Πανεπιστήμιο Δυτικής Αττικής
Σχολή Μηχανικών
Τμήμα Μηχανικών Πληροφορικής και Υπολογιστών

Εργασία Ανάκτησης Πληροφορίας

Διαννής Ιωάννης (21390053)

Ζαφείρας Γρηγόριος (21390060)

https://github.com/nestalert/information_retrieval

Contents

1. Περιγραφή της εργασίας

Ο στόχος της εργασίας είναι η δημιουργία μιας ολοκληρωμένης και αυτοτελούς μηχανής αναζήτησης. Αυτή η μηχανή θα πρέπει να μπορεί να δέχεται μια συλλογή εγγράφων, να την προετοιμάζει για αναζήτηση, και τέλος να επιτρέπει στον χρήστη να κάνει αναζητήσεις μέσω μιας διεπαφής. Ο χρήστης θα έχει την επιλογή να διαλέξει αλγόριθμο αναζήτησης.

Η αποτελεσματικότητα της μηχανής αναζήτησης θα κριθεί από την ταχύτητα, ακρίβεια, και αποτελεσματικότητα με την οποία παρουσιάζει τα αποτελέσματά της.

2. Βήματα υλοποίησης

2.1. Συλλογή δεδομένων

Στόχος: Η δημιουργία μιας συλλογής δεδομένων σε μορφή JSON.

Για τις αρχικές δοκιμές, προτιμήθηκε για λόγους ευκολίας να δημιουργηθεί μια μικρή συλλογή. Ο πρώτος κώδικας που επιτέλεσε αυτόν τον σκοπό ήταν ο scrape, που συλλέγει τα άρθρα «επιπέδου 1» της Wikipedia. Το scrape χρησιμοποιεί το BeautifulSoup για να επεξεργαστεί το «καθαρό» html της Wikipedia, βρίσκει τον τίτλο και περιεχόμενο του άρθρου, και τα αποθηκεύει στο articles.json με τα πεδία “title” και “content”. Όλα τα δεδομένα που περνάνε αυτό το βήμα πρέπει να έχουν αυτήν την δομή.

Όταν παρουσιάστηκε η ανάγκη για δοκιμές με διαφορετικά δεδομένα, υλοποιήθηκε το random_scrape. Το κύριο πρόβλημα στην υλοποίηση μιας τυχαίας

συλλογής ήταν ότι πολλές από τις σελίδες στην Wikipedia είναι υπερβολικά μικρές ή/και όχι άρθρα, αλλά συζητήσεις, φωτογραφίες κ.α. Η λύση του προβλήματος έρχεται (εν μέρει) από το API της Wikipedia. Μέσω αυτού μπορούμε να ζητήσουμε τυχαία άρθρα που πληρούν συγκεκριμένες προϋποθέσεις. Η παράμετρος «rnnamespace» καθορίζει τον τύπο αρχείου. Εδώ είναι 0, που συλλέγει «κύρια άρθρα». Σχετικά με το μήκος του κάθε άρθρου, το API δεν έχει κάποια παράμετρο. Η μόνη λύση που βρέθηκε ήταν να μετράμε τοπικά το μήκος και να μην αποθηκεύουμε άρθρα που είχαν λιγότερο από ένα αυθαίρετο νούμερο χαρακτήρων (1000). Αυτό σημαίνει ότι περίπου 15% των άρθρων απορρίπτονται, οπότε εάν ζητήσουμε 20 άρθρα, θα αποθηκευτούν μόνο 16. Εφόσον δεν μας ενδιαφέρει ακριβώς το νούμερο, θεωρήθηκε αποδεκτό.

Ένα πρόγραμμα που δεν θα χρησιμοποιηθεί τώρα αλλά, παρ' όλα αυτά, είναι συλλέκτης δεδομένων είναι το newsgroup. Κατεβάζει την έτοιμη συλλογή 20 newsgroups και την αποθηκεύει με την σωστή δομή.

2.2. Προεπεξεργασία κειμένου

Στόχος: Η προετοιμασία της συλλογής δεδομένων για αναζήτηση.

Μπορεί στο βήμα 1 να διαχωρίσαμε τίτλο από κείμενο, αλλά το κείμενο είναι ακόμα σε μορφή για HTML – δηλαδή έχει ειδικούς χαρακτήρες που μεταφράζονται από τον φυλλομετρητή (/n, \u2013 κ.α.). Πέρα από αυτό, όμως, το κείμενο είναι γραμμένο για να το διαβάσει ένας άνθρωπος και όχι για να το αναζητήσει μια μηχανή. Πρέπει να πραγματοποιηθούν πέντε αλλαγές στο κείμενο για να ετοιμαστεί:

- **Tokenization**: Το «σπάσιμο» του κειμένου σε διακριτές λέξεις ή «tokens».
- **Lowercasing**: Η αντικατάσταση των κεφαλαίων γραμμάτων με μικρά.
- **Stemming/Lemmatization**: Η ομαδοποίηση παράγωγων λέξεων. Για παράδειγμα, οι λέξεις "dancing", "dancer", "dances" όλες αντικαθίστανται με το "danc".
- **Stop Word Removal**: Η αφαίρεση λέξεων που δεν έχει νόημα να αναζητηθούν λόγω της κοινοτυπίας τους, όπως "me", "the", "on".
- **Special Character Removal**: Η αφαίρεση των ειδικών χαρακτήρων, όπως αναφέραμε.

Το πρόγραμμα processing_cleanup αναλαμβάνει αυτήν την δουλειά, με έτοιμες συναρτήσεις από βιβλιοθήκες.

Σημαντική είναι η σειρά με την οποία εκτελούμε τις παραπάνω αλλαγές. Κάθε βήμα προϋποθέτει σε κάποιο βαθμό ότι όλα τα προηγούμενα έχουν εκτελεστεί επιτυχώς.

2.3. Ευρετήριο

Στόχος: Η δημιουργία ενός αντεστραμμένου ευρετηρίου.

Τώρα έχουμε μια λίστα με άρθρα και τις λέξεις μέσα σε αυτά. Για να κάνουμε την αναζήτηση πιο αποδοτική, θα δημιουργήσουμε ένα αντεστραμμένο ευρετήριο – δηλαδή αντί να έχουμε σε μια λίστα κάθε άρθρο και τις λέξεις που περιέχει, έχουμε λέξεις και τα άρθρα στα οποία βρίσκονται.

Το πρόγραμμα inverted_index είναι αρκετά απλό: για κάθε λέξη, «σκανάρει» το json για τα άρθρα που την περιέχουν. Εάν ένα άρθρο την έχει πολλαπλές φορές, το όνομά του θα αποθηκευτεί επίσης πολλαπλές φορές.

2.4. Μηχανή αναζήτησης

Στόχος: Η υλοποίηση μιας μηχανής αναζήτησης με πολλαπλούς αλγορίθμους.

Αφού έχουμε επεξεργαστεί τα δεδομένα, μπορούμε να υλοποιήσουμε μια μηχανή αναζήτησης. Για τις ανάγκες τις εργασίας έχουμε υλοποιήσει τρείς αλγορίθμους: Boolean με κατάταξη TF-IDF, Okapi BM25, και Vector Space Model (VSM).

Μια παρατήρηση στην αναζήτηση είναι ότι λόγω του stemming που έχει γίνει, εάν ο χρήστης ψάξει λ.χ. το “dance” δεν θα βρεθεί τίποτα γιατί έχει μετατραπεί σε danc. Μια λύση, που δεν έχει υλοποιηθεί για τεχνικούς λόγους, είναι να πραγματοποιούμε την ίδια διαδικασία πρίν στείλουμε το query στην αναζήτηση.

2.4.1. Boolean με κατάταξη TF-IDF

Πριν ασχοληθούμε με την μηχανή αναζήτησης, πρέπει να υλοποιήσουμε τον αλγόριθμο κατάταξης TF-IDF (Term Frequency – Inverse Document Frequency). Για να αυξήσουμε αποδοτικότητα, δεδομένου ότι οι τιμές του TF-IDF παραμένουν σταθερές, αποφασίστηκε να προ-υπολογίσουμε αυτές τις τιμές και να τις φορτώσουμε από ένα αρχείο JSON. Το πρόγραμμα tf-idf κάνει ακριβώς αυτό.

Έχοντας κάνει την προεργασία, η Μπουλιανή αναζήτηση είναι σχετικά απλή. Μπορούμε να αναζητήσουμε ένα όρο μόνο του, ή να κάνουμε λογικές πράξεις AND, OR, και NOT. Όταν οι πράξεις ολοκληρωθούν (δηλαδή, έχουμε το σετ των άρθρων που πληρούν τις προϋποθέσεις), η συνάρτηση search_tfidf τοποθετεί το σκορ της κάθε λέξεις στον όρο. Τέλος, παίρνουμε το αποτέλεσμα.

Η Μπουλιανή αναζήτηση είναι μακράν η πιο γρήγορη, αφού ο φόρτος της εργασίας της έχει προ-υλοποιηθεί.

2.4.2 Okapi BM25

Χρησιμοποιούμε την αρχική συλλογή άρθρων (articles.json) για να συνδέσουμε τα αποτελέσματα της αναζήτησης με τους αρχικούς τίτλους. Όμως, το dataframe που δημιουργούνταν ήταν πολύ μεγάλο, με αποτέλεσμα κάποιες πράξεις να είναι αδύνατες και οι άλλες πολύ πιο αργές. Η λύση σε αυτό το πρόβλημα είναι η συνάρτηση split_and_search, που «σπάει» το dataframe σε n κομμάτια ίσου

μεγέθους. Αυτό κάνει την αναζήτηση bm25 πιο ανακριβή, αλλά δεν βρέθηκε άλλη λύση.

Για την αναζήτηση καθαυτή, χρησιμοποιούμε vectors για να υπολογίσουμε το σκορ bm25.

2.4.3 Vector Space Model

Και εδώ χρησιμοποιούμε το split_and_search. Η συνάρτηση cosine_similarity από το πακέτο sklearn επιτάχυνε κατά πολύ την αναζήτηση. Καθώς δεν υπάρχει οπτικοποίηση των vectors, τα αποτελέσματα είναι σε μορφή σκορ.

2.5 Αξιολόγηση

Μια μηχανή αναζήτησης αξιολογείται με βάση την ταχύτητα και ακρίβεια της, δηλαδή πόσο γρήγορα παρουσιάζει αποτελέσματα και σε αυτά τα αποτελέσματα, πόσα είναι false positives και false negatives.

Για να πετύχουμε αυτό το στόχο, θα χρησιμοποιήσουμε ένα ακόμα dataset, το CISI (**C**entre for **I**nventions and **S**cientific **I**nformation), μαζί με τις βιβλιοθήκες του Scikit για να αξιολογήσουμε το σύστημα. Το CISI αποτελείται από τρία αρχεία: το CISI.ALL, που είναι η βάση δεδομένων που αναζητούμε, το CISI.QRY, που περιέχει κάποιες έτοιμες αναζητήσεις, και το CISI.REL, που έχει τις «σωστές απαντήσεις». Στόχος είναι να φτάσουμε όσο πιο κοντά γίνεται στις τιμές του CISI.REL.

Δυστυχώς, προέκυψε ένα πρόβλημα ασυμφωνίας format μεταξύ του CISI.ALL και του υπάρχοντος συστήματος. Επειδή χρειαζόμαστε το CISI.ALL να έχει μια συγκεκριμένη δομή πριν περάσει στην προ επεξεργασία, και αυτή η δομή αναπόφευκτα θα διαγράψει στοιχεία του CISI.ALL, όπως το όνομα του συγγραφέα και τις αναφορές σε άλλα κείμενα, δεν μπορούμε να το χρησιμοποιήσουμε τελικά ως εργαλείο αξιολόγησης.

Αυτό σημαίνει ότι δεν μπορέσαμε να αξιολογήσουμε το σύστημά μας. Το evaluate είναι ένα καθαρά θεωρητικό πρόγραμμα, που δείχνει πως θα υλοποιούσαμε την αξιολόγηση εάν δούλευε. Σημειώνεται ότι το πρόγραμμα προϋποθέτει τα τρία αρχεία που αναφέρθηκαν.

3. Προσωπικά σχόλια και παρατηρήσεις

Παρόλο που η αξιολόγηση δεν δούλεψε, μπορούμε να εξάγουμε κάποια συμπεράσματα από μόνοι μας.

Προφανώς, όσο μεγαλύτερη η βάση δεδομένων, τόσο πιο αργή και ανακριβής η αναζήτηση.

Σχετικά με την ταχύτητα, μετρήσαμε τα εξής:

Λήψη newsgroup: Αμελητέα (<1 δευτερόλεπτο)

Προεπεξεργασία: 47 δευτερόλεπτα

Index: 2 δευτερόλεπτα

Tf-idf: 6 δευτερόλεπτα

Σύνολο προετοιμασίας: περίπου 56 δευτερόλεπτα.

Αναζήτηση με Boolean: περίπου 2 δευτερόλεπτα

BM25: 8 δευτερόλεπτα

VSM: 4 δευτερόλεπτα

Για τους λόγους που εξηγήσαμε και πιο πάνω, η Μπουλιανή αναζήτηση είναι η πιο γρήγορη. Αξίζει να σημειωθεί ότι η πρώτη εκδοχή του BM25 έκανε πάνω από 2 λεπτά να ολοκληρωθεί, καθώς είχε πολλές επαναλαμβανόμενες πράξεις.

Όσο αναφορά την ακρίβεια, γνωρίζουμε ότι λόγω του split_and_search, η αναζήτηση των BM25 και VSM θα έχει σίγουρα κάποια ανακρίβεια. Προσπαθήσαμε να σπάσουμε την βάση δεδομένων έτσι ώστε να ελαχιστοποιηθεί αυτή η ανακρίβεια.