2024/2025 - Information Retrieval Project

Δημήτρης Κολιγλιάτης
AM:1097458

Γιώργος Αγγελόπουλος
AM:1067435

Περιεχόμενα

Στοιχεία φοιτητών:

Ονοματεπώνυμο: Κολιγλιάτης Δημήτρης
AM:1097458
Έτος: Τέταρτο(4ο)
Email:up1097458@ac.upatras.gr

Ονοματεπώνυμο: Αγγελόπουλος Γιώργος
AM:1067435
Έτος: Έβδομο(7ο)
Email:up1067435@ac.upatras.gr

Εργαλεία:
Η αναφορά γράφτηκε με την χρήση του εργαλείου overleaf σε LaTEX καθώς και ο κώδικας σε python. στο περιβάλλον VScode.

Ερώτημα 1

Για το ερώτημα 1, που απαιτούσε την ανάγνωση και την προεπεξεργασία της συλλογής, έγινε χρήση βασικών βιβλιοθηκών της python με κύριες την pandas, που είναι κατάλληλη για data analysis, καθώς και την nltk με κύριο σκοπό την αφαίρεση stopwords, καθώς και την μετατροπή της συλλογής κειμένων, σε επεξεργάσιμη δομή.

Για τη δημιουργία του ευρετηρίου για το **Boolean Model** χρησιμοποιήθηκε η μέθοδος του ανεστραμμένου ευρετηρίου (Inverted Index). Κάθε όρος που περιέχεται στα έγγραφα της συλλογής αποθηκεύεται ως καταχώρηση (entry) στο ευρετήριο, με τα αντίστοιχα έγγραφα στα οποία εμφανίζεται. Η μορφή του αρχείου είναι η εξής:

• Όρος : [Λίστα εγγράφων]

Για παράδειγμα:
restricted : [141, 295, 431, 813, 1128]
blue : [74, 83, 462, 520, 609, 712, 895, 909, 1121, 1125]

Η δομή αυτή επιλέχθηκε για τους εξής λόγους:

• **Απόδοση**: Η χρήση ανεστραμμένου ευρετηρίου εξασφαλίζει γρήγορη ανάκτηση εγγράφων μέσω βοολεαν queries.

• **Απλότητα**: Η μορφή όρος-λίστα είναι εύκολα επεξεργάσιμη.

• **Επεκτασιμότητα**: Η αποθήκευση των δεδομένων σε μορφή κειμένου επιτρέπει την επέκταση της συλλογής χωρίς ιδιαίτερη επιβάρυνση.

Για το VSM επιλέχθηκε μία πιο λεπτομερής εκδοχή του ανεστραμμένου ευρετηρίου όπως είχαμε και στο Boolean model. Σε αυτήν την εκδοχή, το ευρετήριο απαρτίζεται από όρους και συνοδεύεται από μία λίστα όπου κάθε στοιχείο της είναι το id του κειμένου που εμφανίζεται η λέξη και το Term Frequency βάρος της όπως υπολογίζεται από τον τύπο που περιγράφεται παρακάτω. Μαζί με την λίστα αυτή, αποθηκεύεται και το IDF της λέξης μέσα στην συλλογή.
Άρα τελικώς το ευρετήριο για κάθε όρο έχει αυτήν την μορφή:

• **Term1**: [[Doc_id_1, tf_1] , [Doc_id_2, tf_2], . . . , [Doc_id_i, tf_i]] , IDF_Term1

• **Term2**: [[Doc_id_1, tf_1] , [Doc_id_2, tf_2], . . . , [Doc_id_i, tf_i]] , IDF_Term2

Ο λόγος που επιλέχθηκε αυτή η μορφή είναι:
• Για την **διευκόλυνση** αργότερα και την σωστή λειτουργία του Vector Space μοντέλου καθώς θα χρειαστούν και τα επί μέρους Term Frequencies αλλά και το IDF της λέξης.

• **Απόδοση**: Ακόμα και με την πιο εκτενή μορφή του, το ανεστραμμένο ευρετήριο αποτελεί μία από τις πιο αποδοτικές και γρήγορες δομές για ψάξιμο. Η απόδοση του Vector Space Model βασίζεται πάρα πολύ σε αυτήν την δομή και από άποψη ποιότητας αποτελεσμάτων αλλά και χρονικής απόδοσης.

Για την προεπεξεργασία της συλλογής εφαρμόστηκαν οι εξείς τεχνικές:

• **Αφαίρεση Stop Words**: Οι κοινές λέξεις (π.χ. "the", "and") αφαιρέθηκαν χρησιμοποιώντας τη βιβλιοθήκη nltk.

• **Αφαίρεση Πλεονασμών**: Πλεονάζουσες λέξεις αφαιρέθηκαν ώστε να μειωθεί το μέγεθος του ευρετηρίου.

• **Κανονικοποίηση Κειμένου**: Όλα τα κείμενα μετατράπηκαν σε πεζά για την εξάλειψη διαφορών λόγω κεφαλαίων χαρακτήρων.

Η αφαίρεση stop words μειώνει τον θόρυβο στα queries και βελτιώνει την ακρίβεια. Η κανονικοποίηση αποτρέπει περιττές ασυμφωνίες λόγω διαφορετικών μορφών γραφής.

Σε όλη την υλοποίηση της προεπεξεργασίας και της ευρετηριοποίησης αξιοποιήθηκε κατα κόρον η βιβλιοθήκη Pandas της Python που προσφέρει πολλές χρήσιμες μεθόδους για γρήγορη και αποτελεσματική αποθήκευση και επεξεργασία δεδομένων σε πίνακες. Αυτό μας επέτρεψε σε κάθε πράξη που περιέχει τα κείμενα και στοιχεία τους, αυτά να μπορούν να συνοδεύονται με το αντίστοιχο id που έχουν μέσα στον φάκελο και όχι απλά με την απόλυτη θέση του κειμένου στην σειρά κειμένων μέσα στον φάκελο.

Πιο συγκεκριμένα, αν δεν μεριμνούσαμε να αποθηκεύσουμε και να κουβαλήσουμε το id του κάθε κειμένου, όταν πχ θα φορτώναμε τα κείμενα σε μία λίστα από strings, αυτά θα ήταν διατεταγμένα με την σειρά τους όπως και στον φάκελο αλλά η αναγνώρισή τους θα γινόταν με την σχετική τους θέση στην λίστα. Επειδή λείπουν όμως κείμενα από την συλλογή, υπάρχουν κενά στην αντιστοίχιση σειράς εμφάνισης των θέσεων στην λίστα και πραγματικού id (πχ το κείμενο στην θέση 99 -με αφετηρία το 0- στην λίστα έχει πραγματικά id = 140 στην συλλογή) Άρα το πρόβλημα των ελλιπών κειμένων το προσεγγίσαμε με συνοδευτική αποθήκευση εκτός του περιεχομένου κάθε κειμένου με την σειρά και το id του κάθε κειμένου ώστε να αναγνωρίζεται όπως θα έπρεπε στην τελική σύγκριση για τις μετρικές.

Ερώτημα 2

Τρόπος λειτουργίας Boolean Model

Αρχικά, έγινε η προεπεξεργασία των ερωτημάτων καθώς και της συλλογής των κειμένων που μας δόθηκε όπως εξηγείται στην ενότητα "Ερώτημα 1".
Έπειτα, κάθε αποθηκευμένος όρος που προ-επεξεργάστηκε, διατηρείτε αποθηκευμένος σε ένα αρχείο, για την δημιουργία ενός λεξιλογίου, σε μορφή **text(Vocabulary.csv)**.

Ο κώδικας εκτελεί την συνάρτηση **CreateInvIndex** η οποία, διαπερνώντας έναν - έναν τους όρους του λεξιλογίου, αναζητεί σε ποιά κείμενα υπάρχουν, και με την σειρά του, φτιάχνει το ανεστραμμένο αρχείο (Inverted Index File).
Έτσι, στην διάθεσή μας, έχουμε το αρχείο που μπορούμε να χρησιμοποιήσουμε σαν ευρετήριο, ώστε να αναζητήσουμε κείμενα με βάση τα queries. Η βάση του μοντέλου είναι η στοίβα και τα operators AND, OR, NOT.

Κάνοντας parse τα ερωτήματα απο το Queries.txt - αφού έχουν και αυτά προεπεξεργαστεί - τα εκτελούμε στο μοντέλο, προσθέτοντας ανάμεσά τους operators όπως το AND ή το OR με την συνάρτηση **add boolean operators**. Έτσι, φτιάχνουμε τα infix tokens, τα οποία και μετατρέπουμε σε postfix μορφή με την βοήθεια της συνάρτησης postfix, ώστε να είναι διαχειρίσιμα από την στοίβα.

Τέλος, ορίζοντας την λειτουργία των τελεστών AND, OR, NOT, εκτελούμε το boolean μοντέλο, ρυθμίζοντας τον operator που θα μπεί ανάμεσα σε κάθε όρο. Έτσι, σαρώνεται το postfix query. Αν το token είναι όρος, ανακτούνται τα κείμενα τα οποία περιέχουν τον όρο αυτό και προσθέτει το αποτέλεσμα (σύνολο ID κειμένων) στην στοίβα. Αν είναι ένας από τους τελεστές, υλοποιεί την αντίστοιχη πράξη και ξανά-προσθέτει τα αποτελέσματα στην στοίβα. Έτσι, στο τέλος περιέχεται το αποτέλεσμα στην στοίβα, όπου και επιστρέφεται.

Τρόπος διαχείρισης Queries

Τα ερωτήματα, αποθηκεύτηκαν αρχικά σε μια λίστα όπου και προ επεξεργάζεται η συνάρτηση getQueries, με αποτέλεσμα να περιέχει μόνο τις σημαντικές λέξεις. Έπειτα, με την συνάρτηση add boolean operators, προστίθεται ανάμεσα σε κάθε λέξη ο τελεστής που θέσαμε, και αποθηκεύουμε τα αποτελέσματα προς διαχείριση στο boolean queries οπου και δίνουμε στο μοντέλο εν τέλη να τρέξει.

Ερώτημα 3

Τρόπος Λειτουργίας του VSM

Για το ερώτημα 3 ζητήθηκε η υλοποίηση του Μοντέλου Διανυσματικού Χώρου (Vector Space Model - VSM). Το μοντέλο αυτό κάνει την πολύ γενική (και αρκετά λάθος στις περισσότερες περιπτώσεις) θεώρηση ότι κάθε λέξη σε ένα κείμενο μπορεί να θεωρηθεί ως ξεχωριστός άξονας στον χώρο των εννοιών, δηλαδή ότι είναι ανεξάρτητη από κάθε άλλη. Αυτή η απλή θεώρηση όμως μας επιτρέπει με πολύ απλές πράξεις να αναπαραστήσουμε κάθε κείμενο και ερώτημα ως διανύσματα σε αυτόν τον χώρο λέξεων πλέον και - ακόμα πιο σημαντικό - να τα συγκρίνουμε μεταξύ τους. Η κύρια επιτυχία του μοντέλου είναι ότι, σε αντίθεση με το Boolean, επιστρέφει μία κατάταξη των κειμένων ως προς την σχετικότητά τους με το ερώτημα βάση της ομοιότητας των αντίστοιχων διανυσμάτων. Αν και όχι πάντα σωστή, είναι αρκετά προσεγγιστική ως πρώτη φάση ανάκτησης κειμένων.

Όπως αναφέρθηκε, το μοντέλο θεωρεί κάθε κείμενο και ερώτημα ως μια συλλογή από λέξεις και τα αναπαριστά ως διανύσματα σε έναν χώρο όπου κάθε διάσταση είναι και μία λέξη του λεξιλογίου. Στο "Παράρτημα" υπάρχει λεπτομερής εξήγηση του πως φτιάξαμε το λεξιλόγιό μας από την συλλογή κειμένων και ερωτημάτων που μας δόθηκαν. Με βάση αυτό το λεξιλόγιο καλείται η συνάρτηση **inverseIndexFile()** η οποία υλοποιεί το ανεστραμμένο αρχείο που περιγράψαμε στο "Ερώτημα 1". Επειδή η πολυπλοκότητα του ευρετηρίου αυτού αυξάνεται σε σχέση με το Boolean Model, η δημιουργία του παίρνει λίγο παραπάνω χρόνο.

Αφού δημιουργηθεί το ανεστραμμένο ευρετήριο, καλείται μία συνάρτηση για την δημιουργία των διανυσμάτων των κειμένων και των ερωτημάτων. Μέσα σε αυτήν την συνάρτηση επίσης υπολογίζεται το βάρος κάθε λέξης ανάλογα το κείμενο ή το ερώτημα στο οποίο βρίσκεται. Το βάρος (weight) της λέξης σε κάθε διάνυσμα υπολογίζεται με τον εξής τρόπο:

• Για τα κείμενα, το βάρος βγαίνει από τον τύπο του TF-IDF (Term Frequency - Inverse Document Frequency). Το βάρος μιας λέξης στο κάθε κείμενο δίνεται από εξής τύπο:

> Αν "N" είναι ο αριθμός των κειμένων στην συλλογή, "n" είναι ο αριθμός των κειμένων στην συλλογή που εμφανίζεται ο όρος "term" και TF ο αριθμός εμφάνισης του όρου στο κείμενο, το βάρος είναι:

$$\mathbf{weight_{term} = log_2(1 + TF) * log_2(\frac{N}{n})}$$

Ο λόγος που επιλέχθηκε αυτός ο τύπος είναι για να γίνει κανονικοποίηση των αποτελεσμάτων και του βάρους ώστε να μην επηρεάζεται πολύ από ακραίες τιμές σε συχνότητες εμφάνισης. Ο συγκεκριμένος είναι ο πιο δημοφιλής μεταξύ πολλών επιλογών και ταίριαζε επαρκώς στα δεδομένα του προβλήματός μας.

• Για τα ερωτήματα άλλαξε λίγο ο τύπος του βάρους των λέξεων λόγω της μικρής τους έκτασης ως εξής:

$$\mathbf{weight_{term} = TF * log2(N/n)}$$

Ο λόγος για αυτήν την αλλαγή και προσφορά του TF ως έχει στο γινόμενο είναι για να έχει περισσότερη αξία μία λέξη που εμφανίζεται πολλές φορές στο ερώτημα στο τελικό αποτέλεσμα. Θέλαμε αυτή την ραγδαία αύξηση με τις πολλαπλές εμφανίσεις της λέξης οπότε το αφήσαμε σαν καθαρό παράγοντα μαζί με το ήδη υπολογισμένο IDF.

Εφόσον έχουμε τα βάρη των λέξεων, αμέσως μετά φτιάχνουμε τα διανύσματα των κειμένων και των ερωτημάτων με την συνάρτηση createVectors(). Σε αυτό το στάδιο επίσης υπολογίζουμε και το μέτρο κάθε διανύσματος για να χρησιμοποιηθεί μετέπειτα στον υπολογισμό του συνημιτόνου μεταξύ διανυσμάτων χωρίς να χρειάζεται να υπολογίζουμε κάθε φορά το μέτρο του διανύσματος κάθε κειμένου από την αρχή γιατί αυτό είναι κάπως ακριβή σαν πράξη. Άρα τελικώς δημιουργούμε ένα csv αρχείο με τα διανύσματα των κειμένων και των ερωτημάτων αλλα και με τα μέτρα τους.

Υλοποίηση ερωτημάτων

Στην συνέχεια, περιγράφεται εν συντομία πως υλοποιήθηκαν τα ερωτήματα πάνω στην συλλογή των κειμένων. Σε αυτό το σημείο έχουμε ήδη τα διανύσματα των κειμένων και των ερωτημάτων και απλά πρέπει να τα συγκρίνουμε μεταξύ τους. Η σύγκριση των διανυσμάτων επιτυγχάνεται μέσω του συνημιτόνου της μεταξύ τους γωνίας. Ο τύπος που δίνει το συνημίτονο είναι ο εξής:

$$\mathrm{sim}(\boldsymbol{q}, \boldsymbol{d_j}) = \frac{\vec{\boldsymbol{d_j}} \cdot \vec{\boldsymbol{q}}}{\|\vec{\boldsymbol{d_j}}\| \cdot \|\vec{\boldsymbol{q}}\|} = \frac{\sum_{i=1}^{t} \boldsymbol{w_{i,j}} \cdot \boldsymbol{w_{i,q}}}{\sqrt{\sum_{i=1}^{t} \boldsymbol{w_{i,j}^2}} \cdot \sqrt{\sum_{i=1}^{t} \boldsymbol{w_{i,q}^2}}}$$

Παρατηρούμε μερικά πράγματα εδώ:

• Τα μέτρα των διανυσμάτων τα έχουμε ήδη υπολογίσει οπότε δεν χρειάζεται να τα υπολογίζουμε κάθε φορά. Αυτό έγινε με την σκέψη ότι έχουμε 20 ερωτήματα (ή x στην γενική περίπτωση) και αντί να υπολογίσουμε το μέτρο διανυσμάτων 1000+ διαστάσεων 20 (ή x) φορές, το κάνουμε μόνο μία.

• Για να γλιτώσουμε πράξεις, στο εσωτερικό γινόμενο των διανυσμάτων συνεισφέρουν μόνο οι κοινές και στα δύο διανύσματα λέξεις. Αυτό συμβαίνει διότι λέξεις που δεν υπάρχουν σε κάποιο διάνυσμα απλά έχουν βάρος 0, οπότε κάθε γινόμενο με αυτές θα είναι 0 και συνεπώς δεν συνεισφέρουν καθόλου στο τελικό εσωτερικό γινόμενο. Άρα μας ενδιαφέρουν μόνο λέξεις που υπάρχουν και στο ερώτημα και στο κάθε κείμενο αφού μόνο αυτές έχουν μη μηδενικές τιμές και στα δύο. Με αυτές τις παρατηρήσεις, η υλοποίηση έγινε αρκετά πιο αποδοτική. Σε αρχικές υλοποιήσεις υπολογιζόταν το μέτρο κάθε διανύσματος επί τόπου και είχε περίπου $1.5x$ χειρότερη απόδοση από ότι περιγράψαμε παραπάνω.

Επίσης υπάρχει μία μέθοδος που υλοποιεί την γενική ιδέα του VSM που λέγεται **master_Matrix()**. Σε αυτήν φορτώνουμε όλα τα κείμενα και τα ερωτήματα σαν διανύσματα σε ένα μεγάλο πίνακα με όλους τους όρους, ένα αρκετά αραιό μητρώο. Οι πράξεις εδώ γίνονται πάνω και στις 9000 διαστάσεις των διανυσμάτων, το οποίο προσθέτει σημαντικά περισσότερο overhead στην απόδοση χωρίς ουσία. Έχει μείνει σαν υλοποίηση μόνο για σκοπούς σύγκρισης.

Τέλος κρατάμε μόνο τα ids των κειμένων των οποίων η ομοιότητα είναι πάνω από το threshold που έχουμε θέσει (το παίρνει ως όρισμα η συνάρτηση) και τα ταξινομούμε ανάλογα με την ομοιότητά τους με το ερώτημα. Τα τελικά αρχεία που δημιουργούνται με τις λίστες των ids είναι ταξινομημένα.

Προβλήματα κατά την υλοποίηση

Η υλοποίηση στην πορεία της παρουσίασε μερικά προβλήματα που έπρεπε να προσπεράσουμε. Αρχικά το θέμα με την σωστή αντιστοιχία των κειμένων με τα id τους ξεπεράστηκε εύκολα όπως εξηγήθηκε στο "Ερώτημα 1" με την χρήση του pandas και την αντιστοιχία του κειμένου με το id του και όχι με την θέση του στην λίστα. Έπειτα έπρεπε να σκεφτούμε τρόπους να το κάνουμε πιο αποδοτικό σε θέμα χρόνου, όπου και καταλήξαμε στην παραπάνω περιγραφή και σκέψη. Τέλος, για να μην χρειάζεται να ξανατρέχουν όλες οι μέθοδοι από την αρχή σε κάθε υλοποίηση, στο τέλος κάθε μεθόδου αποθηκεύονται τα αποτελέσματα της σε ένα csv αρχείο για να τα έχουμε έτοιμα προς χρήση από όποια μέθοδο τα χρειαστεί στο μέλλον. Αυτό έγινε χάριν της χρήσης του pandas και της εύκολης ανάγνωσης από ςσ σε DataFrames. Έτσι έχουμε βελτίωση απόδοσης σε διαφορετικά τρεξίματα του μοντέλου για διαφορετικά thresholds πχ. Στο "Παράρτημα" θα βρείτε μία πιο εκτενή εξήγηση της υλοποίησης σε μερικά σημεία της.

Ερώτημα 4

Η λειτουργία **calculate_metrics** έχει σχεδιαστεί για τον υπολογισμό βασικών μετρικών όπως η ακρίβεια, η ανάκληση. Η ακρίβεια μετρά το ποσοστό των ανακτημένων εγγράφων που είναι σχετικά, και υπολογίζεται διαιρώντας τον αριθμό των σχετικών εγγράφων που ανακτήθηκαν με τον συνολικό αριθμό των ανακτημένων εγγράφων. Η ανάκληση μετρά το ποσοστό των σχετικών εγγράφων που ανακτήθηκαν, και υπολογίζεται διαιρώντας τον αριθμό των σχετικών εγγράφων που ανακτήθηκαν με τον συνολικό αριθμό των σχετικών εγγράφων.

Η λειτουργία λαμβάνει ως εισόδους λίστες με ανακτημένα και σχετικά έγγραφα. Υπολογίζει τη διατομή αυτών των λιστών για να προσδιορίσει τον αριθμό των σχετικών εγγράφων που ανακτήθηκαν και στη συνέχεια εφαρμόζει τους τύπους για την ακρίβεια και την ανάκληση. Τελικά, επιστρέφει ένα αποτέλεσμα με τις υπολογισμένες μετρικές.

Boolean μοντέλο

Στο Boolean μοντέλο, παρατηρήθκε η εξής συμπεριφορά στην ανάκτηση κειμένων:
Όταν το ερώτημα περιλαμβάνει τον τελεστή **AND**, το μοντέλο αναζητά έγγραφα που περιέχουν όλους τους όρους του ερωτήματος ταυτόχρονα. Αυτό έχει ως αποτέλεσμα τη δραστική μείωση του αριθμού των εγγράφων που πληρούν τα κριτήρια. Σε περιπτώσεις όπου οι όροι του ερωτήματος είναι σπάνιοι ή δεν συσχετίζονται έντονα μεταξύ τους, το αποτέλεσμα είναι η ανάκτηση ελάχιστων ή ακόμα και μηδενικών εγγράφων.

Αντίθετα, όταν χρησιμοποιείται ο τελεστής **OR**, το μοντέλο αναζητά έγγραφα που περιέχουν έναν ή περισσότερους από τους όρους του ερωτήματος. Αυτό οδηγεί σε μεγάλη διεύρυνση των αποτελεσμάτων, καθώς ακόμη και η ύπαρξη ενός μόνο όρου είναι αρκετή για να συμπεριληφθεί ένα έγγραφο στα αποτελέσματα.

Αυτό το φαινόμενο δείχνει ότι το μοντέλο καταλήγει στα δύο άκρα. Με τον AND περιορίζει υπερβολικά τα αποτελέσματα, ενώ με τον OR παράγει υπερβολικά μεγάλο αριθμό αποτελεσμάτων, μειώνοντας την ακρίβεια και την χρησιμότητα των ανακτήσεων.

ΑΚΡΙΒΕΙΑ:
• Με **AND** η ακρίβεια είναι υψηλή. Ο λόγος είναι ότι ο τελεστής AND περιορίζει τα αποτελέσματα μόνο σε έγγραφα που πληρούν όλα τα κριτήρια του ερωτήματος, με αποτέλεσμα τα περισσότερα ανακτημένα έγγραφα να είναι σχετικά. Ωστόσο, ο πολύ μικρός αριθμός ανακτημένων εγγράφων μπορεί να περιορίσει τη χρησιμότητα της ακρίβειας.

• Με **OR** η ακρίβεια αναμένεται να είναι χαμηλή, καθώς το OR επιστρέφει έναν πολύ μεγάλο αριθμό εγγράφων, πολλά από τα οποία δεν είναι σχετικά. Αυτό μειώνει την αναλογία των σχετικών εγγράφων στα συνολικά ανακτημένα.

ΑΝΑΚΛΗΣΗ:
• Με **AND** η ανάκληση είναι χαμηλή, καθώς ο τελεστής AND περιορίζει δραστικά τα αποτελέσματα, αφήνοντας έξω πολλά σχετικά έγγραφα. Αυτό οφείλεται στο γεγονός ότι τα σχετικά έγγραφα που δεν περιέχουν όλους τους όρους του ερωτήματος δεν θα ανακτηθούν.

• Με **OR** η ανάκληση είναι συνήθως υψηλή, διότι ο τελεστής OR συμπεριλαμβάνει οποιοδήποτε έγγραφο περιέχει έστω και έναν όρο από το ερώτημα. Αυτό σημαίνει ότι η πλειονότητα των σχετικών εγγράφων θα ανακτηθεί, μαζί όμως με πολλά μη σχετικά έγγραφα.

ΧΡΟΝΟΣ:
Ο χρόνος εκτέλεσης κάθε ερωτήματος, είναι πολύ μικρός (0.0001), ειδικότερα στον AND τελεστή όπου η ακρίβεια 5 δεκαδικών ψηφίων δεν είναι αρκετή. Αυτό οφείλετε στην απλότητα του Boolean μοντέλου αφού χρησιμοποιεί λογικές πράξεις (AND, OR, NOT) για να ελέγξει αν ένα έγγραφο πληροί τα κριτήρια αναζήτησης, κάτι που είναι αρκετά απλό και δεν απαιτεί πολύπλοκους υπολογισμούς ή υπολογισμούς απόστασης, όπως τα πιο εξελιγμένα μοντέλα Vector Space. Επίσης, στο Boolean μοντέλο, είτε ένα έγγραφο είναι συμβατό με την αναζήτηση είτε όχι, οπότε δεν απαιτείται η σύγκριση βαθμών ή άλλων παραμέτρων όπως σε πιο περίπλοκα μοντέλα (π.χ., TF-IDF).

Vector Space μοντέλο

Τα αποτελέσματα που πήραμε στο VSM έχουν να κάνουν πάρα πολύ με το threshold της ομοιότητας μεταξύ των διανυσμάτων που θέλαμε να έχουμε. Ενδεικτικά, τα αποτελέσματα που παρουσιάζουμε είναι για similarity ’ 0.1. Να σημειωθεί ότι δεν υπάρχει σταθερή τιμή similarity που θα έπρεπε να στοχεύσει να έχει το μοντέλο καθώς αυτή εξαρτάται πολύ από το ερώτημα και τα κείμενα κάθε συλλογής. Ένα αποδεκτό εύρος τιμών που θα μπορούσαμε να θέσουμε είναι το [0.01, 0.5]
Για threshold = 0.1, οι μετρικές για κάθε ερώτημα είναι οι εξείς:

Query	precision	recall
1	0.8	0.143
2	0.833	0.081
3	0.429	0.094
4	0.294	0.625
5	0.333	0.065
6	0.3	0.255
7	0.5	0.026
8	0.286	0.143
9	0.3	0.529
10	0.4	0.571

11	0.458	0.036
12	0.765	0.12
13	1	0.056
14	0.2	0.023
15	0.75	0.333
16	0.75	0.25
17	0.208	0.455
18	0.429	0.2
19	0.75	0.6
20	1	0.273

Παρατηρούμε ότι σε πολλά ερωτήματα είναι πιο ψηλά το precision από ότι το recall. Αυτό είναι λογικό καθώς όσο πιο μεγάλο είναι το similarity threshold που βάζουμε, τόσο πιο όμοια πρέπει να είναι τα διανύσματα των κειμένων με του ερωτήματος για να τα επιστρέψει το μοντέλο. Άρα είναι και πιο αυστηρό στην ομοιότητα αλλά επίσης επιστρέφει και λιγότερα κείμενα λόγω αυτής της απαίτησης, που σημαίνει ότι το recall θα πέσει πιθανότατα. Αν δούμε τα αποτελέσματα για threshold = 0.05 επιβεβαιώνεται αυτή η σκέψη:

1	0.312	0.179
2	0.172	0.177
3	0.258	0.25
4	0.148	0.75
5	0.118	0.065
6	0.195	0.511
7	0.132	0.132
8	0.102	0.357
9	0.197	0.824
10	0.212	0.857

11	0.343	0.118
12	0.585	0.287
13	1	0.056
14	0.393	0.256
15	0.125	0.333
16	0.571	0.333
17	0.084	0.636
18	0.097	0.2
19	0.25	0.6
20	0.231	0.273

Εδώ βλέπουμε σχεδόν σε όλα τα ερωτήματα να έχει πέσει το precision αλλά να έχει ανέβει (έστω και λίγο) το recall, το οποίο επιβεβαιώνει το σκεπτικό μας. Αν βάζαμε το threshold = 0 τότε τα αποτελέσματα θα είναι σαν να είχαμε το Boolean Model με OR operator καθώς θα επέστρεφε όλα τα κείμενα που θα περιέχουν έστω και μία από τις λέξεις του ερωτήματος μέσα τους. Από την άλλη αν βάζαμε threshold = 1 δεν θα επέστρεφε τίποτα καθώς τότε θα έπρεπε τα διανύσματα να είναι ίδια. Επίσης τα κείμενα που επιστρέφονται είναι σε φθινουσα σειρα ως προς την ομοιότητα οπότε έχουμε και μία ταξινόμηση. Εδώ δεν αξιοποιείται αυτή η λεπτομέρεια αλλά είναι αρκετά σημαντικό ότι το μοντέλο επιστρέφει και τα πιο όμοια κείμενα πρώτα.

Όσον αφορά τον χρόνο εκτέλεσης, το VSM είναι αρκετά πιο ακριβό από το Boolean σε θέμα χρόνου καθώς κάνει πιο πολλές πράξεις και μεταφορές στην μνήμη. Συγκεκριμένα η δικιά μας υλοποίηση του VSM έχει πολλες μεταφορές από και προς στην μνήμη αλλά και μη βελτιστοποιημένο κώδικα ως προς την απόδοση σε μερικά σημεία οπότε και είναι πιο αργό. Κατά μέσο όρο ο χρόνος για την ευρετηριοποίηση και την δημιουργία του λεξιλογίου είναι 2 seconds, για την δημιουργια των διανυσμάτων είναι 8 seconds και για κάθε ερώτημα είναι περίπου 1 second. Από το 0 αν τρέξει το μοντέλο να κάνει και την προεπεξεργασία και να κάνει τα ερωτήματα θέλει περίπου 30 seconds. Αν υπάρχουν ήδη τα απαραίτητα αρχεία, του παίρνει 20 seconds.

Γενικά τα αποτελέσματα των μετρικών είναι μέσα στα πλαίσια των αναμενόμενων και αυτό μας επιβεβαιώνει την θεωρία για το VSM σε σχέση με το Boolean. Το VSM γενικά προσπαθεί να λάβει υπ'όψην του την σημασία των λέξεων ως ένα βαθμό με μερικές αυθαίρετες υποθέσεις, όπως την ανεξαρτησία των λέξεων μεταξύ τους, που είναι λάθος. Αυτό έχει σαν αποτέλεσμα να επιστρέφονται κείμενα που λίγο ή πολύ κάπως θα σχετίζονται με το ερώτημα που δόθηκε και ανάλογα την αυστηρότητα που έχουμε θέσει θα έχουμε λιγότερα αλλά πιο σχετικά κείμενα προς το ερώτημα (higher precision - lower recall). Αρκεί να μην γίνουμε υπερβολικά αυστηροί γιατί τότε κανένα κείμενο δεν θα επιστραφεί (όπως πχ στο Boolean Model με το AND operator που είναι η απόλυτη απαίτηση).

Σύγκριση μοντέλων

Αν συγκρίνουμε τα δύο μοντέλα διαπιστώνουμε ότι το Boolean είναι τάξεις πιο γρήγορο από το VSM λόγω της υπερβολικής απλότητάς του. Ο χρόνος εκτέλεσης στο ίδιο σύστημα για κάθε ερώτημα ήταν περίπου 20 φορές πιο μικρός από αυτόν του VSM ακόμα και μετά τις βελτιώσεις που έγιναν στο τελευταίο. Όμως, ενώ είναι πιο γρήγορο, τα αποτελέσματα του Boolean Model είναι αρκετά έξω από την πραγματικότητα και δεν έχουν καμία σχεδόν αξία ως έχουν καθώς αγγίζουν τα 2 άκρα: ή επιστρέφονται σχεδόν όλα τα κείμενα χωρίς ταξινόμηση είτε οριακά κανένα. Επιπλέον, τα αποτελέσματα δεν έχουν ταξινόμηση ή προτεραιότητα, είναι όλα ισόβαθμα που τους ρίχνει ακόμα περισσότερο την αξία.

Σε αντίθεση με αυτό, το VSM επιστρέφει πιο ρεαλιστικά αποτελέσματα. Οι μετρικές του precision και του recall μας δείχνουν ότι τα αποτελέσματα είναι κάπως σχετικά με το ερώτημα και αυτό εξαρτάται από το threshold που δίνουμε. Επίσης τα αποτελέσματα είναι ταξινομημένα, το οποίο προσδίδει περισσότερη αξία στα αποτελέσματα αφού τα πιο σχετικά κείμενα που προβλέπει το μοντέλο τα φέρνει πιο πάνω στα αποτελέσματα. Όλα αυτά όμως γίνονται επειδή το VSM είναι πιο περίπλοκο ως προς την υλοποίηση του και τις δομές που χρησιμοποιεί. Η πολυπλοκότητα αυτή φαίνεται όχι μόνο στην ποιότητα των αποτελεσμάτων αλλά και στον χρόνο εκτέλεσης ο οποίος είναι αρκετά πιο μεγάλος από το απλό Boolean μοντέλο. Κάθε ερώτημα παίρνει περίπου 1 δευτερόλεπτο να εκτελεστεί, το οποίο είναι αρκετά γρήγορο γενικά αλλα συγκριτικά με το Boolean είναι πολύ πιο αργό.

Και τα δύο μοντέλα όμως χαρακτηρίζονται ως πάρα πολύ γρήγορα και ειδικά το VSM ήταν το προτιμότερο μοντέλο παλαιότερα λόγω της ταχύτητας του, της δυνατότητας για συμπίεση του ευρετηρίου του και της ποιότητας των αποτελεσμάτων του.

Ερώτημα 5

Παράρτημα

Κάθε συνάρτηση έχει επαρκές πλήθος σχολίων αλλά ακολουθεί λεκτική περιγραφή του τι κάνει η κάθε μέθοδος.

ΠΡΟΕΠΕΞΕΡΓΑΣΙΑ

Έχει δημιουργηθεί ένα python αρχείο με το όνομα Vocabulary.py στο οποίο περιέχονται όλες οι μέθοδοι της βασικής προεπεξεργασίας που χρειάζεται γενικά για την ευρετηριοποίηση. Κάθε μέθοδος σε αυτό το στάδιο γράφτηκε με την ιδέα της καλύτερης απόδοσης στο νου οπότε και θα παρουσιαστούν μερικά σημεία που αυτό φαίνεται. Επίσης θα εξηγηθούν τεχνικές προεπεξεργασίας που αξιοποιήθηκαν ή μη και γιατί κατά την περιγραφή των μεθόδων. Στο κομμάτι της προεπεξεργασίας πριν της υλοποίησης των ευρετηρίων, υλοποιήθηκαν οι εξείς μέθοδοι:

· **getQueries() / getRelevant():**
Αυτές οι μέθοδοι υλοποιήθηκαν για μετέπειτα χρήση στην πράξη των ερωτημάτων. Λειτουργούν με παρόμοιο τρόπο καθώς διαβάζουν τα αντίστοιχα txt αρχεία ανά γραμμή. Τα αποτελέσματα της .readline() είναι ένα string με τα περιεχόμενα της γραμμής όπου το τελευταίο στοιχείο είναι το "\n" το οποίο και αφαιρείται και στις δύο μεθόδους. Στην getQueries συγκεκριμένα επίσης ανά string εφαρμόζεται και η μέθοδος removeStopWords() που θα αναφερθεί και πιο κάτω για την αφαίρεση άχρηστων ως προς τα μοντέλα μας λέξεων. Και οι δύο μέθοδοι επιστρέφουν ένα pandas.Series αντικείμενο με δεδομένα είτε τα queries χωρίς τα stopwords είτε τις λίστες των ids των σχετικών ως προς έκαστο ερώτημα κειμένων.

· **removeStopWords()**:
Σαν στάδιο της προεπεξεργασίας θέλαμε να αφαιρέσουμε τις κοινότυπες λέξεις που υπάρχουν σε κάθε κείμενο, τα λεγόμενα stopwords. Ο λόγος είναι ότι δεν θα συνέσφεραν καθόλου στο αποτέλεσμα των μοντέλων πρακτικά παρά μόνο αρνητικά καθώς θα μεγάλωνε το vocabulary και γενικά θα αυξάνονταν οι πράξεις που θα γινόντουσαν χωρίς ουσία. Για χάρην ευκολίας αξιοποίηθηκε η βιβλιοθήκη **nltk** για αυτόν τον σκοπό και τα stopwords που παρέχει. Εμείς απλά φορτώσαμε τα stopwords σε ένα set της python, πήραμε μία λίστα από strings σαν όρισμα, μετατρέψαμε όλες τις λέξεις των string σε πεζά γράμματα, σπάσαμε τα strings σε λέξεις και αφαιρέσαμε όλα τα stopwords από τις λίστες των λέξεων. Μετά ενώσαμε τις λίστες σε ένα string και κάθε string το προσθέσαμε στην επιστρεφόμενη λίστα toGet().

Παρατήρηση γίνεται στο ότι εν τέλει κρατάμε μόνο τα πεζά γράμματα και ο λόγος είναι ότι μπορεί να υπήρχαν ίδιες λέξεις με διαφορετικό casing. Αυτό δημιουργούσε θέμα στα ευρετήρια οπότε κρίθηκε ευκολότερο να μετατραπούν όλες οι λέξεις στο ίδιο casing σε αυτό το στάδιο.

· **ParseDocs()**:
Αυτή η μέθοδος προεπεξεργάζεται την συλλογή κειμένων και τα μαζεύει όλα μαζί στο ίδιο αρχείο csv. Επίσης, όπως αναφέρεται και στην εκφώνηση, λείπουν κάποια κείμενα από την συλλογή, πράγμα που καλούμαστε να διαχειριστούμε για την σωστή αντιστοιχία των κειμένων και των id τους. Για τον λόγο αυτόν χρησιμοποιούμε τα dictionaries της python όπου αποθηκέυουμε το id του κάθε κειμένου ως κλειδί και ως data τις λέξεις που περιέχει (αφού τις επεξεργαστούμε πρώτα με την removeStopWords και αφαιρώντας έξτρα spaces) ως ένα string. Τέλος, επιστρέφεται ένα pandas.Series με indexes τα id των κειμένων και ως δεδομένα σε κάθε index το αντίστοιχο string. Έτσι, πιο μετά, μπορούμε να αποθηκεύσουμε την ουσία όλων των κειμένων σε ένα αρχείο αντί να τα έχουμε στον φάκελο σκόρπια όπως μας δόθηκαν.

· **RemDuplicates()**:
Μία απλή μέθοδος που επιστρέφει ως λίστα τις λέξεις ενός στρινγ χωρίς επαναλαμβανόμενες λέξεις. Δεν διατηρεί την σειρά τον λέξεων.

·**CreateVocab()**:
Η βασική μέθοδος που δημιουργεί το λεξιλόγιο της συλλογής των κειμένων. Η λογική της είναι απλή. Αρχικά δες αν υπάρχει το Docs.csv αρχείο. Αν δεν υπάρχει θα μπει στο except όπου θα καλέσει την parseDocs() και το επιστρεφόμενο pandas.Series το αποθηκεύει στο Docs.csv και ξανακαλεί τον εαυτό της. Αφού πλέον υπάρχει το Docs.csv, ακολουθείται η εξής απλή λογική:
1. Αποθήκευσε όλο το csv που φτιάχτηκε σε ένα DataFrame και αποθήκευσε σε ένα Series τα κείμενα.
2. Εφάρμοσε σε κάθε κείμενο του Σεριες την remDuplicates() ώστε να αφαιρεθούν διπλότυπες εμφανίσεις λέξεων μέσα σε κάθε κείμενο.
3. Πρόσθεσε στο υπάρχον Series και το pandas.Series που επιστρέφει η getQueries() για να προστεθούν λέξεις από τα queries στο λεξιλόγιο σε περίπτωση που δεν υπάρχουν στα κείμενα.
4. Μετάτρεψε τα δεδομένα του Series σε μία λίστα και ένωσέ τα σε ένα μεγάλο masterString το οποίο θα περάσουμε πάλι από την remDuplicates. Η επιστρεφόμενη λίστα από την μέθοδο είναι το vocabulary που θέλαμε.
5. Αποθήκευσε την λίστα σε ένα Series και αυτό αποθήκευσέ το στο Vocabulary.csv.

Μερικές σημειώσεις για το createVocab():
· Στο βήμα 3 προσθέτουμε και τα queries στο σύνολο των τεξτς γιατί διαπιστώθηκε ότι όντως υπήρχε λέξη σταqueries που δεν υπάρχει στα κείμενα και αυτό δημιουργούσε θέματα μετά στα ευρετήρια.

· Ο λόγος που εφαρμόσαμε την remDuplicates πρώτα σε κάθε string ξεχωριστά και μετά στα φιλτραρισμένα όλα μαζί είναι λόγω απόδοσης. Επειδή η remDuplicates σπάει το strinγ σε λίστα λέξεων και κάνει πράξεις πάνω σε αυτές, αν κάναμε πρώτα όλα τα στρινγς ένα masterString και το δίναμε στην συνάρτηση, θα τελείωνε μεν σε ένα βήμα το ζητούμενο αλλά θα έπαιρνε σαφώς παραπάνω χρόνο (όχι αρκετά παραπάνω αλλά άξιο να σημειωθεί). Για αυτό διατηρήθηκε η λογική του “Διαίρει και Βασίλευε” και τρέξαμε την συνάρτηση σε μικρότερα strings.

Επίσης την τρέχουμε δεύτερη φορά διότι ενώ αφαιρέσαμε τις πολλές εμφανίσεις μίας λέξης σε ένα κείμενο, αυτή η λέξη πιθανότατα υπάρχει και σε άλλα κείμενα. Το δεύτερομasterString με τα φιλτραρισμένα texts είναι αρκετά πιο μικρό από το αρχικό χωρίς τα φιλτραρισμένα texts.

· Το Vocabulary.csv έχει χρησιμοποιήθει σαν πάτημα και για τα δύο Inverted Indexing Files.
Υπάρχει και μία μέθοδος που κάνει stemminγ στις λέξεις πάλι με την βοήθεια του **nltk** αλλά δεν την χρησιμοποιήσαμε διότι δεν φάνηκε να χρειάζεται, το λεξιλόγιο ήταν ήδη αρκετά μικρό και δεν θα είχαμε κάποια ουσιώδη βελτίωση στα αποτελέσματα.

Vector Space Model

Στο αρχείο Inverse_Index_VSM.py είναι όλες οι μέθοδοι και ο κώδικας που υλοποιεί το VSM. Έχουν προστεθεί όλα τα απαραίτητα σχόλια στον κώδικα για την κατανόηση του αλλά ακολουθεί μία εξήγηση της λογικής και της εκτέλεσης των μεθόδων. Επίσης όλα τα παραγόμενα αρχεία αποθηκεύονται μέσα στον φάκελο collection που μας δόθηκε. Οι μέθοδοι που φτιάχτηκαν για αυτό είναι και καλά σημεία διαχώρισης των σταδίων του μοντέλου μέχρι να βγάλει αποτέλεσμα. Αυτές είναι οι εξείς:

inverseIndexFile():
Αυτή η μέθοδος δημιουργεί το ευρετήριο για το VSM. Αρχικά φορτώνει σε ένα DataFrame το λεξιλόγιο που ήδη υπάρχει (αν δεν υπάρχει καλεί την createVocab() και καλεί τον εαυτό της πάλι) και αρχικοποιεί τα στοιχεία της στήλης που θα κρατήσει τα κείμενα με το αντίστοιχο tf σε κενές λίστες για να προσαυξηθούν αργότερα. Ακόμα φορτώνει σε ένα διαφορετικό DataFrame όλα τα κείμενα με τα id τους και δημιουργεί και ένα Series που θα κρατήσει τα IDF των λέξεων.

Έπειτα πάει σε κάθε κείμενο και υπολογίζει το tf κάθε λέξης. Για κάθε ξεχωριστή λέξη που βλέπει στο κείμενο, την αναζητά στο DataFrame του λεξιλογίου και προσθέτει στην λίστα το id του κειμένου με το tf που υπολόγισε (ο υπολογισμός είναι κανονικοποιημένος). Επίσης για κάθε εμφάνιση λέξης σε ξεχωριστά κείμενα, αυξάνεται ο IDF μετρητής του IDF Series.

Αφού περαστούν όλα τα κείμενα, έχουμε στο IDF Series μετρήσει όλες τις εμφανίσεις των λέξεων και έτσι το προσθέτουμε σαν στήλη στο vocab_DataFrame. Με αυτό συνεχίζουμε με τον υπολογισμό του tf-idf βάρους για κάθε λέξη στο αντίστοιχο κείμενο και το αποθηκεύουμε στην αντίστοιχη θέση του tf. Αυτό έγινε για να μην χρειάζεται να υπολογίζεται το βάρος πολλές φορές και να το έχουμε κοντά στο αντίστοιχο κείμενο αντί για απλά τον αριθμό εμφανίσεων της λέξης. Τέλος αποθηκεύουμε το αρχείο σαν ένα csv με το όνομα Inverse_Index.csv.

Ενδεικτικά, μία γραμμή του αρχείου αυτού είναι η εξής:

abilities,"[[145, 9.228], [448, 9.228]]",9.228

createVectors(): Η μέθοδος δημιουργεί τα διανύσματα από τα κείμενα και τα ερωτήματα καθώς και υπολογίζει τα μέτρα τους. Αρχικά φορτώνεται το inverted indexing file σε ένα DataFrame και δημιουργείται ένα DataFrame με indexes τα ids των κειμένων το οποίο θα κρατήσει τα διανύσματα και τα μέτρα τους. Τα διανύσματα θα είναι αποθηκευμένα ως δικτιοναριες της python, είναι πιο αποδοτικό στην αποθήκευση τον λέξεων ως άξονες και των βαρών ως τιμές.

Μετά πάμε σε κάθε λέξη του iif και κοιτάμε τα κείμενα στα οποία εμφανίζεται. Για κάθε κείμενο, βρίσκουμε το id του στο άλλο DataFrame και προσθέτουμε την λέξη με το αντίστοιχο βάρος της στο dictionary. Παράλληλα, προσθέτουμε στο μέτρο του κειμένου το τετράγωνο του βάρους. Στο τέλος όλων των επαναλήψεων, κάθε id κειμένου είναι αντιστοιχισμένο σε ένα dictionary με όλες τις λέξεις του κειμένου ως κλειδιά και τα βάρη τους στο κείμενο ως δεδομένα και σε δεύτερη στήλη το άθροισμα των τετραγώνων των βαρών.

Μετά κάνουμε ακριβώς το ίδιο για τα ερωτήματα καλώντας την συνάρτηση getQueries() που υλοποιήσαμε πιο πριν. Στο τέλος καταλήγουμε με ένα αντίστοιχο DataFrame.

Μετά πάμε σε κάθε τιμή που υπάρχει στις στήλες "norm" που κρατάει τα αθροίσματα των τετραγώνων και αποθηκεύουμε την τετραγωνική τους ρίζα για να έχουμε τα μέτρα τους εν τέλει. Τέλος, αποθηκεύουμε τα DataFrames στα αντίστοιχα csv αρχεία για να κάνουμε γρήγορη ανάκτηση των διανυσμάτων αντί να τα φτιάχνουμε κάθε φορά.

Να σημειωθεί ότι ο υπολογισμός των μέτρων προσδίδει ένα overhead υπολογισμού αισθητό σε σύγκριση με το να μην γίνεται αλλά βελτιώνει κατά πολύ την απόδοση των ερωτημάτων. (αναφορικά, χωρίς τα μέτρα, τα διανύσματα φτιάχνονταν σε περιπου 2s ενώ με τα μέτρα παίρνει $8.5s$)

getResults(threshold): Αυτή είναι η μέθοδος που υλοποιεί τα ερωτήματα και επιστρέφει τα σχετικά κείμενα.
Αφού φορτώσει από τα csv τα διανύσματα και το inverted index file σε DataFrames, δημιουργεί ένα dictionary για τα αποτελέσματα και ενα DataFrame για τις μετρικές καθώς και φορτώνει και τις λίστες με τα σχετικά κείμενα που μας δόθηκαν.

Εκτελείται ένας βρόχος για κάθε ερώτημα που μας δόθηκε. Στην αρχή κάθε βρόχου αρχικοποιούμε και έναν timer για να μετρήσουμε πόση ώρα πήρε η εκτέλεση κάθε ερωτήματος. Για κάθε ερώτημα κρατάμε βρίσκουμε τα σχετικά κείμενα βάση των λέξεων του και φτιάχνουμε ένα DataFrame με τις λέξεις του ερωτήματος και τα βάρη τους.

Μετά για κάθε σχετικό κείμενο με το ερώτημα φτιάχνουμε το αντίστοιχο DataFrame και συνενώνουμε το DataFrame της ερώτησης με αυτό πάνω στις κοινές μόνο

λέξεις τους για τον υπολογισμό της ομοιότητας των διανυσμάτων. Εξηγήθηκε στο "Ερώτημα 3" γιατί επιλέξαμε να γίνει αυτό και γιατί είναι σωστό. Αφού φτιάξουμε και το τελικό DataFrame, περνάμε τα διανύσματα στην compute_Similarity μαζί με τα μέτρα τους και αυτή επιστρέφει την τιμή του συνημιτόνου της γωνίας τους την οποία και κρατάμε σε ένα dictionary με το id του κειμένου. Αυτό γίνεται για κάθε κείμενο σχετικό με το ερώτημα. Αφού βρεθούν όλες οι τιμές για όλα τα κείμενα, κρατάμε μόνο αυτές που ξεπερνούν ένα threshold το οποίο το ορίζουμε από το κάλεσμα της συνάρτησης.

Έτσι καταλήγουμε να έχουμε για το ερώτημα μόνο τα πιο σχετικά κείμενα βάση των υπολογισμών. Τυπώνουμε στο τερματικό τα αποτελέσματα των μετρικών με βάση τα αποτελέσματα του ερωτήματος και τον χρόνο που πήρε για να βρεθούν. Επίσης κρατάμε σε ένα DataFrame τις μετρικες και τα αποτελέσματα για να τα αποθηκεύσουμε μετά.

Αφού γίνει αυτό για κάθε ερώτημα, έχουμε τυπώσει όλα τα αποτελέσματα και τέλος αποθηκεύουμε στα αντίστοιχα csv τις μετρικές και την λίστα με τα επιστρεφόμενα κείμενα για κάθε ερώτημα.

calculate_idf(number of docs, idf):
Μία απλή μέθοδος που υπολογίζει το κανονικοποιημένο idf της λέξης. Επειδή υπήρξαν λέξεις που υπήρχαν μόνο σε ερωτήματα και όχι σε κείμενα, το idf τους που πέρναγε σαν όρισμα που είναι ο αριθμός εμφάνισης τους στα κείμενα ήταν 0 οπότε και πέταγε error η πράξη. Για αυτό τα idf = 0 τα κάναμε 1 το οποίο δεν αλλάζει καθόλου στην κανονικοποίηση, παίρνουν μέγιστη τιμή που όμως δεν έχει σημασία.

compute_Similarity(vec_1, vec_1_norm, vec_2, vec_2_norm)
Μέθοδος που υπολογίζει την ομοιότητα μεταξύ τουvec_1 και του vec_2. Επειδή έχουμε ήδη υπολογίσει τα μέτρα τους, απλά κάνουμε το dot product μεταξύ τους και επιστρέφουμε το ανάλογο αποτέλεσμα που μας λέει ο τύπος. Αυτό βελτιώνει κατά πολύ την απόδοση του ερωτήματος.

compute_Similarity(vec_1, vec_2,)
Η ίδια μέθοδος όμως χωρίς τα μέτρα των διανυσμάτων, τα υπολογίζει επί τόπου. Κρατήθηκε σαν υλοποίηση για σκοπούς σύγκρισης.

calculate_metrics(retrieved_docs, relevant_docs):
Αν επιστράφηκαν κείμενα, υπολογίζει τις μετρικές precision και recall και τις επιστρέφει με 3 δεκαδικά ψηφία.

master_Matrix():
Μέθοδος που υλοποιήθηκε για να συγκριθεί η θεωρητική προσέγγιση του VSM όπου συγκρίνονται τα διανύσματα με 9000+ διαστάσεις. Ακολουθεί παρόμοιο σκεπτικό με την getResults() αλλά φορτώνει όλα τα διανύσματα ερωτημάτων και κειμένων σε ένα τεράστιο μητρώο όπου σαν ινδεξες έχει όλο το λεξιλόγιο

και σαν στήλες έχει τα id των κειμένων και τα ερωτήματα. Έτσι οι υπολογισμοί γίνονται μεταξύ ολόκληρων στηλών. Σαν απόδοση ήταν αρκετά πιο αργό από την προηγούμενη μέθοδο.
Συγκριτικά, συνολικός χρόνος για όλα τα ερωτήματα με όλα τα απαραίτητα αρχεία να υπάρχουν μεταξύ των δύο υλοποιήσεων: getResults: 20,58 seconds getResults (χωρίς τα vectors.csv): 29,076 seconds master_Matrix: 51,331 seconds