλτράρει αφαιρώντας τα embeddings που αντιστοιχούν σε σημεία στίξης με βάση μια προκαθορισμένη λίστα. Έτσι, το πλήθος των τελικών embeddings που προκύπτει είναι σημαντικά μικρότερο σε σχέση με πριν, αφού κάνουμε την υπόθεση ότι τα embeddings των σημείων στίξης δεν επηρεάζουν σημαντικά την απόδοση του μοντέλου.

2. **Query Encoder f_Q**
   Όμοια τμηματοποιούμε κάθε ερώτημα σε διακριτά tokens q_1, q_2, …, q_n στα οποία προσθέτουμε πάλι το token [CLS] του **BERT** μαζί με το πρόθεμα [Q]. Αν μετά από αυτές τις προσθήκες το ερώτημα συνεχίζει να έχει λιγότερο πλήθος tokens από ένα pre – defined threshold, τότε επεκτείνουμε την ακολουθία με ειδικά tokens [mask] μέχρι να φτάσουμε στο επιθυμητό μήκος. Πάλι περνάμε σαν είσοδο την τελική ακολουθία και ο κωδικοποιητής αναλαμβάνει να πραγματοποιήσει μια αναπαράσταση του κάθε token με βάση το περιεχόμενό του.

Συνεπώς, το **ColBERT** είναι αρκετά ευέλικτο όσον αφορά τον τρόπο που αντιμετωπίζει τα ερωτήματα αναζήτησης λόγω της ικανότητάς του να κατανοεί το ευρύτερο σημασιολογικό πλαίσιο. Έτσι, μπορεί να διαχειριστεί λέξεις με πολλαπλές σημασίες με μεγαλύτερη ακρίβεια. προσφέροντας καλύτερη εμπειρία αναζήτησης για τον χρήστη επιστρέφοντας πιο ακριβείς και σχετικές απαντήσεις. Από την άλλη, είναι προφανές ότι η υλοποίησή του απαιτεί μεγάλη υπολογιστική πολυπλοκότητα αυξάνοντας κατ' επέκταση τις ανάγκες μας σε hardware. Τέλος, ένα ακόμα αρνητικό το οποίο αξίζει να αναφέρουμε είναι η απαίτηση σημαντικής ποσότητας μνήμης για την αποθήκευση των προ-εκπαιδευμένων μοντέλων και δεδομένων, γεγονός που καθορίζει σημαντικά και την συνολική απόδοση του μοντέλου.

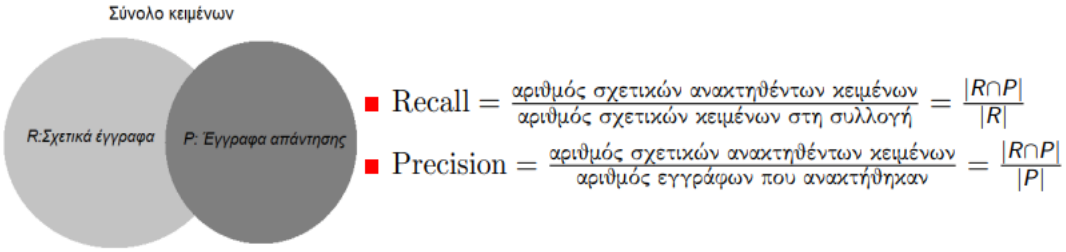
1.4 Ερώτημα 4

Στόχος μας είναι να συγκρίνουμε την απόδοση των δύο μοντέλων χρησιμοποιώντας ένα πλήθος διαφορετικών μετρικών, προκειμένου να έχουμε μεγαλύτερη αξιοπιστία στη τελική μας σύγκριση. Επιλέγουμε να χρησιμοποιήσουμε μετρικές αξιολόγησης σχετικότητας και πιο συγκεκριμένα τις πιο κάτω μετρικές:

1. **Precision – Recall Curve**

   1.1 Η ακρίβεια προσδιορίζει την ικανότητα του μοντέλου ανάκτησης πληροφορίας που διαθέτουμε να επιστρέφει τα κορυφαία σχετικά κείμενα με βάση το score τους κατά τον έλεγχο σχετικότητας. Στη προκειμένη περίπτωση, τα score της τελικής κατάταξης των ανακτημένων εγγράφων από τη συλλογή προκύπτουν από την εφαρμογή των **Vector Space Model** και **ColBERT**. Με άλλα λόγια, εκφράζει το ποσοστό των ανακτημένων κειμένων, τα οποία είναι όντως σχετικά με το ερώτημα του χρήστη. Το πλήθος και ο βαθμός σχετικότητας των σχετικών αυτών κειμένων καθορίζεται από κάποιους ειδικούς, οι οποίοι τα κατατάσσουν αυτά σε κλίμακα εύρους 0 – 2, με 0 να σημαίνει «καθόλου σχετικό» και 2 «αρκετά σχετικό».

   1.2 Η ανάκληση εκφράζει την ικανότητα του μοντέλου μας να επιστρέψει όλη τη συλλογή με τα σχετικά κείμενα, την οποία αναφέραμε ακριβώς πιο πάνω, δηλαδή το ποσοστό των σχετικών κειμένων τα οποία απαντάνε στα ερωτήματα του χρήστη.



Εικόνα 5: Σχέση ανάμεσα σε Ανάκληση και Ακρίβεια

2. **Mean Average Precision (MAP)**

   Η Μέση Ακρίβεια (AP) δίνει έμφαση στην τοποθέτηση των πιο σχετικών εγγράφων στις πιο υψηλές θέσεις της κατάταξης. Πιο συγκεκριμένα, υπολογίζουμε το μέσο όρο των τιμών του precision από όλα τα σχετικά κείμενα της συλλογής για κάθε ερώτημα χρήστη. Συνεπώς, για τον υπολογισμό του MAP θα αξιοποιήσουμε την έννοια της Μέσης Ακρίβειας (AP), καθώς τώρα υπολογίζουμε τη μέση τιμή όλων των τιμών του AP από όλα τα ερωτήματα χρήστη. Δίνεται από τη σχέση:

$$\mathbf{MAP} = \frac{1}{N}\sum_{j=1}^{N}\frac{1}{Q_j}\sum_{i=1}^{Q_j} P(doc_i)$$

   Q_j: ο αριθμός των σχετικών κειμένων για το ερώτημα j
   N: ο αριθμός των ερωτημάτων
   $P(doc_i)$: η τιμή της ακρίβειας του σχετικού κειμένου i

3. **precision@k**

   Υπάρχουν περιπτώσεις στις οποίες είναι απαραίτητο να συγκρίνουμε την απόδοση των μοντέλων ανάκτησης για ατομικές πληροφοριακές ανάγκες. Ο λόγος που μας οδηγεί σε αυτή την απόφαση είναι ότι η χρήση Μέσων Τιμών (MAP) που προκύπτουν από την εκτέλεση διαφόρων ερωτημάτων μπορεί να αποκρύπτει σημαντικές ανωμαλίες στο μοντέλο ανάκτησης που εξετάζεται, οπότε πρέπει για μεγαλύτερη ασφάλεια στην εκτίμηση απόδοσης να μελετήσουμε αν το ένα μοντέλο είναι καλύτερο από το άλλο για κάθε μία από τις πρότυπες πληροφοριακές ανάγκες. Στις περιπτώσεις αυτές, μόνο μία τιμή precision υπολογίζεται για κάθε ερώτημα και η τιμή αυτή θεωρούμε ότι λειτουργεί ως σύνοψη του συνολικού Διαγράμματος Ακρίβειας – Ανάκλησης. Συνήθως αυτή η τιμή είναι η ακρίβεια σε κάποιο προκαθορισμένο επίπεδο k στην ουρά ανάκλησης. Δίνεται από τη σχέση:

   $$\textbf{precision@k} = \frac{1}{k}\sum_{j=1}^{k} r_j$$

   r_j = 1 αν το κείμενο στη θέση j είναι σχετικό αλλιώς 0.

Οι μετρικές 2, 3 μας επιστρέφουν μια μοναδική τιμή, οπότε για να συγκρίνουμε την απόδοση των δύο μοντέλων αρκεί να συγκρίνουμε τις τιμές των **MAP** και **precision@k**. Όσο μεγαλύτερες είναι αυτές οι τιμές, τόσο πιο αποδοτικά ανακτά το μοντέλο μας τα σχετικά έγγραφα. Για τη 1η περίπτωση των γραφικών precision – recall για να συγκρίνουμε τις αποδόσεις, μια καλή τεχνική θα ήταν να υπολογίσουμε το εμβαδόν της περιοχής που περικλείεται ανάμεσα στις γραφικές και στον άξονα x. Όμοια με πριν, όσο μεγαλύτερη είναι η τιμή του εμβαδού που υπολογίζουμε, τόσο πιο αποδοτικό είναι το μοντέλο μας. Ωστόσο, επειδή υπάρχει περίπτωση για κάποια ερωτήματα να λειτουργεί καλύτερα το ένα μοντέλο (**VSM**) και για κάποια το άλλο (**ColBERT**), υπολογίζουμε τη μέση τιμή των εμβαδών για όλα τα ερωτήματα, προκειμένου να έχουμε μια συνολική εικόνα για το ποιο από τα δύο μοντέλα λειτουργεί καλύτερα σε γενικές γραμμές.

1.5 Περιβάλλον Υλοποίησης και Βιβλιοθήκες

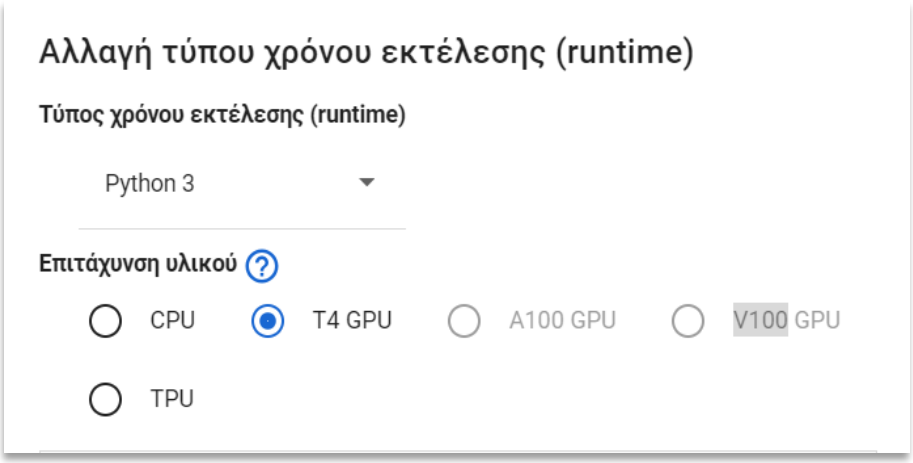
Όλα τα παραπάνω τα οποία εν συντομία αναπτύχθηκαν θεωρητικά, καλούμαστε τώρα να τα υλοποιήσουμε και σε κώδικα. Για τα ερωτήματα 1, 2, 3 επιλέγουμε να χρησιμοποιήσουμε ως περιβάλλον ανάπτυξης αυτό της *PyCharm*, το οποίο διαθέτουμε ήδη εγκατεστημένο από το *JetBrains*. Η έκδοση που χρησιμοποιούμε είναι η 2023.3.2 και επιπλέον οι βιβλιοθήκες που εισάγουμε κατά την εκτέλεση του προγράμματος είναι οι εξής:

1. **numpy**
   Από αυτή τη βιβλιοθήκη αξιοποιούμε τις συναρτήσεις **np.linalg.norm( )** και **np.dot( )**, προκειμένου να εφαρμόσουμε στο Ερώτημα 2 πάνω στα dataframes μας την πράξη της Ευκλείδειας νόρμας (L2) και του εσωτερικού γινομένου στοιχείο προς στοιχείο.
2. **pandas**
   Από αυτή τη βιβλιοθήκη αξιοποιούμε την **pd.Dataframe( )**, προκειμένου να κατασκευάσουμε όλα τα dataframes στα οποία θα αποθηκεύουμε τις διάφορες τιμές που προκύπτουν κατά τη διαδικασία παραγωγής των τελικών scores του cosine similarity στο Ερώτημα 2.
3. **csv**
   Από αυτή τη βιβλιοθήκη αξιοποιούμε τη συνάρτηση **csv.reader(csvfile, dialect='excel', **fmtparams)** προκειμένου να διαβάσουμε στο Ερώτημα 4 τα ανακτηθέντα έγγραφα που μας επιστρέφει το μοντέλο **ColBERT**, τα οποία έχουμε μεταβιβάσει σε ένα αρχείο csv από το περιβάλλον του *Google Colab* σε αυτό της *PyCharm*.
4. **math**
   Από αυτή τη βιβλιοθήκη αξιοποιούμε τη συνάρτηση **math.log10(x)** για τον υπολογισμό του δεκαδικού λογαρίθμου στις συχνότητες TF και IDF όταν καθορίζουμε τα βάρη των διανυσμάτων μας στο Ερώτημα 2.
5. **pyplot**
   Από αυτή τη βιβλιοθήκη αξιοποιούμε όλες τις απαραίτητες συναρτήσεις για το Ερώτημα 4 προκειμένου να χαράξουμε σωστά τις γραφικές παραστάσεις των δύο μοντέλων στη μετρική Precision – Recall. Τέτοιες συναρτήσεις είναι οι: **plt.figure( )**, **plt.plot( )** κ. ο. κ.
6. **collections**
   Από αυτή τη βιβλιοθήκη εισάγουμε τη κλάση **class collections.defaultdict(default_factory=None, /[, ...])**, η οποία είναι ιδιαίτερα χρήσιμη για την κατασκευή του **Ανεστραμμένου Ευρετηρίου** στο Ερώτημα 1, καθώς μας επιτρέπει να προσθέτουμε αντικείμενα (δηλαδή δυάδες της μορφής [ID κειμένου, πλήθος εμφανίσεων όρου]) στη λίστα που αντιστοιχεί στο εκάστοτε key (δηλαδή κάθε μοναδικός όρος) χωρίς να χρειάζεται να ελέγξουμε αν αυτό υπάρχει ήδη στο λεξικό.
7. **os**
   Από αυτή τη βιβλιοθήκη αξιοποιούμε τη συνάρτηση **os.chdir(path)** προκειμένου να μεταβούμε στο Ερώτημα 1 στο φάκελο docs που περιέχει τα 1209 κείμενα της συλλογής και στη συνέχεια να ακολουθήσουμε τη διαδικασία που περιγράφεται στην υποενότητα **2.1** για την κατασκευή του **Ανεστραμμένου Ευρετηρίου**.
8. **ast**
   Από αυτή τη βιβλιοθήκη αξιοποιούμε τη συνάρτηση **ast.literal_eval(node_or_string)**, καθώς σε κάθε ερώτημα στο αρχείο CSV όπου

έχουμε αποθηκευμένα τα αποτελέσματα του **ColBERT** αντιστοιχεί ένα αλφαριθμητικό το οποίο αντιπροσωπεύει τη λίστα με τα ταξινομημένα ανακτηθέντα κείμενα. Συνεπώς, είναι απαραίτητο κάθε ένα από τα αλφαριθμητικά αυτά να μετατραπούν σε πραγματικές λίστες με περιεχόμενο ακεραίους αριθμούς, δηλαδή τα IDs των κειμένων.

9. **re**

Επιπλέον, για την υλοποίηση του Μοντέλου **ColBERT** εργαστήκαμε, όπως προτείνεται, στο περιβάλλον του *Google Colab* εφαρμόζοντας τις εξής ρυθμίσεις:



Εικόνα 6: Ρυθμίσεις *Google Colab*

Η χρήση της *T4 GPU* για την υλοποίηση του **ColBERT** παρέχει ένα σημαντικό προβάδισμα σε ταχύτητα και απόδοση σε σύγκριση με τη χρήση της *CPU*, κάτι που είναι κρίσιμο για απαιτητικές εφαρμογές μηχανικής μάθησης, όπως στη δικιά μας περίπτωση. Πιο συγκεκριμένα, οι περισσότερες βιβλιοθήκες deep learning, όπως η **PyTorch** που αναλύουμε και παρακάτω, είναι σχεδιασμένες για να λειτουργούν καλύτερα με *GPU* λόγω της ικανότητάς της να επεξεργάζονται μεγάλους όγκους δεδομένων και πολύπλοκα νευρωνικά δίκτυα αποδοτικότερα.

Αρχικά, εγκαθιστούμε το πακέτο **virtualenv**, το οποίο είναι απαραίτητο για την δημιουργία ενός εικονικού περιβάλλοντος. Έπειτα, δημιουργούμε ένα καινούριο εικονικό περιβάλλον που το ονομάζουμε **myenv** και το ενεργοποιούμε ώστε να απομονώσει το περιβάλλον της *Python* για να εμποδίσει τυχόν επικαλύψεις με άλλα πακέτα. Για την υλοποίηση εισάγουμε ακόμα το προ – εκπαιδευμένο μοντέλο **ColBERTv2** του *Stanford* από το αντίστοιχο repository **stanford – futuredata** του *GitHub*.

Οι βιβλιοθήκες που χρησιμοποιούμε είναι:

1. **torch**: είναι συντομογραφία για την ανοικτού κώδικα βιβλιοθήκη μηχανικής μάθησης **Pytorch** η οποία έχει σχεδιαστεί για την εκτέλεση deep learning και μηχανικής μάθησης μέσω νευρωνικών δικτύων συνεπώς είναι απαραίτητη για την υλοποίηση του **ColBERT**.
2. **sys**: χρησιμοποιείται η εντολή **sys.path.insert(0, 'ColBERT/')** για να προσθέσει το μονοπάτι του **ColBERT** στο περιβάλλον της *Python* που χρησιμοποιούμε.
3. **faiss**: είναι ανοικτού κώδικα βιβλιοθήκη και έχει σχεδιαστεί για αποτελεσματική αναζήτηση ομοιότητας και ομαδοποίηση συνόλων δεδομένων σε χώρους υψηλών διαστάσεων που και αυτό είναι απαραίτητο για το **ColBERT**.

4. **cuda**: χρησιμοποιείται για την επιτάχυνση της GPU.
5. **colbert**: χρησιμοποιείται ώστε να πάρουμε όλες τις απαραίτητες βιβλιοθήκες για την σωστή εκτέλεση του μοντέλου. Θα χρησιμοποιήσουμε τις βιβλιοθήκες **Indexer**, **Searcher**, **Run**, **RunConfig**, **ColBERTConfig**, **Queries** και **Collection**.

Επίσης, συνδέουμε το *Google Colab* με το drive μας, ώστε να έχει πρόσβαση στα απαραίτητα αρχεία και φακέλους της συλλογής μας.

```
from google.colab import drive
drive.mount('/content/drive')
```

ΥΛΟΠΟΙΗΣΗ

2.1 Ερώτημα 1

Αρχικά ορίζουμε - αρχικοποιούμε 3 μεταβλητές που θα χρησιμοποιηθούν στην συνέχεια:

1. **inverted_index**: είναι ένα defaultdict, το οποίο αρχικοποιείται ως κενή λίστα. Το deafultdict είναι μια υποκλάση της κλάσης dictionary και επιστρέφει ένα αντικείμενο που μοιάζει με λεξικό. Χρησιμοποιείται ώστε να δημιουργήσει το ανεστραμμένο ευρετήριο, όπου κάθε λέξη θα συσχετίζεται με μια λίστα πλειάδων που θα περιέχουν τα IDs των κειμένων και τον αριθμό εμφανίσεων της λέξης στο αντίστοιχο κείμενο.
2. **document_count:** είναι dictionary και θα χρησιμοποιηθεί για να αποθηκεύσουμε τα id των κειμένων που εμφανίζεται κάθε λέξη, για να υπολογίζεται το πλήθος των κειμένων σε επόμενα ερωτήματα.
3. **path:** μεταβλητή η οποία περιέχει την διαδρομή που βρίσκεται ο φάκελος με την συλλογή των κειμένων (docs).

```
# Question 1
# defaultdict to store the inverted index
inverted_index = defaultdict(list)

# Dictionary to store the  documents each word appears in
document_count = {}

path = (r"C:\Users\me\PycharmProjects\pythonProject1\docs")
os.chdir(path)
```

Στην συνέχεια ξεκινάει η διαδικασία δημιουργίας του Ανεστραμμένου Ευρετηρίου. Η διαδικασία αρχίζει με μια επανάληψη που διατρέχει κάθε αρχείο μέσα στο φάκελο docs με την χρήση της εντολής **os.listdir(path)** και αποθηκεύουμε στην μεταβλητή **file_path** την διαδρομή για το συγκεκριμένο αρχείο (**os.path.join(path, file)**). Ανοίγουμε, διαβάζουμε το αρχείο και στη μεταβλητή **dictionary** αποθηκεύονται οι λέξεις που περιέχει το συγκεκριμένο κείμενο. Επίσης, δημιουργούμε ένα ακόμη λεξικό (**count**) που θα χρησιμοποιηθεί για να αποθηκεύουμε την συχνότητα εμφάνισης κάθε λέξης στο τρέχων κείμενο. Ύστερα, διατρέχουμε κάθε λέξη του dictionary και αν βρεθεί ανανεώνουμε το **count** για την συγκεκριμένη λέξη. Επιπλέον, για κάθε λέξη που υπάρχει στο τρέχων κείμενο εισάγουμε στο **inverted_index** το ID (είναι το όνομα του αρχείου) και την συχνότητα εμφάνισης (**count**) αν η λέξη υπάρχει ήδη, αλλιώς εισάγεται η λέξη μαζί με τις αντίστοιχες πληροφορίες. Τέλος, στον ίδιο βρόγχο επανάληψης ανανεώνουμε το **document_count** για κάθε λέξη του κειμένου αν υπάρχει στο dictionary προσθέτουμε το id του, αλλιώς προσθέτουμε όλη την λέξη μαζί με το ID.

```
for file in os.listdir(path):
    file_path = os.path.join(path, file)
    with open(file_path, 'r') as folder:
        text = folder.read()

        dictionary = text.split()

        # Create a dictionary to store word counts for each document
        count = {}

        for word in dictionary:
            count[word] = count.get(word, 0) + 1

        # Update the inverted index with word counts for the current document
        for word, count in count.items():
            inverted_index[word].append((file, count))

            # Update the document count for the current word
            if word in document_count:
                document_count[word].add(file)
            else:
                document_count[word] = {file}
```

Τέλος εκτυπώνουμε το τελικό **inverted_index** και το αποθηκεύουμε σε ένα αρχείο CSV.

```
inverted_index_csv = r'C:\Users\me\PycharmProjects\pythonProject1\inverted_index.csv'

# Print the inverted index
for word, documents in inverted_index.items():
    print(f'{word}: {documents}')

# Save the inverted index into a CSV file
with open(inverted_index_csv, 'w', newline='', encoding='utf-8') as csvfile:
    csv_writer = csv.writer(csvfile)
    csv_writer.writerow(['Word', 'Documents'])

    for word, documents in inverted_index.items():
        csv_writer.writerow([word, documents])
```

2.2 Ερώτημα 2

Ξεκινάμε με τον υπολογισμό του παράγοντα IDF. Αρχικά, για κάθε λέξη **word** που έχουμε στο ευρετήριό μας διατρέχουμε το **document_count**, το οποίο είναι λεξικό με κλειδί τη συγκεκριμένη λέξη και περιεχόμενο τη λίστα των κειμένων στο οποίο εμφανίζεται αυτή η λέξη. Αποθηκεύουμε μετά τη σάρωση στη λίστα **documents** τα κείμενα όπου εμφανίζεται η κάθε λέξη και υπολογίζουμε το **idf** ως τον δεκαδικό λογάριθμο του μήκους του φακέλου που έχουμε όλα τα έγγραφα της συλλογής διά το μήκος της λίστας **documents**. Έτσι καταφέρνουμε να βρούμε τον λόγο από όλα τα κείμενα της συλλογής προς τα κείμενα που περιλαμβάνουν τη κάθε λέξη. Το **idf** είναι επίσης λεξικό με κλειδί το συγκεκριμένο term και περιεχόμενο κλειδιού τη τιμή του IDF του κλειδιού. Αφού ολοκληρώσουμε την επεξεργασία και γεμίσουμε το λεξικό με τιμές, το μετατρέπουμε σε dataframe και το αποθηκεύουμε σε ένα αρχείο CSV για να μπορούμε να το επεξεργαστούμε πιο εύκολα.

```
# Question 2
# Calculate IDF for each word and store it in a dictionary
idf = {}
for word, documents in document_count.items():
    idf_value = math.log10((len([f for f in os.listdir(path)])) / len(documents))
    idf[word] = idf_value

# Create a DataFrame with IDF values
idf = pd.DataFrame.from_dict(idf, orient='index', columns=['IDF'])

csv_file_path = r"C:\Users\me\PycharmProjects\pythonProject1\idf.csv"
# Save the DataFrame to a CSV file
idf.to_csv(csv_file_path, index=False)
```

Με όμοιο τρόπο ακριβώς κατασκευάζουμε και το dataframe **tf** με τη μόνη διαφορά ότι τώρα ορίζουμε ένα μετρητή **count**, ο οποίος μεταβαίνει στο Ανεστραμμένο Ευρετήριο και εντοπίζει για κάθε κείμενο **document** το πλήθος των εμφανίσεων κάθε λέξης του **word** που είναι αποθηκευμένο στη λίστα **occurrences**. Έτσι, υπολογίζουμε το TF κάθε term κάθε κειμένου ξεχωριστά ως το δεκαδικό λογάριθμο της τιμής **count** συν 1. Στο τελικό dataframe που προκύπτει συμπληρώνουμε μηδενικά για τα terms τα οποία δεν υπάρχουν καθόλου μέσω της συνάρτησης **fillna( )**.

```
# Create a dictionary to store TF values for each word in each document
tf = {}
for word, occurrences in inverted_index.items():
    for document, count in occurrences:
        if document in tf:
            tf[document][word] = 1 + math.log10(count)
        else:
            tf[document] = {word: 1 + math.log10(count)}

# Create a DataFrame where rows are document titles and columns are words with their TF values
tf = pd.DataFrame.from_dict(tf, orient='index')
tf.fillna(0, inplace=True)  # Fill missing values with 0

csv_file_path = r"C:\Users\me\PycharmProjects\pythonProject1\doc_tf.csv"
# Save the DataFrame to a CSV file
tf.to_csv(csv_file_path, index=False)
```

Για τον υπολογισμό του βάρους tf-idf πολλαπλασιάζουμε στη συνέχεια κάθε στήλη (term) του dataframe **tf** με την αντίστοιχη τιμή του dataframe **idf** και το τελικό αποτέλεσμα το αποθηκεύουμε στο dataframe **tfidf**, το οποίο επίσης αποθηκεύουμε σε ένα CSV αρχείο για διευκόλυνση.

```
# Iterate through the TF and IDF DataFrames to calculate TF-IDF
tfidf = tf.copy()
# Multiply each TF value by the corresponding IDF value
for column in tfidf.columns:
    tfidf[column] = tfidf[column] * idf.loc[column, 'IDF']

csv_file_path = r"C:\Users\me\PycharmProjects\pythonProject\doc_weights.csv"
# Save the DataFrame to a CSV file
tfidf.to_csv(csv_file_path, index=False)
```

Στη συνέχεια, θέλουμε να υπολογίσουμε τα αντίστοιχα βάρη για κάθε ερώτημα από τη συλλογή ερωτημάτων χρήστη, οπότε πάλι θα αποθηκεύσουμε την απόλυτη διαδρομή του αρχείου με τα ερωτήματα στη μεταβλητή **query_path**, θα διαβάσουμε κάθε ερώτημα που αντιστοιχεί σε μία γραμμή του αρχείου μέσω της **readlines( )** και θα το αποθηκεύσουμε στη λίστα **questions**. Έπειτα, κατασκευάζουμε μια λίστα **all_words** με όλες τις λέξεις του Ανεστραμμένου Ευρετηρίου ως περιεχόμενο και ένα κενό dataframe **df**, όπου θα αποθηκεύουμε τους παράγοντες TF κάθε ερωτήματος. Αρχικοποιούμε πάλι με 0 το μετρητή **count** όπως και πριν και ξεκινάμε να διατρέχουμε τη λίστα με τα ερωτήματα. Έτσι, για κάθε ερώτημα **question** της λίστας, το μετατρέπουμε αρχικά με την **upper( )** σε κεφαλαία γράμματα για να ταιριάζει με το case sensitivity των terms του Ανεστραμμένου Ευρετηρίου και αποθηκεύουμε όλες τις διακριτές του λέξεις μέσω της **split( )** στη λίστα **words**. Στη συνέχεια, για κάθε ερώτημα βρίσκουμε τις κοινές λέξεις του Ανεστραμμένου Ευρετηρίου και του ερωτήματος παίρνοντας τη τομή των δύο συνόλων **words** και **all_words** και τις αποθηκεύουμε στη λίστα **common_words**. Ορίζουμε και μία λίστα **word_counts** που διατηρεί τις τιμές του μετρητή για κάθε κοινή λέξη, την οποία αρχικοποιούμε με 0. Τέλος, για κάθε κοινή λέξη ψάχνουμε στο αλφαριθμητικό **question** πόσες φορές εμφανίζεται μέσω της **findall( )**, η οποία ωστόσο μας επιστρέφει τη λέξη όσες φορές αυτή εμφανίζεται στην πρόταση, γι' αυτό και παίρνουμε το μήκος αυτής της λίστας, μέσω της **len( ),** ως τιμή για το μετρητή **count**. Το TF κάθε λέξης του ερωτήματος δίνεται πάλι ως ο δεκαδικός λογάριθμος της τιμής count συν 1 και αποθηκεύεται στο αρχικό dataframe **df**.

```
query_path = r"C:\Users\me\PycharmProjects\pythonProject1\Queries_20"
# Read questions from the file
with open(query_path, 'r') as file:
    questions = file.readlines()

# Create a list of words from the inverted index
all_words = list(inverted_index.keys())

# Create a DataFrame with zeros
df = pd.DataFrame(0, index=range(len(questions)), columns=all_words)

# Iterate through questions and update the DataFrame
count = 0
for i, question in enumerate(questions):
    words = question.upper().split()
    # Find common words between the question and the inverted index
```

```
    common_words = set(words) & set(inverted_index)
    word_counts = {word: 0 for word in common_words}

    for word in common_words:
        # Count the number of occurrences of the word in the question
        pattern = r'\b' + re.escape(word) + r'\b'
        count = len(re.findall(pattern, question.upper()))
        # Update the DataFrame with the log-transformed count
        if count > 0:
            df.at[i, word] = 1 + math.log10(count)

df = df.fillna(0)
csv_file_path = r"C:\Users\me\PycharmProjects\pythonProject1\query_tf.csv"
# Save the DataFrame to a CSV file
df.to_csv(csv_file_path, index=False)
```

Για τον υπολογισμό του βάρους tf-idf των ερωτημάτων όμοια με πριν πολλαπλασιάζουμε στη συνέχεια κάθε στήλη (term) του dataframe **df** με την αντίστοιχη τιμή του dataframe **idf** και το τελικό αποτέλεσμα το αποθηκεύουμε στο dataframe **query_tfidf**, το οποίο επίσης αποθηκεύουμε σε ένα CSV αρχείο.

```
query_tfidf = df.multiply(idf['IDF'], axis=1)
csv_file_path = r"C:\Users\me\PycharmProjects\pythonProject1\query_weights.csv"
# Save the DataFrame to a CSV file
query_tfidf.to_csv(csv_file_path, index=False)
```

Το τελικό μας βήμα αποτελεί ο υπολογισμός των Ευκλείδειων νορμών κάθε γραμμής των **tfidf** και **query_tfidf**, το οποίο πραγματοποιείται μέσω της συνάρτησης **linalg.norm( )** της βιβλιοθήκης numpy. Τα τελικά αποτελέσματα για τις νόρμες κειμένων και ερωτημάτων τα αποθηκεύουμε στα αντίστοιχα dataframes **euclidean_norms_d** και **euclidean_norms_q**. Το εσωτερικό γινόμενο υπολογίζεται μέσω της συνάρτησης **dot( )** με ορίσματα τις τιμές των αντίστοιχων γραμμών των dataframes **tfidf** και **query_tfidf.T**. Ο λόγος για τον οποίο εφαρμόζουμε τον τελεστή αναστροφής (T) στο διάνυσμα των βαρών των ερωτημάτων είναι για να υπολογιστεί σωστά το εσωτερικό γινόμενο, καθώς όπως γνωρίζουμε και από τη Γραμμική Άλγεβρα ισχύει ότι: $<\mathbf{u}, \mathbf{w}> := \mathbf{u}\mathbf{w}^T$. Τα τελικά αποτελέσματα αποθηκεύονται και αυτά σε ένα dataframe **dot_product**, ενώ τα γινόμενα των νορμών **euclidean_norms_d** και **euclidean_norms_q** αποθηκεύονται στο datafame **norm_product**, όπου πολλαπλασιάζουμε κάθε γραμμή του πρώτου με κάθε στήλη του δεύτερου. Προφανώς, οι διαστάσεις και για τα δύο τελευταία dataframes που αναφέραμε είναι 1209x20, γεγονός που επιβεβαιώνει ότι έχουμε πράξει ορθά, καθώς κάθε κελί των δύο πινάκων αντιπροσωπεύει ένα συγκεκριμένο κείμενο ως προς ένα συγκεκριμένο ερώτημα. Για να υπολογίσουμε λοιπόν το dataframe **cosine_similarity** απλώς διαιρούμε το **dot_product** με το **norm_product** μέσω της **div( )** και το τελικό dataframe το οποίο προκύπτει θα είναι και το ζητούμενο.

```
# Calculate the Euclidean norm for each row of doc_weights
euclidean_norms = np.linalg.norm(tfidf.values, axis=1)
# Create a DataFrame with the Euclidean norm values
euclidean_norms_d = pd.DataFrame({'Euclidean Norm': euclidean_norms}, index=tfidf.index)
```

```
# Calculate the Euclidean norm for each row of query_weights
euclidean_norms = np.linalg.norm(query_tfidf.values, axis=1)
# Create a DataFrame with the Euclidean norm values
euclidean_norms_q = pd.DataFrame({'Euclidean Norm': euclidean_norms}, index=query_tfidf.index)

# Calculate the dot product
dot_products = np.dot(tfidf.values, query_tfidf.values.T)
# Create a DataFrame with the dot product values
dot_product = pd.DataFrame(dot_products, index=tfidf.index, columns=query_tfidf.index)

# Calculate the product of the Euclidean norms
norm_product = euclidean_norms_d.values @ euclidean_norms_q.values.T
# Create a DataFrame with the result
norm_product = pd.DataFrame(norm_product, index=euclidean_norms_d.index,
columns=euclidean_norms_q.index)

# Calculate the cosine similarity
cosine_similarity = dot_product.div(norm_product)
print(cosine_similarity)
```

2.3 Ερώτημα 3

Αρχικά, όλα τα κείμενα και τα απαραίτητα αρχεία είναι αποθηκευμένα στο φάκελο **docs** μέσα στο drive και είναι σε συμπιεσμένα σε .zip μορφή, οπότε τον αποσυμπιέζουμε και τα περνάμε στο μονοπάτι **"/content/docs"**.

```
!unzip /content/drive/MyDrive/ir/docs.zip -d /content
```

Δημιουργείται έτσι η συνάρτηση **read_documents( )**, η οποία διαβάζει κάθε αρχείο που βρίσκεται μέσα στο φάκελο docs με τον ίδιο τρόπο όπως και στο **Vector Space Model** και αναπαριστά κάθε αρχείο ως λίστα με id το όνομα του αρχείου και το περιεχόμενο του εγγράφου.

```
def read_documents(directory_path):
    documents = []
    for file in os.listdir(directory_path):
        file_path = os.path.join(directory_path, file)
        with open(file_path, 'r', encoding="utf8") as folder:
            text = folder.read()
            doc_id = file
            document_tuple = (doc_id, text)
            documents.append(document_tuple)
    return documents
```

Η συνάρτηση **read_queries( )**, διαβάζει το αρχείο που περιέχει τα ερωτήματα και τα αποθηκεύει σε λίστα.

```
def read_queries(file_path):
    queries = []
    with open(file_path, 'r', encoding="utf8") as file:
        text = file.read()
        queries_list = text.split('\n')
        queries_list = [query.strip() for query in queries_list if query.strip()]
        queries.extend(queries_list)
    return queries
```

Στην συνέχεια ορίζουμε τα configurations του **ColBERT** ως εξής:

- **nbits**: κωδικοποιεί κάθε διάσταση με 2 bit.
- **max_id** : Ορίζουμε το μέγιστο ID των αρχείων που είναι 1209 και συμβολίζει ότι στα 1209 ID θα χρησιμοποιηθεί ο Indexer.
- **index_name** : το όνομα του δείκτη.
- **checkpoint**: περιέχει το μονοπάτι που βρίσκεται το checkpoint.

Όσον αφορά το indexing, γίνεται με βάση το προ - εκπαιδευμένο μοντέλο και με βάση τα configurations που ορίσαμε παραπάνω. Δημιουργεί έναν δείκτη και στην συνέχεια δεικτοδοτεί τα 1209 έγγραφα.

```
# Indexing
with Run().context(RunConfig(nranks=1, experiment='indexing')):
   config = ColBERTConfig(doc_maxlen=doc_maxlen, nbits=nbits, kmeans_niters=4)
   indexer = Indexer(checkpoint=checkpoint, config=config)
   indexer.index(name=index_name, collection=[doc[1] for doc in documents[:max_id]], overwrite=True)
```

Στην συνέχεια δημιουργούμε τον **Searcher** και για κάθε **query** εκτυπώνουμε τα ανακτημένα κείμενα αντιστοιχίζοντας το **passage_id** με το σωστό **doc_id**. Επίσης αποθηκεύομε σε ένα αρχείο CSV τα **query** και για κάθε **query** τα ανακτημένα κείμενα, ώστε να μπορούμε να τα επεξεργαστούμε στο Ερώτημα 4.

```
# Searching
with Run().context(RunConfig(experiment='indexing')):
   searcher = Searcher(index=index_name, collection=[doc[1] for doc in documents])


# Create a dictionary to store lists of document IDs for each query
colbert_result = {}


# Iterate over all queries
for query in queries:
   print(f"#> {query}")
   results = searcher.search(query, k=1209)  # Retrieve all passages
   result_list = []
   for i, (passage_id, passage_rank, passage_score) in enumerate(zip(*results)):
      doc_id = documents[int(passage_id)][0]
      print(f"\t [{passage_rank}] \t\t {passage_score:.1f} \t\t  Document ID: {doc_id}")
      result_list.append(doc_id)
   colbert_result[query] = result_list




# Display or save the dictionary
print(colbert_result)

# Save the dictionary to a CSV file
csv_file_path = '/content/colbert_result.csv'
with open(csv_file_path, 'w', newline='') as csvfile:
   writer = csv.writer(csvfile)

   # Write the header
   writer.writerow(['Query', 'DocumentIDs'])

   # Write the data
```

```
    for query, document_list in colbert_result.items():
        writer.writerow([query, document_list])

print(f"Query document lists saved to {csv_file_path}")
```

2.4 Ερώτημα 4

Για το συγκεκριμένο ερώτημα αρχικά θα ταξινομήσουμε σε φθίνουσα σειρά τα αποτελέσματα του **Vector Space Model** με βάση το cosine similarity, τα αποθηκεύουμε στο dataframe **sorted_doc_ids** και τα εξάγουμε και στο CSV "**cosine_similarity.csv"**. Στην συνέχεια, αντικαθιστούμε τις τιμές του dataframe **cosine similarity** με το ID κάθε κειμένου στο οποίο αντιστοιχούν, το οποίο το μετατρέπουμε σε ακέραιο για κάθε ερώτημα, ώστε τα πιο σχετικά ID να βρίσκονται στην κορυφή της λίστας και τα λιγότερο σχετικά στο τέλος. Τα αποτελέσματα τα αποθηκεύουμε σε μια λίστα **retrieved_docs** στην οποία κλειδί είναι το ID του ερωτήματος και το περιεχόμενο είναι λίστα που περιέχει τα id των κειμένων.

```
# Question 4
# Precision - Recall and MAP Metrics
# Sort each column and replace values with doc IDs
sorted_doc_ids = cosine_similarity.apply(lambda col: col.sort_values(ascending=False).index)
csv_file_path = r"C:\Users\me\PycharmProjects\pythonProject1\cosine_similarity.csv"
# Save the DataFrame to a CSV file
sorted_doc_ids.to_csv(csv_file_path, index=False)
# Convert document IDs to integers and get the retrieved docs
retrieved_docs = {query_id: [int(doc_id) for doc_id in doc_ids] for query_id, doc_ids in
sorted_doc_ids.items()}
```

Σχετικά με τα αποτελέσματα του **ColBERT** τα έχουμε μεταβιβάσει από το περιβάλλον του *Google Colab* και αποθηκεύσει σε CSV, του οποίου η πρώτη στήλη περιέχει τα ερωτήματα και η δεύτερη τα id των κειμένων που είναι ήδη τοποθετημένα σε φθίνουσα σειρά. Οπότε διατρέχουμε κάθε γραμμή του CSV και ελέγχουμε αν στη δεύτερη στήλη τα id των κειμένων είναι ακέραιοι και σε λίστα. Αν είναι τότε τα αποθηκεύουμε στο λεξικό **query_docs** όπου το κλειδί είναι το ID των ερωτημάτων και τα περιεχόμενα είναι λίστα με όλα τα κείμενα.

```
colbert_path = r"C:\Users\me\PycharmProjects\pythonProject1\colbert_result.csv"
query_docs = {}
# Read the ColBERT results into a DataFrame
with open(colbert_path, 'r') as csvfile:
    csvreader = csv.reader(csvfile)
    next(csvreader)  # Skip the header row

    for query_id, row in enumerate(csvreader):
        # Parse the string representation of the list into an actual list
        doc_ids = ast.literal_eval(row[1])
        # Ensure doc_ids is a list and contains integers
        if isinstance(doc_ids, list) and all(isinstance(id, int) for id in doc_ids):
                query_docs[query_id] = doc_ids
```

Αναφορικά με το αρχείο CSV **"Relevant_20"**, κάθε γραμμή του περιέχει τα σχετικά κείμενα για κάθε ερώτημα, οπότε διατρέχουμε το αρχείο, μετατρέπουμε τα id των κειμένων σε ακεραίους και τα αποθηκεύουμε στην λίστα **relevant_docs**, όπου το κλειδί είναι το id των ερωτημάτων και τα περιεχόμενα είναι λίστα με όλα τα κείμενα.

```
# Get the relevant docs
file_path = r"C:\Users\me\PycharmProjects\pythonProject1\Relevant_20"
relevant_docs = {}
with open(file_path, 'r') as file:
    for query_id, line in enumerate(file):
        # Convert all space-separated numbers in the line to integers and store them as a set
        doc_ids = set(map(int, line.split()))
        # Assign this set to the corresponding query ID
        relevant_docs[query_id] = doc_ids
```

Τώρα θα υλοποιήσουμε τις μετρικές precision - recall, preciosion@k και Mean Average Length (MAP). Οι τιμές των precision, recall και η MAP θα υπολογιστούν στην ίδια συνάρτηση **calculate_metrics( )**, ενώ η τιμή precision@k υπολογίζεται από την συνάρτηση **precision_at_k( )**.

Όσον αφορά την συνάρτηση **calculate_metrics( )**, δέχεται σαν όρισμα την λίστα με τα σχετικά κείμενα και την λίστα με τα ανακτημένα. Αρχικά ορίζονται δυο μεταβλητές που θα χρησιμοποιηθούν σαν μετρητές, η **ret_count** που είναι μετρητής για τα ανακτημένα κείμενα και η **rel_count**, η οποία είναι μετρητής για τα σχετικά. Επίσης ορίζονται οι λίστες **precisions** και **recalls**. Ύστερα, διατρέχουμε με μια επανάληψη όλα τα κείμενα που βρίσκονται μέσα στη λίστα **retrieved_docs** και κάθε φορά που μεταβαίνουμε σε καινούριο κείμενο αυξάνεται ο μετρητή **ret_count** κατά 1. Ελέγχουμε αν το κείμενο που εξετάζουμε αυτή τη στιγμή βρίσκεται και στην λίστα **relevant_docs** και αν βρίσκεται, τότε σημαίνει ότι είναι σχετικό κείμενο, οπότε αυξάνουμε και τον μετρητή **rel_count** κατά 1. Υπολογίζουμε τις τιμές precision και το recall για το συγκεκριμένο κείμενο εφαρμόζοντας τους τύπους που έχουν αναφερθεί στην υποενότητα **1.4** και τα αποθηκεύουμε στις λίστες **precisions** και **recalls** αντίστοιχα, οι οποίες επιστρέφονται όταν καλούμε την συνάρτηση. Για τον υπολογισμό της MAP θα υπολογίσουμε έξω από τον βρόχο επανάληψης τον μέσο όρο των precisions (**avg_precision**) και το αποτέλεσμα επιστρέφεται επίσης μαζί με τα υπόλοιπα.

```
def calculate_metrics(relevant_docs, retrieved_docs):
    ret_count = 0
    rel_count = 0
    precisions = []
    recalls = []

    # Iterate through each relevant document
    for doc in retrieved_docs:
        ret_count += 1
        # Check if the relevant document is in the retrieved documents
        if doc in relevant_docs:
            rel_count += 1
            precision = rel_count / ret_count
            precisions.append(precision)
            recall = rel_count / len(relevant_docs)
            recalls.append(recall)
    avg_precision = sum(precisions) / len(relevant_docs)

    return precisions, recalls, avg_precision
```

Για να μπορούμε να συγκρίνουμε την απόδοση των δύο μοντέλων με βάση το γράφημα θα δημιουργήσουμε μια συνάρτηση **calculate_area( )**, η οποία θα υπολογίζει το εμβαδόν που περικλείεται ανάμεσα στον άξονα x και κάθε γραφική για όλα τα ερωτήματα. Ο υπολογισμός του εμβαδού γίνεται με βάση τον ορισμό του Riemann, όπου προσεγγίζουμε το εμβαδό αθροίζοντας τα εμβαδά μικρότερων ορθογωνίων παραλληλογράμμων με πλάτος την απόσταση δύο διαδοχικών τιμών της λίστας **recalls** (x – axis) και ύψος το ημιάθροισμα δύο διαδοχικών τιμών της λίστας **precisions** (y – axis). Το πλήθος των ορθογωνίων που θα χρησιμοποιήσουμε θα είναι ίσο με το πλήθος των τιμών της λίστας recalls, δηλαδή για κάθε ερώτημα χωρίζουμε τον άξονα των τετμημένων με βάση το πλήθος των τιμών ανάκλησης σε αντίστοιχα υποδιαστήματα. Τέλος, το εμβαδό **area** που θα προκύπτει κάθε φορά θα είναι το άθροισμα των εμβαδών των μέχρι τώρα ορθογωνίων που έχουμε υπολογίσει με το εμβαδόν του ορθογωνίου που υπολογίζουμε τη συγκεκριμένη χρονική στιγμή. Η συνάρτηση θα δέχεται σαν όρισμα την λίστα **recalls**, τη λίστα **precisions** που θα έχει δημιουργηθεί από την προηγούμενη συνάρτηση και θα επιστρέφει τη τιμή της μεταβλητής **area**.

```
def calculate_area(recalls, precisions):
    area = 0.0
    for i in range(1, len(recalls)):
        # Calculate the area of the trapezoid
        width = recalls[i] - recalls[i-1]
        height = (precisions[i] + precisions[i-1]) / 2
        area += width * height
        return area
```

Για την μετρική precision@k, όπως έχουμε αναφέρει ήδη δημιουργούμε την συνάρτηση **precision_at_k( )**, οποία παίρνει σαν ορίσματα την λίστα με τα σχετικά κείμενα, την λίστα με τα ανακτημένα και το όριο **k**. Ελέγχει άμα η λίστα των ανακτηθέντων κειμένων είναι μεγαλύτερη του ορίου **k** και άμα είναι, τότε η λίστα «κόβεται» και πλέον περιέχει μόνο τα πρώτα **k** κείμενα. Στην συνέχεια, διατρέχουμε την καινούρια λίστα και για κάθε κείμενο ελέγχουμε αν είναι σχετικό και επιστρέφουμε τη τιμή 1 αν είναι. Υπολογίζουμε το άθροισμα των σχετικών κειμένων **relevant_count** και η συνάρτηση επιστρέφει το αποτέλεσμα της διαίρεσης του αθροίσματος προς το όριο **k**.

```
def precision_at_k(relevant_docs, retrieved_docs, k):
    if len(retrieved_docs) > k:
        retrieved_docs = retrieved_docs[:k]
    relevant_count = sum([1 for doc in retrieved_docs if doc in relevant_docs])
    return relevant_count / k
```

Αφού τελειώσαμε με την ανάλυση των συναρτήσεων τώρα θα εξηγήσουμε τον τρόπο εφαρμογής τους. Αρχικά, ορίζουμε 4 μεταβλητές που μας είναι απαραίτητες:

- **queries** : το πλήθος των ερωτημάτων.
- **top_k**: το όριο k που χρειάζεται για την μετρική precision@k.
- **avg_precisions**: λίστα που θα περιέχει τους μέσους όρους των precisions για κάθε ερώτημα για το μοντέλο **Vector Space**.

- **avg_precisions_colbert**: λίστα που θα περιέχει τους μέσους όρους των precisions για κάθε ερώτημα για το μοντέλο **ColBERT**.
- **areas_cosine_similarity**: λίστα που θα περιέχει τα εμβαδά για κάθε ερώτημα για το μοντέλο **Vector Space**.
- **areas_colbert**: λίστα που θα περιέχει τα εμβαδά για κάθε ερώτημα για το μοντέλο **ColBERT**.
- **precisions_at_k_colbert:** λίστα που θα περιέχει τις τιμές του precision@k για κάθε ερώτημα για το μοντέλο **ColBERT.**
- **precisions_at_k_cosine:** λίστα που θα περιέχει τις τιμές του precision@k για κάθε ερώτημα για το μοντέλο **Vector Space**.

```
queries = 20
top_k = 400
avg_precisions = []
avg_precisions_colbert = []
areas_cosine_similarity = []
areas_colbert = []
precisions_at_k_colbert = []
precisions_at_k_cosine = []
```

Στην συνέχεια, διατρέχουμε κάθε γραμμή (ερώτημα) της λίστας **relevant_docs** και για κάθε γραμμή αποθηκεύουμε στη λίστα **relevant_docs_set** τα σχετικά κείμενα που αντιστοιχούν στο συγκεκριμένο **query_id**. Επίσης, με βάση το id των ερωτημάτων βρίσκουμε και αποθηκεύουμε σε λίστες τα ανακτηθέντα κείμενα για το Vector Space Model (**retrieved_docs_list**) και για **ColBERT** (**retrieved_docs_list_colbert**). Για κάθε μοντέλο ξεχωριστά καλούμε την συνάρτηση **calculate_metrics( )** και την τρίτη παράμετρο που μας επιστρέφει, που είναι ο μέσος όρος των precisions, την προσθέτουμε στην λίστα **avg_precisions** ή **avg_precisions_colbert** ανάλογα με το μοντέλο. Επιπλέον, υπολογίζουμε την τιμή του MAP που θα είναι το άθροισμα των παραπάνω λιστών προς το πλήθος των ερωτημάτων. Ύστερα, καλούμε και την συνάρτηση **precision_at_k( )** και για τα δύο μοντέλα και εκτυπώνουμε τα αποτελέσματά της για κάθε ερώτημα. Τις δύο τιμές που μας επιστρέφει η **calculate_metrics( )**, που είναι το recall και το precision, θα τις αποτυπώσουμε σε ένα γράφημα, όπου ο άξονας x θα είναι το recall και ο y το precision. Τέλος, υπολογίζουμε τα εμβαδά για κάθε γραφική καλώντας την συνάρτηση **calculate_area( )**, τα οποία αποθηκεύουμε σε αντίστοιχες λίστες και τα εκτυπώνουμε.

```
for query_id, relevant_docs_set in relevant_docs.items():
    retrieved_docs_list = retrieved_docs.get(query_id, [])
    precisions, recalls, avg_precision = calculate_metrics(relevant_docs_set, retrieved_docs_list)
    avg_precisions.append(avg_precision)
    map_values = sum(avg_precisions) / queries

    # Calculate precision@k for Cosine Similarity
    precision_at_k_cosine = precision_at_k(relevant_docs_set, retrieved_docs_list, top_k)
    precisions_at_k_cosine.append(precision_at_k_cosine)

    # Print or store these values for comparison
    print(f"Precision@{top_k} for Cosine Similarity (Query {query_id + 1}): {precision_at_k_cosine}")

    plt.figure()
    plt.plot(recalls, precisions, marker='o', label = 'Cosine Similarity')
```

```
    plt.title(f"Precision - Recall for Query {query_id + 1}")
    plt.xlabel("Recall")
    plt.ylabel("Precision")
    plt.grid(True)
    plt.legend()
    plt.show()

    # After calculating precisions and recalls for each query
    area_cosine_similarity = calculate_area(recalls, precisions)
    areas_cosine_similarity.append(area_cosine_similarity)

    # Now you can print or plot the AUC valuesD
    print(f"Area for Cosine Similarity (Query {query_id + 1}): {area_cosine_similarity}")


for query_id, relevant_docs_set in relevant_docs.items():
    retrieved_docs_list_colbert = query_docs.get(query_id, [])
    precisions_colbert, recalls_colbert, avg_precision_colbert = calculate_metrics(relevant_docs_set,
retrieved_docs_list_colbert)
    avg_precisions_colbert.append(avg_precision_colbert)
    map_values_colbert = sum(avg_precisions_colbert) / queries

    # Calculate precision@k for ColBERT
    precision_at_k_colbert = precision_at_k(relevant_docs_set, retrieved_docs_list_colbert, top_k)
    precisions_at_k_colbert.append(precision_at_k_colbert)

    # Print or store these values for comparison
    print(f"Precision@{top_k} for ColBERT (Query {query_id + 1}): {precision_at_k_colbert}")

    plt.figure()
    plt.plot(recalls_colbert, precisions_colbert, marker='o', label = 'ColBERT', color = 'orange')
    plt.title(f"Precision - Recall for Query {query_id + 1}")
    plt.xlabel("Recall")
    plt.ylabel("Precision")
    plt.grid(True)
    plt.legend()
    plt.show()

    # After calculating precisions and recalls for each query
    area_colbert = calculate_area(recalls_colbert, precisions_colbert)
    areas_colbert.append(area_colbert)

    # Now you can print or plot the AUC values
    print(f"Area for ColBERT (Query {query_id + 1}): {area_colbert}")
```

Μετά το τέλος του βρόγχου επανάληψης και αφού έχουμε επεξεργαστεί όλα τα ερωτήματα τυπώνουμε το τελικό αποτέλεσμα και της MAP και για τα δύο μοντέλα και στη συνέχεια υπολογίζουμε το μέσο όρο των εμβαδών κάθε μοντέλου και εκτυπώνουμε και αυτό το αποτέλεσμα.

```
print(f"MAP Metric for Cosine Similarity: {map_values}")
print(f"MAP Metric for ColBERT: {map_values_colbert}")

# Calculate the mean area to compare the general efficiency of both models
mean_area_cosine_similarity = sum(areas_cosine_similarity) / queries
mean_area_colbert = sum(areas_colbert) / queries

print(f"Mean Area under Precision-Recall Curve for Cosine Similarity: {mean_area_cosine_similarity}")
print(f"Mean Area under Precision-Recall Curve for ColBERT: {mean_area_colbert}")
```

```
mean_precision_at_k_cosine_similarity = sum(precisions_at_k_cosine)/queries
mean_precision_at_k_colbert = sum(precisions_at_k_colbert)/queries

print(f"Mean Value of Precision@{top_k} for Cosine Similarity: {mean_precision_at_k_cosine_similarity}")
print(f"Mean Value of Precision@{top_k} for ColBERT: {mean_precision_at_k_colbert}")
```

ΑΠΟΤΕΛΕΣΜΑΤΑ – ΠΑΡΑΤΗΡΗΣΕΙΣ

Η συλλογή κειμένων αναφοράς πάνω στην οποία εργαζόμαστε για την εκτίμηση απόδοσης ανάκτησης των δύο μοντέλων είναι η *Cystic Fibosis*. Η συγκεκριμένη συλλογή αποτελείται από 1239 έγγραφα που έχουν δημοσιοποιηθεί στο χρονικό διάστημα 1974 – 1979 σχετικά με το συγκεκριμένο ζήτημα. Εμείς έχουμε λάβει στην υλοποίησή μας ένα υποσύνολο αυτής της συλλογής με 1209 έγγραφα τα οποία επεξεργαζόμαστε. Κάθε ένα από τα κείμενα διαθέτει ένα μοναδικό αναγνωριστικό (SN) το οποίο ξεκινάει από τη τιμή 0001 και αυξάνεται σειριακά μέχρι τη τιμή 1239. Επίσης, η συλλογή περιέχει ένα σύνολο 100 ερωτημάτων, από τα οποία εμείς αξιοποιούμε μόνο τα 20 για να μελετήσουμε την απόδοση ανάκλησης των μοντέλων μας. Για κάθε ερώτημα (QU) διατηρούμε επίσης ένα μοναδικό αναγνωριστικό (QN) με τιμές από 00001 ως 00020, το πλήθος των σχετικών κειμένων που του αντιστοιχούν (NR) καθώς και το βαθμό σχετικότητας κάθε εγγράφου που έχει προκύψει από τους 4 εμπειρογνώμονες (RD). Έχουμε τρεις πιθανές βαθμολογίες:

0: καθόλου σχετικό,
1: μέτρια σχετικό και
2: πολύ σχετικό

3.1 Ερώτημα 1

Η παρακάτω εικόνα αποτελεί ένα τμήμα του **Ανεστραμμένου Ευρετηρίου** που προκύπτει και ακολουθεί την δομή που έχουμε περιγράψει στην υποενότητα **1.1**. Πιο συγκεκριμένα βλέπουμε ότι σε κάθε μοναδικό όρο αντιστοιχεί μια λίστα που περιέχει το ID του κειμένου στο οποίο εντοπίζεται και την συχνότητα εμφάνισής του σε αυτό.

```
LYSOZYME: [('01170', 2), ('01228', 1), ('01229', 2)]
LACTOFERRIN: [('01170', 1), ('01228', 2), ('01229', 2)]
ANTIPROTEOLYTIC: [('01170', 2)]
EXPOSITION: [('01170', 1)]
COMPLEMENTINDUCED: [('01171', 1)]
UNSTIMULATED: [('01172', 2)]
WALLWORK: [('01173', 2)]
8YEAR: [('01174', 1)]
ALBUMEN: [('01174', 1)]
BEARING: [('01175', 1)]
OVERSULPHATED: [('01175', 1)]
COINCIDE: [('01175', 1)]
LOCATIONS: [('01175', 1)]
MULTIORGAN: [('01175', 1)]
ANTONOWICZ: [('01176', 1)]
BMMEC: [('01176', 1)]
```

3.2 Ερώτημα 2

Ο πίνακας term-document που περιέχει τα αποτελέσματα του cosine similarity κάθε κειμένου με κάθε ερώτημα φαίνεται παρακάτω. Προφανώς και οι τιμές του συνημιτόνου ανήκουν στο διάστημα [-1,1], ωστόσο επειδή αναφερόμαστε σε ομοιότητα οι τιμές θα είναι οπωσδήποτε θετικές, άρα θα ανήκουν στο διάστημα [0,1], καθώς σε θεωρητική μελέτη επικεντρωνόμαστε μόνο στο 1ο τεταρτημόριο του ορθοκανονικού επιπέδου.

	0	1	2	...	17	18	19
00001	0.004662	0.086696	0.001493	...	0.001856	0.015787	0.002239
00006	0.016930	0.067133	0.087017	...	0.001077	0.040690	0.001270
00007	0.020077	0.088164	0.078352	...	0.012724	0.027816	0.016087
00008	0.017144	0.043705	0.062975	...	0.013677	0.026875	0.017310
00018	0.000631	0.011961	0.000690	...	0.001308	0.008812	0.001542
...	...	...	...	...	...	...	...
00117	0.000004	0.000013	0.000001	...	0.000899	0.006206	0.001034
00558	0.000001	0.000012	0.000001	...	0.000093	0.005271	0.000001
00651	0.001253	0.007179	0.001372	...	0.001954	0.001422	0.002379
01119	0.001229	0.005155	0.001347	...	0.001582	0.004792	0.001936
00128	0.004926	0.000000	0.005406	...	0.003739	0.005604	0.004748

3.3 Ερώτημα 3

Στο μοντέλο **ColBERT** εκτυπώνεται το κάθε ερώτημα σε φυσική γλώσσα και από κάτω η κατάταξη των κειμένων σε φθίνουσα σειρά με βάση το score ομοιότητας αντιστοιχίζοντας σε κάθε score το ID του κειμένου. Παρακάτω βλέπουμε ενδεικτικά αν τρέξουμε το κώδικά μας τα κορυφαία 32 κείμενα που αντιστοιχούν στο ερώτημα με ID 00016.

```
#> What are the gastrointestinal complications of CF after the neonatal period exclude liver disease and meconium ileus
        [1]             17.4            Document ID: 00301
        [2]             17.4            Document ID: 01119
        [3]             17.3            Document ID: 00991
        [4]             17.3            Document ID: 00596
        [5]             17.1            Document ID: 00659
        [6]             17.0            Document ID: 00788
        [7]             17.0            Document ID: 00798
        [8]             16.7            Document ID: 00649
        [9]             16.7            Document ID: 00909
        [10]            16.5            Document ID: 00333
        [11]            15.5            Document ID: 00875
        [12]            15.4            Document ID: 00186
        [13]            15.2            Document ID: 00473
        [14]            15.2            Document ID: 00322
        [15]            15.1            Document ID: 00801
        [16]            14.9            Document ID: 00772
        [17]            14.8            Document ID: 01226
        [18]            14.5            Document ID: 00396
        [19]            14.4            Document ID: 00690
        [20]            14.4            Document ID: 01002
        [21]            14.4            Document ID: 00908
        [22]            14.2            Document ID: 00796
        [23]            14.1            Document ID: 00550
        [24]            14.1            Document ID: 00036
        [25]            14.0            Document ID: 00096
        [26]            14.0            Document ID: 00661
        [27]            13.8            Document ID: 00870
        [28]            13.7            Document ID: 00775
        [29]            13.5            Document ID: 00895
        [30]            13.2            Document ID: 00175
        [31]            13.2            Document ID: 00518
        [32]            13.2            Document ID: 00900
```

3.4 Ερώτημα 4

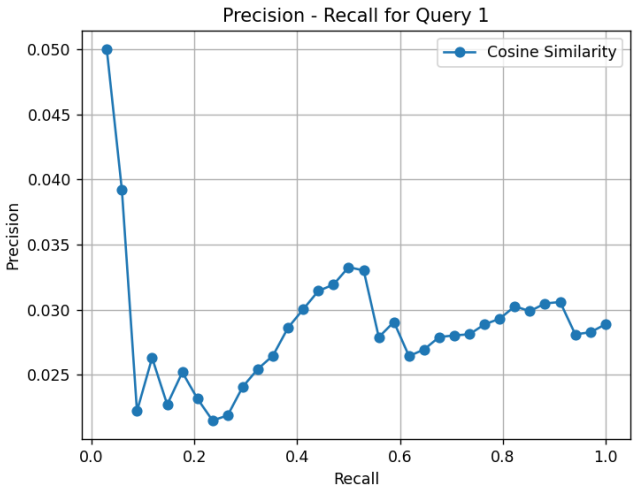
Πρώτου μελετήσουμε την απόδοση ανάκτησης κάθε μοντέλου αναμένουμε ότι το μοντέλο **ColBERT** θα λειτουργεί πολύ πιο αποδοτικά σε σχέση με το **Vector Space Model**. Ο βασικός λόγος είναι ότι το **ColBERT** λόγο της αρχιτεκτονικής που χρησιμοποιεί κατανοεί καλύτερα την σημασιολογία των λέξεων με βάση το περιεχόμενο και τα συμφραζόμενα ενώ το **Vector Space Model** αντιμετωπίζει δυσκολίες όταν προκύπτει Συνωνυμία η Πολυσημία στους όρους των κειμένων. Επίσης, η ακρίβεια του **Vector Space Model** είναι διφορούμενη καθώς σε έγγραφα με μεγαλύτερη έκταση το ότι εντοπίζεται ένας όρος με μεγάλη συχνότητα δεν σημαίνει απαραίτητα ότι το συνολικό περιεχόμενο του κειμένου είναι τελείως σχετικό με την πληροφοριακή ανάγκη του χρήστη. Από τη άλλη το **ColBERT** είναι καλύτερα εκπαιδευμένο στην κατανόηση του συνολικού νοήματος του κειμένου και έτσι στην πραγματικότητα δεν είναι καθόλου απίθανο να σημειώσουμε θεωρητικά βελτίωση στην επίδοση ανάκτησης της τάξης του 10%-30% ή και παραπάνω χρησιμοποιώντας το **ColBERT**.

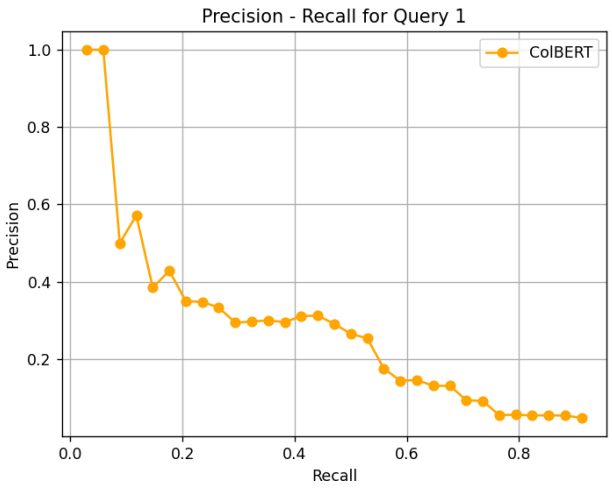
Τώρα βλέπουμε στην πράξη τι απόκλιση έχουν τα δύο μοντέλα όσον αφορά τις επιδόσεις τους συγκρίνοντας τα αποτελέσματα των αντίστοιχων μετρικών που αναλύσαμε στην υποενότητα **1.4.**

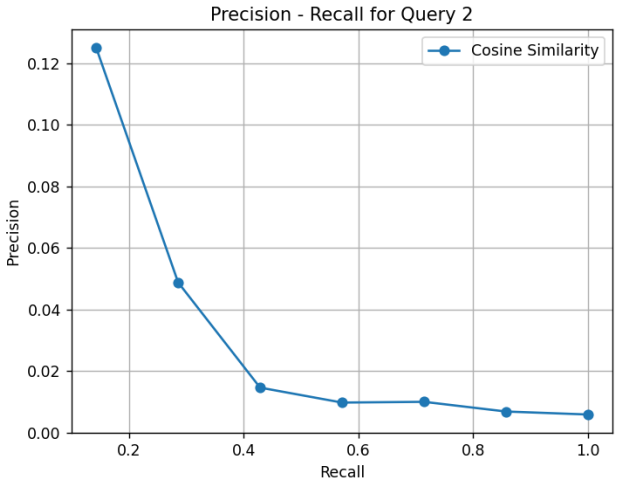
Όσον αφορά την μετρική MAP επιβεβαιώνουμε ότι όντως το **ColBERT** λειτουργεί αποδοτικότερα, καθώς η μέση τιμή της μέσης ακρίβειάς του είναι περίπου 6 φορές μεγαλύτερη σε σχέση με την μέση τιμή της μέσης ακρίβειας που προκύπτει από το **Vector Space Model**.

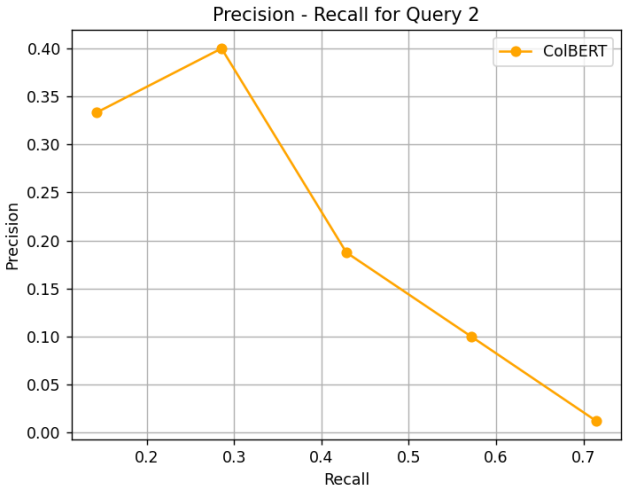
```
MAP Metric for Cosine Similarity: 0.040536234732691065
MAP Metric for ColBERT: 0.23716929659274122
```

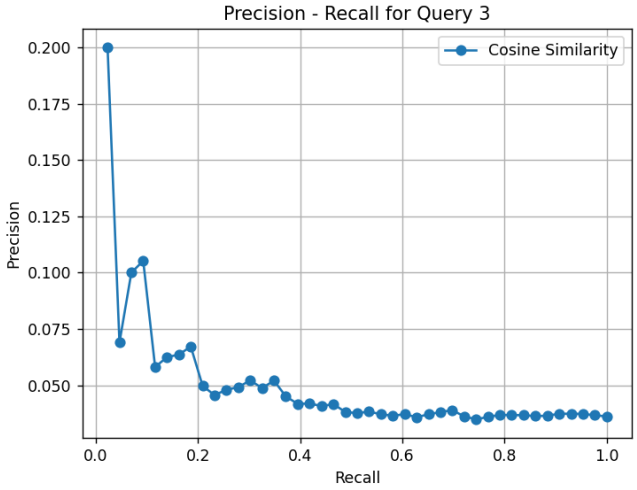
Έπειτα, παραθέτουμε ενδεικτικά τις 9 πρώτες γραφικές ακρίβειας - ανάκλησης για το **Vector Space Model** και το **ColBERT**:

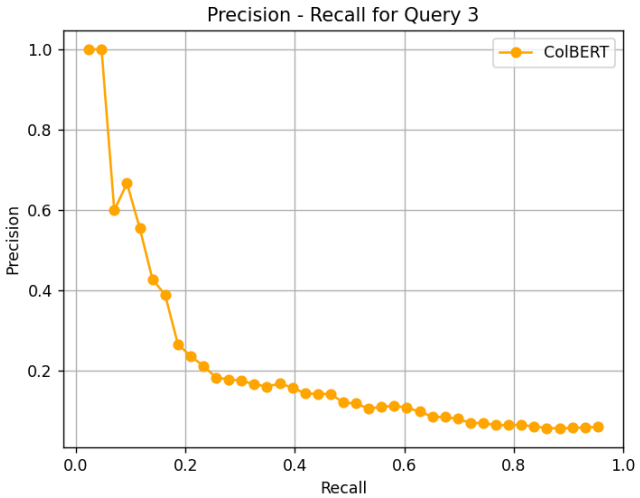


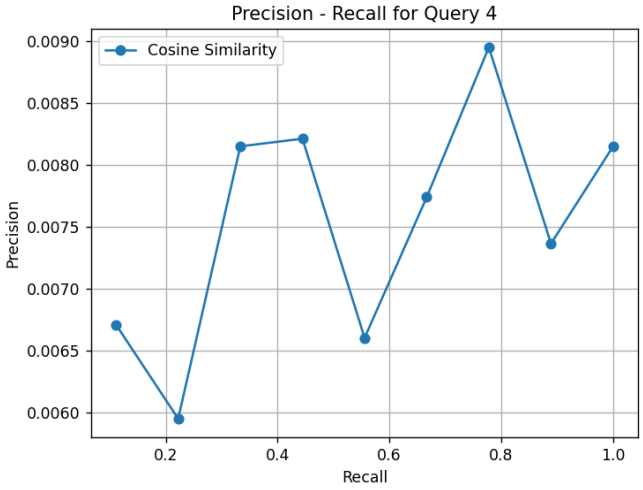


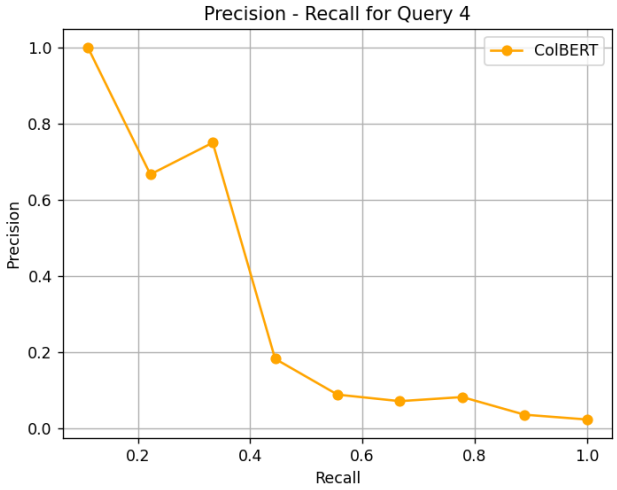


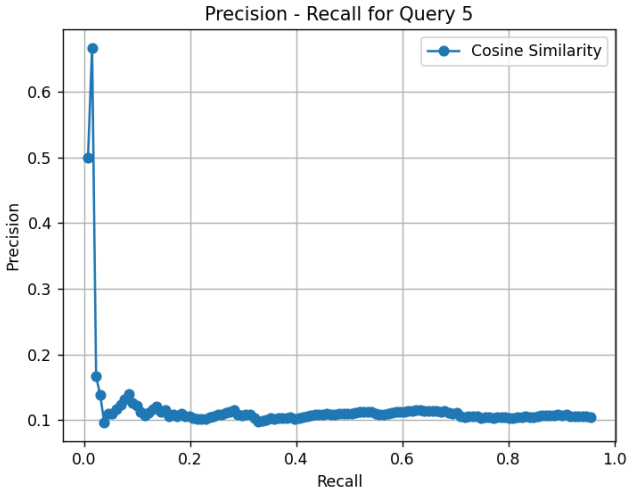


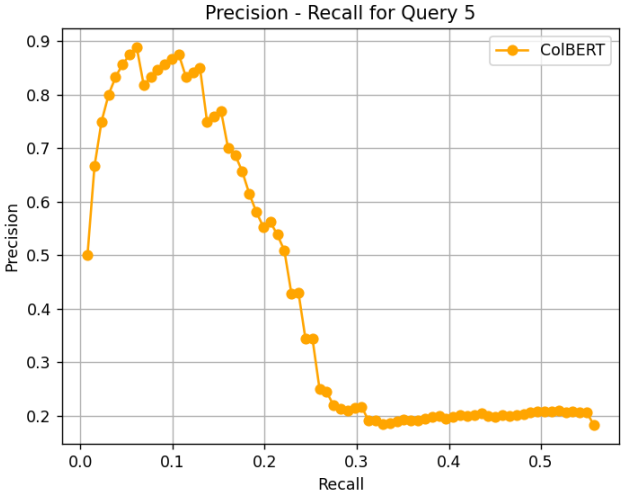


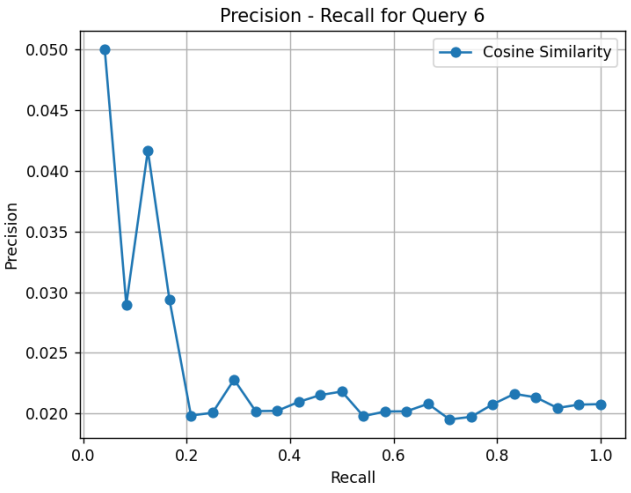


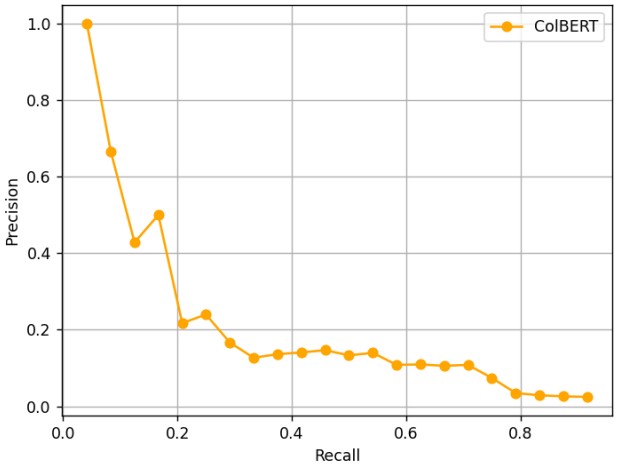

Precision - Recall for Query 4
Cosine Similarity
Precision
Recall
0.0090
0.0085
0.0080
0.0075
0.0070
0.0065
0.0060
0.2
0.4
0.6
0.8
1.0

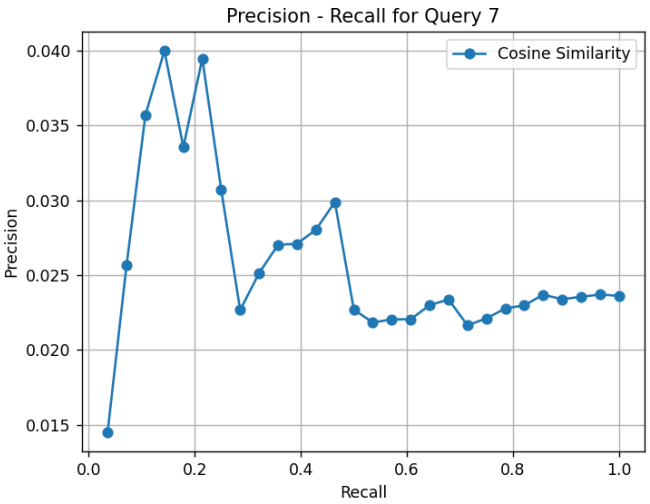

Precision - Recall for Query 4
ColBERT
Precision
Recall
1.0
0.8
0.6
0.4
0.2
0.0
0.2
0.4
0.6
0.8
1.0

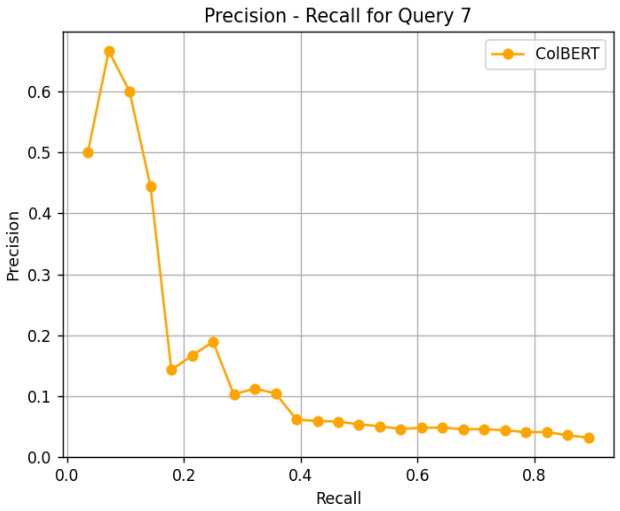

Precision - Recall for Query 5
Cosine Similarity
Precision
Recall
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0

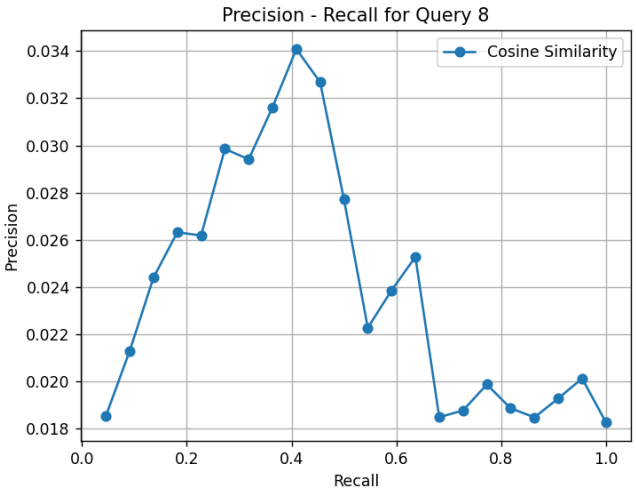

Precision - Recall for Query 5
ColBERT
Precision
Recall
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.0
0.1
0.2
0.3
0.4
0.5

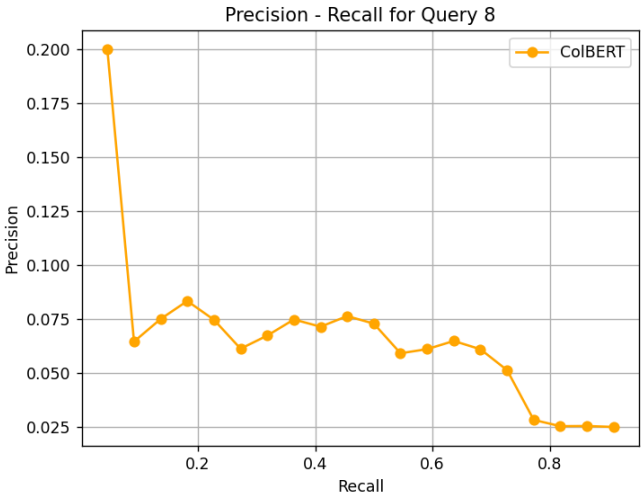

Precision - Recall for Query 6
Cosine Similarity
Precision
Recall
0.050
0.045
0.040
0.035
0.030
0.025
0.020
0.0
0.2
0.4
0.6
0.8
1.0

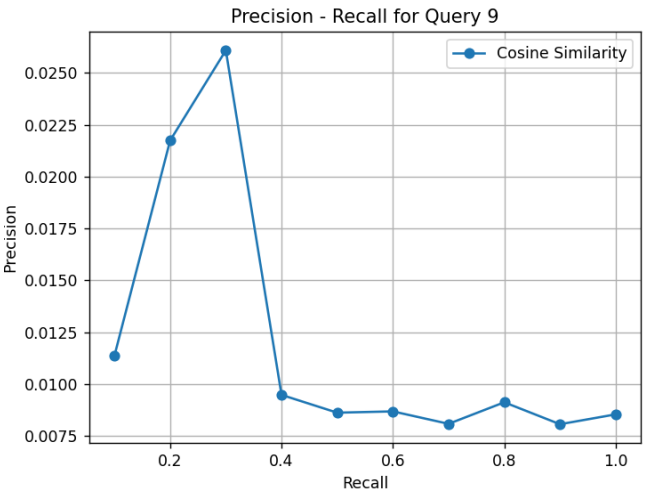

Precision - Recall for Query 6
ColBERT
Precision
Recall
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8

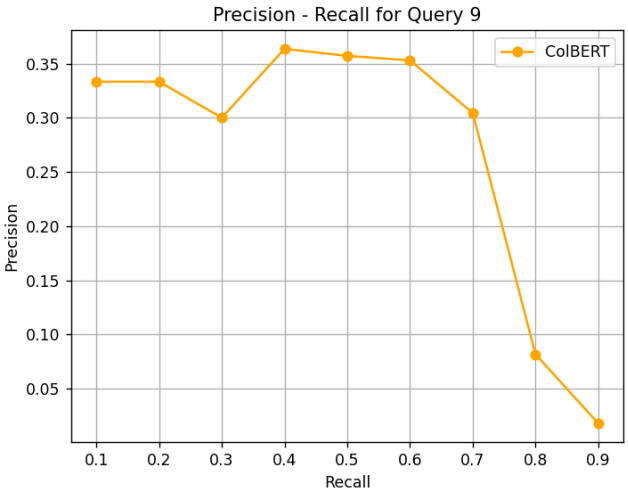

Precision - Recall for Query 7
Cosine Similarity
Precision
Recall
0.040
0.035
0.030
0.025
0.020
0.015
0.0
0.2
0.4
0.6
0.8
1.0


Precision - Recall for Query 7
ColBERT
Precision
Recall
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.2
0.4
0.6
0.8


Precision - Recall for Query 8
Cosine Similarity
Precision
Recall
0.034
0.032
0.030
0.028
0.026
0.024
0.022
0.020
0.018
0.0
0.2
0.4
0.6
0.8
1.0


Precision - Recall for Query 8
ColBERT
Precision
Recall
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.2
0.4
0.6
0.8


Precision - Recall for Query 9
Cosine Similarity
Precision
Recall
0.0250
0.0225
0.0200
0.0175
0.0150
0.0125
0.0100
0.0075
0.2
0.4
0.6
0.8
1.0


Precision - Recall for Query 9
ColBERT
Precision
Recall
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

Παρατηρούμε ότι σε όλες τις γραφικές παραστάσεις η ακρίβεια σε σχέση με την ανάκληση είναι πολύ μεγαλύτερη στο **ColBERT** γεγονός το οποίο είναι αναμενόμενο για τους λόγους που αναλύσαμε και παραπάνω. Ωστόσο, για να έχουμε μια γενική ιδέα για το ποσοστό απόκλισης των δύο μοντέλων δεν θα συγκρίνουμε μεμονωμένα τις γραφικές για κάθε ερώτημα σημείο προς σημείο ή τα εμβαδά τους για το καθένα ξεχωριστά, αλλά θα συγκρίνουμε τον μέσο όρο των εμβαδών ο οποίος θα μας δώσει μια αντιπροσωπευτική τιμή για όλα τα εμβαδά. Πιο συγκεκριμένα διαπιστώνουμε ότι σε αυτήν την περίπτωση το **ColBERT** έχει ακρίβεια ανάκλησης πάλι σχεδόν 6 φορές μεγαλύτερη σε σχέση με το **Vector Space**. Άρα βλέπουμε ότι και η συγκεκριμένη μετρική ευνοεί σωστά περισσότερο το **ColBERT**.

```
Mean Area under Precision-Recall Curve for Cosine Similarity: 0.03886720959992993
Mean Area under Precision-Recall Curve for ColBERT: 0.21924042025673032
```

Τέλος, σχετικά με την τελευταία μετρική που επιλέξαμε, πάλι θα μελετήσουμε την μέση τιμή όλων των τιμών της precison@k που προκύπτει από όλα τα ερωτήματα. Για τιμή του k επιλέγουμε αρχικά το 400, καθώς δεν θα είναι αντικειμενικό μέτρο σύγκρισης η τιμή που θα προκύψει για μικρές τιμές του, αφού το **Vector Space Model** κατά κύρια βάση αργεί να επιστρέψει τα πολύ σχετικά κείμενα λόγω της μικρότερης ακρίβειάς του. Τελικά, πάλι παρατηρούμε ότι η μέση τιμή της μετρικής για το **ColBERT** είναι σχεδόν διπλάσια αυτή τη φορά σε σχέση με το **Vector Space Model** και συνεπώς επιβεβαιώνεται για ακόμη μια φορά η καλύτερη αποδοτικότητα του **ColBERT**. Για μεγαλύτερες τιμές του k αναμένουμε οι δύο μετρικές να συγκλίνουν ακόμα περισσότερο, καθώς όσο προχωράμε στην ανάκληση επιστρέφονται και από τα δύο μοντέλα όλο και περισσότερα σχετικά έγγραφα. Αντίθετα, αν θέλαμε μια ακόμα πιο αντιπροσωπευτική τιμή, θα μπορούσαμε να μικρύνουμε τη τιμή του k, καθώς και στη πράξη μας ενδιαφέρει η ακρίβεια στα πρώτα αποτελέσματα που επιστρέφονται, εφόσον ο χρήστης θα χάσει το ενδιαφέρον του όσο συνεχίζει να ψάχνει στη λίστα με τα επιστρεφόμενα κείμενα.

```
Mean Value of Precision@400 for Cosine Similarity: 0.03725
Mean Value of Precision@400 for ColBERT: 0.071125
```

Άρα γενικά όλες οι μετρικές υποδεικνύουν ότι το **ColBERT** είναι προτιμότερο σε σχέση με το **Vector Space Model**, παρ' όλο που και το τελευταίο μας παρέχει σε γενικές γραμμές σχετικά καλά αποτελέσματα. Συνεπώς, οι μετρικές επιβεβαιώνουν και στην πράξη την υπεροχή του πρώτου μοντέλου έναντι του δεύτερου, όπως ακριβώς αναφέρουμε και στη θεωρητική μελέτη των δύο μοντέλων στην ενότητα **1**.

ΑΝΑΦΟΡΕΣ

4.1 Αναφορές

Πιο κάτω παρατίθενται όλες οι πηγές, σύνδεσμοι και βιβλιογραφία που μελετήσαμε κατά την υλοποίηση της συγκεκριμένης εργασίας:

[1] Khattab, Omar, and Matei Zaharia. "Colbert: Efficient and effective passage search via contextualized late interaction over bert." *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 2020.

[2] Tsatsaronis, George, and Vicky Panagiotopoulou. "A generalized vector space model for text retrieval based on semantic relatedness." *Proceedings of the Student Research Workshop at EACL 2009*. 2009.

[3] Singh, Vaibhav Kant, and Vinay Kumar Singh. "Vector space model: an information retrieval system." *Int. J. Adv. Engg. Res. Studies/IV/II/Jan.-March* 141.143 (2015).

[4] Korra, R. "Performance evaluation of Multilingual Information Retrieval (MLIR) system over Information Retrieval (IR)." *Proceedings of the International Conference system Recent Trends in Information Technology (ICRTIT).–2017*. 2017.

[5] Patil, Manish, et al. "Inverted indexes for phrases and strings." *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. 2011.

[6] Ricardo Baeza-Yates, Berthier Ribeiro-Neto "Ανάκτηση Πληροφορίας οι έννοιες και η τεχνολογία πίσω από την αναζήτηση", Εκδόσεις Τζιόλα, 2η έκδοση, 2021.

[7] *https://github.com/stanford-futuredata/ColBERT?tab=readme-ov-file*

[8] *https://colab.research.google.com/github/stanford-futuredata/ColBERT/blob/main/docs/intro2new.ipynb*

ΠΑΡΑΡΤΗΜΑ

5.1 Κώδικας – Σχόλια

Κώδικας Python σε *PyCharm* Ερωτημάτων 1, 2, 4:

```
import os
from collections import defaultdict
import math
import numpy as np
import pandas as pd
import ast
import re
from matplotlib import pyplot as plt
import csv

# Question 1

# defaultdict to store the inverted index
inverted_index = defaultdict(list)

# Dictionary to store the  documents each word appears in
document_count = {}

path = (r"C:\Users\chryssa_pat\PycharmProjects\pythonProject\docs")
os.chdir(path)


for file in os.listdir(path):
    file_path = os.path.join(path, file)
    with open(file_path, 'r') as folder:
        text = folder.read()

        dictionary = text.split()

        # Create a dictionary to store word counts for each document
        count = {}

        for word in dictionary:
            count[word] = count.get(word, 0) + 1

        # Update the inverted index with word counts for the current document
        for word, count in count.items():
            inverted_index[word].append((file, count))

            # Update the document count for the current word
            if word in document_count:
                document_count[word].add(file)
            else:
                document_count[word] = {file}
inverted_index_csv = r"C:\Users\chryssa_pat\PycharmProjects\pythonProject\inverted_index.csv"

# Print the inverted index
for word, documents in inverted_index.items():
    print(f"{word}: {documents}")

# Save the inverted index into a CSV file
```

```
with open(inverted_index_csv, 'w', newline='', encoding='utf-8') as csvfile:
    csv_writer = csv.writer(csvfile)
    csv_writer.writerow(['Word', 'Documents'])

    for word, documents in inverted_index.items():
        csv_writer.writerow([word, documents])

# Question 2
# Calculate IDF for each word and store it in a dictionary
idf = {}
for word, documents in document_count.items():
    idf_value = math.log10((len([f for f in os.listdir(path)])) / len(documents))
    idf[word] = idf_value

# Create a DataFrame with IDF values
idf = pd.DataFrame.from_dict(idf, orient='index', columns=['IDF'])

csv_file_path = r"C:\Users\chryssa_pat\PycharmProjects\pythonProject\idf.csv"
# Save the DataFrame to a CSV file
idf.to_csv(csv_file_path, index=False)

# Create a dictionary to store TF values for each word in each document
tf = {}
for word, occurrences in inverted_index.items():
    for document, count in occurrences:
        if document in tf:
            tf[document][word] = 1 + math.log10(count)
        else:
            tf[document] = {word: 1 + math.log10(count)}

# Create a DataFrame where rows are document titles and columns are words with their TF values
tf = pd.DataFrame.from_dict(tf, orient='index')
tf.fillna(0, inplace=True)  # Fill missing values with 0

csv_file_path = r"C:\Users\chryssa_pat\PycharmProjects\pythonProject\doc_tf.csv"
# Save the DataFrame to a CSV file
tf.to_csv(csv_file_path, index=False)

# Iterate through the TF and IDF DataFrames to calculate TF-IDF
tfidf = tf.copy()
# Multiply each TF value by the corresponding IDF value
for column in tfidf.columns:
    tfidf[column] = tfidf[column] * idf.loc[column, 'IDF']

csv_file_path = r"C:\Users\chryssa_pat\PycharmProjects\pythonProject\doc_weights.csv"
# Save the DataFrame to a CSV file
tfidf.to_csv(csv_file_path, index=False)

query_path = r"C:\Users\chryssa_pat\PycharmProjects\pythonProject\Queries_20"
# Read questions from the file
with open(query_path, 'r') as file:
    questions = file.readlines()

# Create a list of words from the inverted index
all_words = list(inverted_index.keys())

# Create a DataFrame with zeros
df = pd.DataFrame(0, index=range(len(questions)), columns=all_words)

# Iterate through questions and update the DataFrame
```

```
count = 0
for i, question in enumerate(questions):
    words = question.upper().split()
    # Find common words between the question and the inverted index
    common_words = set(words) & set(inverted_index)
    word_counts = {word: 0 for word in common_words}

    for word in common_words:
        # Count the number of occurrences of the word in the question
        pattern = r'\b' + re.escape(word) + r'\b'
        count = len(re.findall(pattern, question.upper()))
        # Update the DataFrame with the log-transformed count
        if count > 0:
            df.at[i, word] = 1 + math.log10(count)

df = df.fillna(0)
csv_file_path = r"C:\Users\chryssa_pat\PycharmProjects\pythonProject\query_tf.csv"
# Save the DataFrame to a CSV file
df.to_csv(csv_file_path, index=False)

query_tfidf = df.multiply(idf['IDF'], axis=1)
csv_file_path = r"C:\Users\chryssa_pat\PycharmProjects\pythonProject\query_weights.csv"
# Save the DataFrame to a CSV file
query_tfidf.to_csv(csv_file_path, index=False)

# Calculate the Euclidean norm for each row of doc_weights
euclidean_norms = np.linalg.norm(tfidf.values, axis=1)
# Create a DataFrame with the Euclidean norm values
euclidean_norms_d = pd.DataFrame({'Euclidean Norm': euclidean_norms}, index=tfidf.index)

# Calculate the Euclidean norm for each row of query_weights
euclidean_norms = np.linalg.norm(query_tfidf.values, axis=1)
# Create a DataFrame with the Euclidean norm values
euclidean_norms_q = pd.DataFrame({'Euclidean Norm': euclidean_norms}, index=query_tfidf.index)

# Calculate the dot product
dot_products = np.dot(tfidf.values, query_tfidf.values.T)
# Create a DataFrame with the dot product values
dot_product = pd.DataFrame(dot_products, index=tfidf.index, columns=query_tfidf.index)

# Calculate the product of the Euclidean norms
norm_product = euclidean_norms_d.values @ euclidean_norms_q.values.T
# Create a DataFrame with the result
norm_product = pd.DataFrame(norm_product, index=euclidean_norms_d.index,
columns=euclidean_norms_q.index)

# Calculate the cosine similarity
cosine_similarity = dot_product.div(norm_product)
print(cosine_similarity)

# Question 4
# Precision - Recall and MAP Metrics
#cosine similarity
# Sort each column and replace values with doc IDs
sorted_doc_ids = cosine_similarity.apply(lambda col: col.sort_values(ascending=False).index)
csv_file_path = r"C:\Users\chryssa_pat\PycharmProjects\pythonProject\cosine_similarity.csv"
# Save the DataFrame to a CSV file
sorted_doc_ids.to_csv(csv_file_path, index=False)
# Convert document IDs to integers and get the retrieved docs
retrieved_docs = {query_id: [int(doc_id) for doc_id in doc_ids] for query_id, doc_ids in
```

```
sorted_doc_ids.items()}

#colbert
colbert_path = r"C:\Users\chryssa_pat\PycharmProjects\pythonProject\colbert_result.csv"
query_docs = {}
# Read the ColBERT results into a DataFrame
with open(colbert_path, 'r') as csvfile:
    csvreader = csv.reader(csvfile)
    next(csvreader)  # Skip the header row

    for query_id, row in enumerate(csvreader):
        # Parse the string representation of the list into an actual list
        doc_ids = ast.literal_eval(row[1])

        # Convert each element in the list to an integer
        doc_ids = [int(id) for id in doc_ids]

        # Ensure doc_ids is a list of integers
        if isinstance(doc_ids, list) and all(isinstance(id, int) for id in doc_ids):
            query_docs[query_id] = doc_ids
# Get the relevant docs
file_path = r"C:\Users\chryssa_pat\PycharmProjects\pythonProject\Relevant_20"
relevant_docs = {}
with open(file_path, 'r') as file:
    for query_id, line in enumerate(file):
        # Convert all space-separated numbers in the line to integers and store them as a set
        doc_ids = set(map(int, line.split()))
        # Assign this set to the corresponding query ID
        relevant_docs[query_id] = doc_ids

def calculate_metrics(relevant_docs, retrieved_docs):
    ret_count = 0
    rel_count = 0
    precisions = []
    recalls = []

    # Iterate through each relevant document
    for doc in retrieved_docs:
        ret_count += 1
        # Check if the relevant document is in the retrieved documents
        if doc in relevant_docs:
            rel_count += 1
            precision = rel_count / ret_count
            precisions.append(precision)
            recall = rel_count / len(relevant_docs)
            recalls.append(recall)
    avg_precision = sum(precisions) / len(relevant_docs)

    return precisions, recalls, avg_precision

def calculate_area(recalls, precisions):
    area = 0.0
    for i in range(1, len(recalls)):
        # Calculate the area of the trapezoid
        width = recalls[i] - recalls[i-1]
        height = (precisions[i] + precisions[i-1]) / 2
        area += width * height
    return area
```

```
def precision_at_k(relevant_docs, retrieved_docs, k):
    if len(retrieved_docs) > k:
        retrieved_docs = retrieved_docs[:k]
    relevant_count = sum([1 for doc in retrieved_docs if doc in relevant_docs])
    return relevant_count / k


queries = 20
top_k = 400
avg_precisions = []
avg_precisions_colbert = []
areas_cosine_similarity = []
areas_colbert = []
precisions_at_k_colbert = []
precisions_at_k_cosine = []

for query_id, relevant_docs_set in relevant_docs.items():
    retrieved_docs_list = retrieved_docs.get(query_id, [])
    precisions, recalls, avg_precision = calculate_metrics(relevant_docs_set, retrieved_docs_list)
    avg_precisions.append(avg_precision)
    map_values = sum(avg_precisions) / queries


    # Calculate precision@k for Cosine Similarity
    precision_at_k_cosine = precision_at_k(relevant_docs_set, retrieved_docs_list, top_k)
    precisions_at_k_cosine.append(precision_at_k_cosine)

    # Print or store these values for comparison
    print(f"Precision@{top_k} for Cosine Similarity (Query {query_id + 1}): {precision_at_k_cosine}")

    plt.figure()
    plt.plot(recalls, precisions, marker='o', label = 'Cosine Similarity')
    plt.title(f"Precision - Recall for Query {query_id + 1}")
    plt.xlabel("Recall")
    plt.ylabel("Precision")
    plt.grid(True)
    plt.legend()
    plt.show()

    # After calculating precisions and recalls for each query
    area_cosine_similarity = calculate_area(recalls, precisions)
    areas_cosine_similarity.append(area_cosine_similarity)

    # Now you can print or plot the AUC valuesD
    print(f"Area for Cosine Similarity (Query {query_id + 1}): {area_cosine_similarity}")

for query_id, relevant_docs_set in relevant_docs.items():
    retrieved_docs_list_colbert = query_docs.get(query_id, [])
    precisions_colbert, recalls_colbert, avg_precision_colbert = calculate_metrics(relevant_docs_set,
retrieved_docs_list_colbert)
    avg_precisions_colbert.append(avg_precision_colbert)
    map_values_colbert = sum(avg_precisions_colbert) / queries

    # Calculate precision@k for ColBERT
    precision_at_k_colbert = precision_at_k(relevant_docs_set, retrieved_docs_list_colbert, top_k)
    precisions_at_k_colbert.append(precision_at_k_colbert)

    # Print or store these values for comparison
    print(f"Precision@{top_k} for ColBERT (Query {query_id + 1}): {precision_at_k_colbert}")
```

```
    plt.figure()
    plt.plot(recalls_colbert, precisions_colbert, marker='o', label = 'ColBERT', color = 'orange')
    plt.title(f"Precision - Recall for Query {query_id + 1}")
    plt.xlabel("Recall")
    plt.ylabel("Precision")
    plt.grid(True)
    plt.legend()
    plt.show()

    # After calculating precisions and recalls for each query
    area_colbert = calculate_area(recalls_colbert, precisions_colbert)
    areas_colbert.append(area_colbert)

    # Now you can print or plot the AUC values
    print(f"Area for ColBERT (Query {query_id + 1}): {area_colbert}")


print(f"MAP Metric for Cosine Similarity: {map_values}")
print(f"MAP Metric for ColBERT: {map_values_colbert}")

# Calculate the mean area to compare the general efficiency of both models
mean_area_cosine_similarity = sum(areas_cosine_similarity) / queries
mean_area_colbert = sum(areas_colbert) / queries

print(f"Mean Area under Precision-Recall Curve for Cosine Similarity: {mean_area_cosine_similarity}")
print(f"Mean Area under Precision-Recall Curve for ColBERT: {mean_area_colbert}")

mean_precision_at_k_cosine_similarity = sum(precisions_at_k_cosine)/queries
mean_precision_at_k_colbert = sum(precisions_at_k_colbert)/queries

print(f"Mean Value of Precision@{top_k} for Cosine Similarity: {mean_precision_at_k_cosine_similarity}")
print(f"Mean Value of Precision@{top_k} for ColBERT: {mean_precision_at_k_colbert}")
```

Κώδικας Python σε *Google Colab* Ερωτήματος 3:

```
!apt install python3.10-venv
!pip install virtualenv
!virtualenv myenv
!source myenv/bin/activate
import torch
print(torch.cuda.is_available())

!pip install git+https://github.com/stanford-futuredata/ColBERT.git
!git -C ColBERT/ pull || git clone https://github.com/stanford-futuredata/ColBERT.git
import sys; sys.path.insert(0, 'ColBERT/')

import google.colab
!pip install -U pip
!pip install -e ColBERT/['faiss-gpu','torch']

# Import necessary components from colbert library
import colbert
from colbert.modeling.colbert import ColBERT
from colbert import Indexer, Searcher
from colbert.infra import Run, RunConfig, ColBERTConfig
```

```
from colbert.data import Queries, Collection

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

    print("ColBERT is imported successfully!")
except ImportError as e:
    print(f"Error importing ColBERT: {e}")

from google.colab import drive
drive.mount('/content/drive')

drive.mount("/content/drive", force_remount=True)

!unzip /content/drive/MyDrive/ir/docs.zip -d /content

import os
import csv

def read_documents(directory_path):
    documents = []
    for file in os.listdir(directory_path):
        file_path = os.path.join(directory_path, file)
        with open(file_path, 'r', encoding="utf8") as folder:
            text = folder.read()
            doc_id = file
            document_tuple = (doc_id, text)
            documents.append(document_tuple)
    return documents

# Set the path to your documents and queries

path = '/content/docs'
queries_path = '/content/Queries_20'

# Read documents
documents = read_documents(path)

def read_queries(file_path):
    queries = []
    with open(file_path, 'r', encoding="utf8") as file:
        text = file.read()
        queries_list = text.split('\n')
        queries_list = [query.strip() for query in queries_list if query.strip()]
        queries.extend(queries_list)
    return queries
# Read queries
queries = read_queries(queries_path)
```

```
# Set up ColBERT configurations
nbits = 2
doc_maxlen = 300
max_id = 1209
index_name = 'index'
checkpoint = '/content/drive/MyDrive/ir/colbertv2.0'

# Indexing
with Run().context(RunConfig(nranks=1, experiment='indexing')):
    config = ColBERTConfig(doc_maxlen=doc_maxlen, nbits=nbits, kmeans_niters=4)
    indexer = Indexer(checkpoint=checkpoint, config=config)
    indexer.index(name=index_name, collection=[doc[1] for doc in documents[:max_id]], overwrite=True)

# Searching
with Run().context(RunConfig(experiment='indexing')):
    searcher = Searcher(index=index_name, collection=[doc[1] for doc in documents])


# Create a dictionary to store lists of document IDs for each query
colbert_result = {}


# Iterate over all queries
for query in queries:
    print(f"#> {query}")
    results = searcher.search(query, k=1209)  # Retrieve all passages
    result_list = []
    for i, (passage_id, passage_rank, passage_score) in enumerate(zip(*results)):
        doc_id = documents[int(passage_id)][0]
        print(f"\t [{passage_rank}] \t\t {passage_score:.1f} \t\t  Document ID: {doc_id}")
        result_list.append(doc_id)
    colbert_result[query] = result_list

# Display or save the dictionary
print(colbert_result)

# Save the dictionary to a CSV file
csv_file_path = '/content/colbert_result.csv'
with open(csv_file_path, 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)

    # Write the header
    writer.writerow(['Query', 'DocumentIDs'])

    # Write the data
    for query, document_list in colbert_result.items():
        writer.writerow([query, document_list])

print(f"Query document lists saved to {csv_file_path}")
```