Ονοματεπώνυμα ΑΕΜ:

Σκαπέρδας Ευστράτιος 2919

Τρεμόπουλος Αλέξανδρος 2894

Υλοποίηση **Crawler**:

Εκτελούμε τον crawler σαν ξεχωριστεί process στο background μαζεύοντας κείμενο από τις ιστοσελίδες. Για τον crawler δίνουμε διαδοχικά 3 παραμέτρους. Η πρώτη είναι το site που θα ξεκινήσει το crawling η δεύτερη είναι πόσες σελίδες θα κάνει crawl και η 3η είναι πόσα threads θα χρησιμοποιηθούν. Ο crawler μας αποτελείται από 3 συναρτήσεις. Η μια είναι η find_links η οποία κάνει request για να πάρει το κείμενο μιας ιστοσελίδας. Η άλλη είναι η find handler που χειριζόμαστε τα έγγραφα που έχουν γίνει crawled η αυτές που είναι σε αναμονή. Ο crawler.py είναι αυτός που κάνει όλη την διαδικασία δίνει εντολή να μαζευτούνε links,διαβάζει τα κείμενα που περιέχουνε και στέλνει το κείμενο στον Index.Επίσης καλεί την συνάρτηση csv_handler η οποία είναι αυτή που φτιάχνει ένα csv αρχείο που αποθηκεύεται το λεξικό μας. Η διαδικασία crawling γίνεται με έναν υβριδικό αλγόριθμο που χρησιμοποιεί συνδυασμό bfs και dfs και διαβάζει τις σελίδες από το αρχείο queue.txt.Οι σελίδες που γίνονται crawl αποθηκεύονται στο αρχείο crawled.txt και σβήνονται από το queue.txt. Θα αναλύσουμε αυτά τα αρχεία:

file_handler.py: Χρησιμοποιούμε την βιβλιοθήκη os.Αυτό είναι το αρχείο το οποίο διαχειρίζεται folders και txt.Περιέχει τις εξής συναρτήσεις:

1η) create_proj_directory(directory): Εδώ δημιουργούμε ένα folder που θα περιέχει τα αρχεία queue.txt και crawled.txt.

2η) create_files(project,first_url): Εδώ δημιουργούνται τα δυο txt που προαναφέρθηκαν. Στο queue.txt γράφουμε το πρώτο link που δώσαμε.

3η) write_to_file(path,data): Δημιουργεί ένα καινούριο φάκελο και γράφει πληροφορίες.

4η) delete_file_inside(path): Διαγράφει το περιεχόμενο ενός φακέλου.

5η) file_to_set(file_name): Διαβάζουμε έναν φάκελο και μετατρέπουμε κάθε γραμμή του σε set items.

6η) set_to_file(links,file_name): Γράφουμε κάθε item ενός set σε μια γραμμή ενός φακέλου.

find_links.py:

Χρησιμοποιούμε από την βιβλιοθήκη html.parser την HTMLParser και από την urllib την parse.Έχουμε την κλάση **FindLinks(HTMLParser).**Ανοίγουμε ένα url και

ψάχνουμε για υπόλοιπα urls κάτω από κάποιες προϋποθέσεις και τα αποθηκεύουμε σε ένα set.Η κλάση περιέχει τις εξής συναρτήσεις:

1η) handle_starttag(self,tag,attrs): Αυτή η συνάρτηση ανήκει στην HTMLParser και την κάνουμε overwrite.Όταν καλείται η συνάρτηση feed() της HTMLParser και συναντήσουμε το tag <a> καλείται αυτή η συνάρτηση. Αν κληθεί και το tag είναι <a> ψάχνουμε αν το attribute είναι href. Το href καθορίζει τον προορισμό του link. Αν είναι href τότε φτιάχνουμε ένα absolute url και το προσθέτουμε σε ένα set με links.

2η) get_links(self): Επιστρέφει απλά τα links που βρέθηκαν.

3η) error(self,message): Όταν βρει ένα error απλά το προσπερνάει.

crawler.py:

Εδώ έχουμε την κλάση Crawler. Παίρνουμε την urllib.request και πιο συγκεκριμένα την urlopen. Επίσης από την bs4 χρησιμοποιούμε την BeautifulSoup. Όλες οι συναρτήσεις εκτός από τον κατασκευαστή είναι static. Οι συναρτήσεις είναι οι εξής:

1η) __init__(self,project,base_url): Ξεκινάει με το όνομα του project και το homepage. Δημιουργεί τα αρχεία queue.txt και crawled.txt. Καλεί τις συναρτήσεις startup() και crawling της κλάσης.

2η) start_up(): Αφού δημιουργηθούνε τα αρχεία του project ξεκινάει ο crawler.

3η) crawling(self,page_url): Διαβάζει ένα url το οποίο το δίνει σαν όρισμα στην collect_links και όλο αυτό στην add_links(θα εξηγηθούνε). Το url διαγράφεται από queue και προστίθεται στα crawl.

4η) collect_links(self,page_url): Χρησιμοποιεί το **find_links.py** αρχείο. Με την βιβλιοθήκη urlopen ζητάει άδεια να διαβάσει την html ενός website και στέλνει το url στην save_text η οποία θα κρατήσει το κείμενο. Αν δεν μπορέσει να ανοίξει το url εκτυπώνουμε το error που βρήκε. Τέλος επιστρέφει τα υπόλοιπα links που βρήκε με την βοήθεια της get_links() από την **FindsLinks().**

5η) save_text(page_url): Χρησιμοποιώντας τις βιβλιοθήκες που αναφέρθηκαν διαβάζει το text κάνει κάποια προεπεξεργασία πριν το δώσει σαν όρισμα στον index μαζί με το url. write_a_csv(Index(text, page_url)) σε αυτήν την γραμμή καλείται η συνάρτηση write_a_csv από τον csv_handler με όρισμα ένα αντικείμενο του Index ο οποίος έχει σαν όρισμα το επεξεργασμένο text και την σελίδα που βρέθηκε. Ουσιαστικά κάνουμε overwrite το λεξικό μας έχοντας ανανεώσει τον inverted index μας. Θα αναλυθεί πιο διεξοδικά.

6η) add_links(links): Ότι link βρίσκεται σε ένα άλλο link και δεν έχει ήδη βρεθεί η γίνει crawled το προσθέτουμε στο queue μας.

7η) update_files(): Χρησιμοποιεί την συνάρτηση set_to_file() από την **file_handler.py**.

Υλοποίηση **Indexer**:

Εδώ δημιουργούμε και επεξεργαζόμαστε τον ανεστραμμένο κατάλογο. Διαβάζοντας το text κάθε σελίδας το ανανεώνουμε. Όπως προαναφέρθηκε η κλάση **Index** στο **Index.py** καλείται από τον **crawler.py.** Ο Indexer θα χρησιμοποιήσει 2 αρχεία. Τα **Index.py** και **csv_handler.py.**

Index.py:

Εδώ έχουμε την κλάση Index.Χρησιμοποιούμε τις βιβλιοθήκες re,string και από τις nltk.corpus, nltk.stem, nltk.tokenize τις stopwords, WordNetLemmatizer και word_tokenize αντίστοιχα. Έχουμε την μεταβλητή the_dict στην οποία αποθηκεύουμε τον συνολικό inverted index ο οποίος ανανεώνεται για κάθε νέα σελίδα που έρχεται. Όλες οι συναρτήσεις εκτός από τον κατασκευαστή είναι static Οι συναρτήσεις είναι οι εξής:

1η) __init__(self,text,url): Στον κατασκευαστή καλείται η συνάρτηση preprocess για το συγκεκριμένο αντικείμενο και η the_dictionary.

2η) __iter__(self): Αυτό μας επιτρέπει να κάνουμε iterate το κείμενο.

3η) preprocess(text): Εδώ επεξεργαζόμαστε το κείμενο κάθε σελίδας διαγράφοντας σύμβολα, stopwords, νούμερα και σημεία στίξης. Κρατάμε μόνο λατινικούς χαρακτήρες και μετατρέπουμε τα γράμματα σε μικρά. Επίσης μετατρέπουμε την κάθε λέξη στην ριζά της. Τέλος ταξινομούμε αλφαβητικά και κρατάμε σε μια μεταβλητή το συνολικό μέγεθος του κειμένου. Για να γίνουν όλα αυτά χρησιμοποιούνται όλες οι βιβλιοθήκες που προαναφέρθηκαν. Επιστρέφει το επεξεργασμένο κείμενο.

4η) docs_dictionary(text): Εδώ φτιάχνεται ένα ‘μικρό’ λεξικό το οποίο αποθηκεύει τις πληροφορίες του κειμένου για κάθε σελίδα που έρχεται. Δηλαδή τις λέξεις και την συχνότητα τους σε αυτήν την σελίδα.

5η) the_dictionary(text,page_url): Εδώ ανανεώνεται για κάθε σελίδα ο συνολικός inverted index. Γίνεται κα η κλήση της docs_dictionary η οποία μας επιστρέφει τις πληροφορίες που βρήκε. Για κάθε λέξη του συγκεκριμένου url ελέγχουμε αν υπάρχει στο dictionary ήδη. Αν δεν υπάρχει προσθέτουμε στον inverted_index μια νέα γραμμή με την λέξη η οποία περιέχει επίσης τη συχνότητα της στο url, το url, και τις συνολικές λέξεις που περιέχει αυτό το url. Αυτές οι τρεις πληροφορίες είναι μια λίστα. Αν υπάρχει ήδη λέξη στον inverted index τότε αυτά τα τρία στοιχεία τα προσθέτουμε σε μια λίστα η οποία περιέχει τόσες υπολίστες όσες είναι οι φορές που έχει βρεθεί η λέξη σε διαφορετικά site.

csv_handler.py:

Χρησιμοποιούνται οι βιβλιοθήκες pandas και os. Αποθηκεύουμε τον inverted index σε ένα csv αρχείο. Αυτό υλοποιείται εδώ. Οι συναρτήσεις είναι οι εξής:

1η) write_a_csv(obj): Εδώ δημιουργούμε το csv αρχείο αν δεν υπάρχει. Αν υπάρχει κάνουμε overwrite τον inverted index. Έτσι κάθε φορά που κάνουμε crawl μια νέα σελίδα ανανεώνεται και το csv αρχείο. Για την υλοποίηση χρησιμοποιούμε την βιβλιοθήκη pandas.

2η) read_a_csv(csv_name): Διαβάζουμε ένα csv.

3η) delete_a_csv(csv_name): Διαγράφουμε ένα csv.

Κλήση **Crawler** και **Indexer**:

Crawler_Indexer_run.py: Χρησιμοποιούνται οι βιβλιοθήκες threading, Queue και shutil. Εδώ είναι η συνάρτηση main η οποία ξεκινά τον crawler. Αρχικά ζητάμε από τον χρήστη να δώσει το homepage, τον αριθμό των σελίδων που θέλει να γίνουν crawled, και τον αριθμό των thread που θα χρησιμοποιηθούν. Δυστυχώς με τον τρόπο που σώζουμε τον inverted index(csv) δεν καταφέρνουμε να διατηρούμε τον inverted index από το προηγούμενο crawl. Οπότε αν ξανακάνουμε crawl από το μηδέν διαγράφουμε τον inverted index και το φτιάχνουμε από την αρχή. Τρέχουμε τον κατασκευαστή του crawler και μετά την main. Η main υλοποιείται με multithreading.Γίνεται πρώτα η κλήσης της create_threads και μετά της crawl. Έχει τις εξής 4 συναρτήσεις:

1η) create_threads(): Υλοποιούμε τα threads που έχει δώσει ο χρήστης που θα υπάρχουν όσο τρέχει η main. Αρχίζει η διαδικασία του multithreading καλώντας την συνάρτηση nextjob().

2η) next_job(): Υλοποιεί τη νέα διαδικασία που είναι σε αναμονή. Διαβάζει ένα url από το queue και καλεί την συνάρτηση crawling της κλάσης crawler περνώντας αυτό το url σαν όρισμα.

3η) new_jobs(): Η new_jobs με την crawl αλληλοκαλούνται και διαβάζει ένα τα url του queue ώστε να γίνουν επόμενες ‘εργασίες’.

4η) crawl(): Καλεί την new_jobs όσο υπάρχουν σελίδες στο queue.

Υλοποίηση **Query Processor**:

Ο query processor συνδέεται και με τον server μας(localhost) οπού ο χρήστης εκεί πέρα δίνει το query που θέλει να αναζητήσει και το ποσά top-k έγγραφα θέλει να επιστραφούν. Ανάλογα με το query του χρήστη υπολογίζουμε την ομοιότητα των εγγράφων με το query χρησιμοποιώντας tf-idf και cosine similarity και του εμφανίζουμε στον localhost τα top-k έγγραφα. Στον query processor χρησιμοποιούνται τα αρχεία **server.py, Similarity.py, QueryRun.py** και **index.html**.

index.html:

Εδώ φτιάχνουμε το interface του localhost μας. Δεν θα επεκταθούμε πολύ μιας και δεν είναι το θέμα της εργασίας. Μέσα στο αρχείο αυτό φτιάχνουμε 3 input. Ένα για το query του χρήστη, ένα για τα top-k, και ένα κουμπί για να γίνει το search. Γίνεται χρήση της γλώσσας css ώστε να διαμορφώσουμε το localhost βάζοντας χρώματα, σκιές, σχήματα, και για να κεντράρουμε το output.

server.py:

Εδώ φτιάχνουμε έναν απλό http server. Χρησιμοποιούνται οι εξής βιβλιοθήκες: Η urllib, η logging, από την http.server οι SimpleHTTPRequestHandler και HTTPServer, και από την bs4 η BeautifulSoup. Έχουμε μια κλάση S που δέχεται το SimpleHTTPRequestHandler. Έχουμε τις εξής συναρτήσεις:

1η) _set_response(self): Θέτει το response 200 και στέλνει ένα header.

2η) do_GET(self): Ο τρόπος που διαβάζουμε τι έχει δώσει ο χρήστης είναι από το url. Δηλαδή παίρνουμε το path και ανακτούμε τις πληροφορίες. Κάνοντας split και replace διαβάζουμε το κομμάτι που θέλουμε από το path και έπειτα απομονώνουμε το query και το top-k. Αν ο χρηστής δεν βάλει ποσά έγγραφα θέλουν να επιστραφούνε επιστρέφουμε μέχρι 10 αποτελέσματα. Εδώ θα γίνει κλήση των αρχείων QueryRun και Similarity. Κάνουμε preprocess μέσω της QueryRun το query του χρήστη και υπολογίζουμε το tf του query. Έπειτα φτιάχνουμε ένα αντικείμενο της κλάσης Similarity στο οποίο δίνουμε το query του χρήστη το tf του και το top-k.Αυτό θα μας επιστρέψει τα αποτελέσματά-urls. Για κάθε url διαβάζουμε τον τίτλο του και το εκτυπώνουμε μαζί με το url στον localhost. Υπάρχει και η συνάρτηση run(server_class=HTTPServer, handler_class=S,port=8080) η οποία είναι εκτός κλάσης. Σε αυτήν την συνάρτηση δίνουμε ένα port για να τρέξουμε τον http server και την συνδέουμε με την κλάση S που αναφέρθηκε πάνω. Όπως λέει και η εκφώνηση ο Crawler-Indexer είναι διαφορετική διαδικασία από τον QueryProcessor. Για αυτό και ο **server.py** έχει μια ξεχωριστεί main.Σε αυτήν την main χρησιμοποιούμε την βιβλιοθήκη sys και καλούμε την συνάρτηση run για να υλοποιηθεί ο server.

QueryRun.py

Σε αυτό το αρχείο γίνεται preprocess στο query του χρήστη και υπολογίζεται το TF, για αυτό το query. Όπως και στο αρχείο **Index.py**, χρησιμοποιούνται για το preprocess οι εξής βιβλιοθήκες : re,string και από τις nltk.corpus, nltk.stem, nltk.tokenize τις stopwords, WordNetLemmatizer και word_tokenize αντίστοιχα. Στο αρχείο υπάρχει μία global μεταβλητή, η users_tf_list που είναι μία λίστα με το tf για κάθε λέξη του χρήστη. Χρησιμοποιούμε την κλάση **QueryRun,** όπου έχουμε τις εξής συναρτήσεις(Όλες οι συναρτήσεις εκτός από τον κατασκευαστή είναι static):

1η) preprocess(text): Εδώ επεξεργαζόμαστε το κείμενο κάθε σελίδας διαγράφοντας σύμβολα, stopwords, νούμερα και σημεία στίξης. Κρατάμε μόνο λατινικούς χαρακτήρες και μετατρέπουμε τα γράμματα σε μικρά. Επίσης μετατρέπουμε την κάθε λέξη στην ριζά της. Τέλος ταξινομούμε αλφαβητικά και κρατάμε σε μια

μεταβλητή το συνολικό μέγεθος του κειμένου. Για να γίνουν όλα αυτά χρησιμοποιούνται όλες οι βιβλιοθήκες που προαναφέρθηκαν. Επιστρέφει το επεξεργασμένο κείμενο. Επίσης τις λέξεις πριν τις βάλουμε σε λίστα, τις βάζουμε σε set, έτσι ώστε να αφαιρεθούν τα duplicates.

2η) users_tf(size_of_search): Εδώ υπολογίζουμε το tf για κάθε επεξεργασμένη λέξη του query του χρήστη αφού γίνουν preprocessed. Το tf κάθε λέξης ισούται με 1/ size_of_search καθώς δεν υπάρχουν duplicates.

Similarity.py

Εδώ υπολογίζουμε την ομοιότητα του query του χρήστη με τα έγγραφα που περιέχουν έστω μία λέξη από αυτές που αναζήτησε ο χρήστης. Για τα έγγραφα αυτά υπολογίζουμε το cosine similarity, με τα βάρη να ισούνται με TFxIDF και επιστρέφουμε τις topK ιστοσελίδες. Το αρχείο περιέχει την κλάση Similarity και χρησιμοποιούνται οι βιβλιοθήκες pandas, csv και math. Αυτή η κλάση χρησιμοποιείται στον server.py όπου φτιάχνεται αντικείμενο της κλάσης για κάθε query του χρήστη. Η κλάση χρησιμοποιεί 5 μεταβλητές, οι οποίες είναι οι εξής :
- idf :μία λίστα που κρατάει το idf για κάθε URL που περιέχει έστω μία λέξη από το query

-tf: μία λίστα που κρατάει το tf για κάθε URL που περιέχει έστω μία λέξη από το query

-urls: μία λίστα που κρατάει όλα τα URL που περιέχουν έστω μία λέξη από το query

-weight_of_url : μία λίστα που κρατάει τα βάρη (TFxIDF) για κάθε URL που περιέχει έστω μία λέξη από το query

-results_urls: Τα URL που θα επιστραφούν στο τέλος στον χρήστη.

Όλες οι συναρτήσεις εκτός από τον κατασκευαστή είναι static .Οι συναρτήσεις της είναι οι εξής:

1η) __init__(self, search, query_tf, topK): Εδώ υπάρχουν 3 ορίσματα. Το search είναι οι λέξεις του query που αναζήτησε ο χρήστης αφού όμως προεπεξεργαστούν στην κλάση QueryRun. Το query_tf είναι το tf του query το οποίο και αυτό υπολογίζεται στην QueryRun. Τέλος διαβάζει τα topK έγγραφα που θέλει ο χρήστης να επιστραφούν. Για κάθε query οι λίστες idf , tf, urls , weight_of_url γίνονται clear, ώστε να υπολογιστούν από την αρχή.

2η) copy_index(): Αντιγράφουμε το csv αρχείο μας με τον inverted index σε ένα νέο csv, έτσι ώστε αν εκείνη τη στιγμή που γίνεται crawling αναζητηθεί ένα query, να έχουμε τον προηγούμενο inverted index, ώσπου το crawling να τελειώσει.

3η) tf_idf(list_search): Εδώ υπολογίζεται το βάρος για όλες τις σελίδες που περιέχουν έστω μία από τις λέξεις του query. Διαβάζουμε τα περιεχόμενα του αντιγραμμένου inverted index. Στην αρχή, αρχικοποιούμε τις λίστες που θα

χρησιμοποιηθούν. Επειδή μία λέξη μπορεί να υπάρχει σε πολλά URLS, οι λίστες tf, urls , weight_of_url αποτελούνται από υπολίστες σε αριθμό όσες και οι λέξεις του query , όπου η κάθε υπολίστα περιέχει τα αποτελέσματα για μία λέξη. Γίνεται μια διπλή for ,η οποία για κάθε γραμμή στον index, ελέγχει αν κάθε λέξη του query υπάρχει εκεί. Για κάθε λέξη του query που υπάρχει στον index κρατάμε όλες τις πληροφορίες, δηλαδή, τη συχνότητα της λέξης, το URL που βρέθηκε και το πλήθος των distinct λέξεων του URL. Αυτές οι πληροφορίες κρατούνται για όλα τα URLS που βρέθηκε η λέξη. Έπειτα υπολογίζουμε πόσα URL περιέχουν την λέξη και διαβάζουμε από το αρχείο "numOfSites.txt" που φτιάχνεται στον **Crawler_Indexer_run.py** πόσες σελίδες κάναμε crawl. Στη συνέχει υπολογίζεται το idf της λέξης με τον εξής

$$idf(w) = log(\frac{N}{df_t})$$

τύπο:

, όπου N είναι το πλήθος όλων των URLS και dft είναι το πλήθος των URLS που περιέχουν τη λέξη. Για να υπολογιστεί το tf χρειάζεται μία ακόμη for για να σκανάρει όλα τα URLS που περιέχουν τη λέξη. Ο τύπος που χρησιμοποιούμε είναι ο

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

εξής : , όπου ο αριθμητής είναι το πόσες φορές εμφανίζεται η λέξη στο URL, και ο παρονομαστής είναι το πλήθος των λέξεων που περιέχει το URL. Τέλος αποθηκεύουμε στη λίστα urls , το url ακριβώς στις ίδες θέσεις της λίστας που είναι και τα tf/idf. Δηλαδή στη 0 θέση κάθε λίστας από τις τρείς υπάρχουν οι πληροφορίες για την ίδια λέξη.

4η) <u>cosine_similarity(query_tf, top_k):</u> Φτιάχνουμε μία λίστα, την distinct_urls, που περιέχει κάθε URL, από μία μόνο φορά. Έπειτα σε μία διπλή for υπολογίζουμε το βάρος W=TFxIDF κάθε URL. Στη συνέχεια αν το βάρος W δεν είναι 0 το προσθέτουμε στη λίστα weight_of_url και στην ίδια θέση προσθέτουμε το URL στην distinct_urls. Μετά υπολογίζουμε το cosine similarity κάθε URL(distinct_urls) με το query. Χρησιμοποιούμε τον εξής τύπο :

$$S_{cosine}(q,d) = \cos(\theta) = \frac{\vec{q} \bullet \vec{d}}{|\vec{q}| \cdot |\vec{d}|} = \frac{\sum_{i=1}^{M} w_{t_i,q} \cdot w_{t_i,d}}{\sqrt{\sum_{i=1}^{M} w_{t_i,q}^2} \cdot \sqrt{\sum_{i=1}^{M} w_{t_i,d}^2}}$$

. Ο αριθμητής είναι το άθροισμα για όλες τις σελίδες, του γινομένου του βάρους της κάθε σελίδας με το βάρος του query, το οποίο είναι το tf του, αφού το idf είναι 1. Ο παρονομαστής είναι το γινόμενο των 2 μέτρων, των βαρών των URL και των βαρών του query.

Τώρα αφού έχουμε υπολογίσει την ομοιότητα κάθε σελίδας με το query υπολογίζουμε τα topk URLS που θα επιστραφούν. Υπολογίζουμε το max cosine similarity, το αποθηκεύουμε πρώτο σε μία λίστα, το διαγράφουμε από τη λίστα με τα cosines και από τα distinct urls και συνεχίζουμε τη λούπα, μέχρι να

συμπληρώσουμε τον αριθμό των topK ή μέχρι να μην έχουμε άλλη σελίδα για επιστροφή. Επιστρέφουμε στον server.py τα αποτελέσματα.