گزارش فاز اول پروژه بازیابی اطلاعات

بخش ۱. پیش‌پردازش اولیه

از بخش‌های ابتدایی هر پروژه‌ی بازیابی پیش پردازش متنی متون اولیه است . در این بخش با برخی از توابع به سادهسازی و یکسان سازی متون می پردازیم. با توجه به اینکه دو نوع داده‌ی اولیه (فارسی و انگلیسی) داریم از دو کتابخانه Hazm برای متون فارسی و از nltk برای متون انگلیسی استفاده شده است.
برای ساده سازی عملیات‌ها ابتدا برای بخش فارسی فایل xml داده شده را با یک تابع به صورت فایل csv تبدیل نموده که فرمت دو داده یکسان شده و استخراج داده‌های لازم از آنها راحت تر باشد.
دو تابع اصلی این بخش تابع pre_proccess و most_freq_words می‌باشد که طبق خواسته داک متن پیش پردازش شده و کلمات پر تکرار را نشان میدهد. تابع pre_proccess رشته‌ی متنی را ورودی گرفته و داخل آن با فراخوانی تابع prepare_text متن نرمال سازی شده را برگردانده می‌شود . عملیات‌هایی که در این تابع اصلی صورت می‌گیرند :
- عملیات *tokenize*:
به این صورت که با استفاده از تابع‌های اماده‌ی کتابخانه nltk و hazm متن به صورت کلمه به کلمه توکن می‌شود.
- عملیات *normalization*:
در این عملیات برای داده‌ی Ted Talk ، ابتدا تمامی حروف را lower case کرده سپس از لیست آماده stopwords استفاده شده است تا کلمات پر تکرار حذف شوند اما برخی از متنها شامل این کلمات بودند کاملا حذف می‌شدند ، لذا لیستی از کلمات پرتکرار ایجاد کرده و ۳۰ کلمه‌ی پر تکرار را از کل متن حذف شده است برای حذف علائم نگارشی نیز از تابع آماده‌ی isalpha() کمک گرفته شد.
برای داده‌ی WIKI نیز آرایه‌ی punctuations حذف شدند و با تابع آماده‌ی normalize متن نرمال سازی شد.
-عملیات *stemming \ lemmatization*:
در این عملیات از lemmatize دو کتابخانه ذکر شده استفاده شده است ، lemmatize کردن کلمات و افعال در متون انگلیسی بر اساس verb یا noun بودن صورت میگیرد و در فارسی بن فعل‌ها را باز می‌گرداند. stemming کردن متون فارسی موجب تغییر مفهوم کلمات می‌شد.

کلاس tokenizer در hazm قابلیت جایگزین سازی برخی دیگر از علائم مانند ایموجی‌ها ، هشتگ، لینک و ... را نیز دارد که اگر در متنی باشد با " . " جایگزین شده و در تابع prepare_text به space تغییر یافته است.

در نهایت داده ی پیش پردازش شده به صورت یک فایل csv در محل پروژه جهت استفاده در سایر بخشها ذخیر شد.

برای شمارش تعداد کلمات پرتکرار نیز از تابع FreqDist (کتابخانه nltk) در هر دو نوع متن استفاده شده است. این تابع تکرار token ها را شمارش می‌کند.
برای پیاده سازی مراحل پیش پردازش ذکر شده از این لینک برای متون کمک گرفته شده است.

بخش۳. فشردهسازی

در حوزه بازیابی اطلاعات به‌طور معمول با داده‌های زیادی روبرو هستیم که بخشی از این داده‌ها در حافظه کامپیوتر قرار می‌گیرد که با توجه به محدودیت حافظه‌های کامپیوتر, تنها قادر هستیم بخش کوچکی از داده‌ها را در آن قرار دهیم. فشرده سازی یکی از روش‌هایی است که بخش اعظمی از این معضل را حل می‌کند و قادر می‌سازد بتوانیم با داده‌های بیشتری سروکار داشته باشیم بدون داشتن نگرانی بابت محدودیت حافظه یا دیسک . ازین رو در این قسمت از دو نوع روش فشرده سازی به نام‌های variable byte و gamma code استفاده شده است که می‌توان نحوه‌ی پیاده سازی آنها را در فایل compress.py, به ترتیب در تابع‌های variable_byte_encoder و gamma_code_encoder یافت. هردو روش به‌جای اعداد دهدهی, با بیت‌ها سروکار دارند, بنابراین پس از اندکی تغییر در نمایه‌های ساخته‌شده در قسمت‌های قبل و تبدیل به نمایه فاصله‌ای (لطفا به تابعcreate_distance_index توجه شود), تمامی اعداد موجود در نمایه با استفاده از کتابخانه bitarray به تعدادی از بیت‌های متناظر تبدیل می‌شوند (که این تبدیل نیز در هردو تابع های encoder انجام می‌شود). در نهایت در تابع memory_usage, می‌توان میزان بهینه‌کردن استفاده از حافظه را در دو قالب مشاهده کرد:

۱. بدست‌آوردن حافظه مصرفی نمایه ها به طور مستقیم با استفاده از متد getsizeof(), که هردو چاپ اول این تابع مربوط به این روش می‌باشد.

۲. بدست‌آوردن حافظه مصرفی مربوط به هر term با استفاده از getsizeof() و حساب کردن مجموع آنها, که دو چاپ دوم این‌تابع مربوط به این روش می‌باشد.

اما علت استفاده از روش دوم چیست؟ در روش اول, با توجه به این لینک, گویا دیکشنری در پایتون خود یک روش فشرده سازی و براساس تعداد کلیدهای موجود می‌زند (گزاره آخر براساس شواهد خود تیم ماست), از طرفی طبق توضیحات این لینک, کتابخانه bitarray متاسفانه serializable نیست و در ابتدا بایستی به رشته های کاراکتری تبدیل شود, در نتیجه در روش اول نتایج ناخواستانه‌ای را مشاهده کردیم و برای بهتر نشان‌دادن عملکرد توابع پیادهسازی شده, از روش دوم استفاده شده است.

همچنین با توجه به توضیحات داک پروژه, تنها تابع ناهمفشردهسازی برای روش variable_byte و در تابع variable_byte_decoder پیادهسازی شده است.

بخش۴. اصلاح پرسمان

همانطور که می‌دانید, سیستم‌های بازیابی اطلاعات عملکردشان براساس پرسمان‌هایی که کاربران می‌دهند ارزیابی می‌شوند, از طرفی وجود خطاهایی بین این پرسمان‌ها انکارناپذیر است و سیستم‌های بازیابی بایستی توانایی تشخیص و اصلاح خطاها تا حداکثر ممکن داشته باشند. ازین رو در فایل spelling_correction.py بااستفاده از روش‌های jaccard_distance و مارتیس لونشتاین (که با توجه به توضیحات داک پروژه تنها به پیادهسازی فاصله بین دو کلمه بسنده کرده‌ایم) به دنبال پیادهسازی این ویژگی بودیم. با توجه به توضیحات این لینک و این لینک, دو نوع تابع jaccard_distance پیادهسازی شده است که نام‌های آنها به ترتیب jaccard_dist_type1 و jaccard_dist_type2 می‌باشد.علاوه برآن این روش نمایه های bigram ساخته‌شده در قسمت‌های قبلی را استفاده می‌کند.

بخش۵. جستجو و بازیابی اسناد