بازیابی پیشرفته‌ی اطلاعات

فاز اول پروژه

الهه خدایی ۹۶۱۱۰۲۰۴
نیما فتحی ۹۶۱۰۶۷۴۱
علیرضا توکلی ۹۶۱۰۵۶۳۷

بخش اول.

در این بخش در پیش از هرکاری باید دو سری دیتا را پیش پردازش کرد به گونه‌ای که از فرمت .xml و .csv به یک سری نوشته تبدیل کرده تا بتوان داکیومنت‌های بدست آمده را پیش‌پردازش کرد.

برای داده به زبان انگلیسی از کتابخانه pandas استفاده می‌کنیم. و دو ستون title و description را به عنوان مجموعه داکیومنت به پیش‌پردازنده ها پاس می‌دهیم.

برای داده ویکی‌پدیا به زبان فارسی نیز چون در فرمت .xml است از کتابخانه xml.etree استفاده می‌کنیم. می‌دانیم که این فرمت یک حالت درختی و تگ محور دارد. برای استخراج به شکل عنوان و نوشته دو بخش مختلف از هر سند رو استخراج کرده:

۱- revision.text

۲- title.text

و این دو بخش را برای پیش‌پردازش به پیش‌پردازنده فارسی که با کتابخانه hazm نوشته شده است پاس می‌دهیم.

اما برای خود پیش پردازش دو کلاس جداگانه نوشته شده که از کلاس PreProcess که base محسوب می‌شود ارث‌بری کرده. در این کلاس استفاده‌های خواسته شده نوشته شده است از جمله متودهای

۱- Normalize: در این فانکشن ما مراحل مختلف را اعمال می‌کنیم از جمله توکن کردن کلمات، حذف اعداد، stemming ، حذف علائم نگارشی و ...

۲- tokenization: در این فانکشن نیز به کمک فانکشن‌های آماده در nltk و hazm جملات را به توکن‌های کلمه‌ای می‌شکنیم.

۳- stemming: در این فانکشن نیز مراحل استیمینگ را باز هم با توابع آماده بر روی لیست توکن‌ها انجام می‌دهیم.

۴- remove punctuations: در این بخش علائم نگارشی را حذف کرده (+ اعداد).

۵- find stopwords: در این فانکشن کلمات سکون (stopwords) را پیدا کرده و در کلمات انگلیسی چون لیست آنها را داریم کلماتی که عضو لیست عادی این کلمات هستند و درصد قابل قبولی از تعداد کلمات را دارا می‌باشند را حذف کرده، برای فارسی اما فقط کلمات با تکرار بالا را حذف می‌کنیم(زیرا دیتایی در این مورد در کتابخانه‌های پردازش متن فارسی وجود ندارد)

۶- remove stopwords: کلمات گفته شده را حذف کرده و در لیستی ذخیره می‌کنیم.

۷- plot stopwords: کلماتی که حذف می‌کنیم را در یک barplot با تعداد تکرار آنها نمایش می‌دهیم.

۸- clean documents: داکیومنت‌های استخراج شده در بخش قبل (در فایل infotmation_retrieval_system.py) را به این فانکشن پاس داده و پیش پردازش بر روی آنها انجام می‌شود.

بخش دوم.

در این بخش دو نمایه‌ی bigram و positional پیادسازی شده‌اند که توضیحات آن به شرح زیر می‌باشد.

.Positional

در این بخش به ازای هر لغت لیستی از اسنادی که این لغت را شامل می‌شوند بعلاوه جایگاه آن در سند را داریم. برای پویاسازی این نمایه (به روز بودن پایگاه داده پس از اعمال تغییرات) دستوراتی مانند delete doc و add doc را هم پیادسازی می‌کنیم. دو تابع دیگر هم برای نمایش posting list و position پیادسازی می‌کنیم.

.Bigram

در این روش هم ترکیب‌های دوتایی از تمام لغات را ذخیره می‌کنیم. در این قسمت توابعی نظیر load و save و add token پیادسازی شده‌اند برای مواقعی که خواستیم نمایه‌ها را ذخیره یا بارگذاری کنیم. Delete doc و show bigram هم برای پویا بودن در این قسمت پیادسازی شده‌اند.

بخش سوم.

هدف اصلی این بخش فشردهسازی نمایهها میباشد. فایل compressing indexes بدنهی اصلی این بخش میباشد و نیازمندیها در فایل Compressor پیادهسازی شدهاند. دو روش عمده برای فشردهسازی نمایهها به کار میبریم که به شرح زیر میباشند.

.Gamma code

همانگونه که گفته شد این بخش نوعی فشردهسازی میباشد که کد گاما را پیادهسازی میکند. پیادهسازی کد گاما به همان روش معمول انجام میگیرد. برای مثال تابع encode آن بصورت زیر میباشد.

```
86      @staticmethod
87      def gamma_encode(numbers):
88          gaps = Compressor.calculate_gaps(numbers)
89          code_str = ""
90          for n in gaps:
91              code_str += Compressor.gamma_encode_number(n)
92          return Compressor.bin_to_byte(code_str)
93
94      @staticmethod
95      def gamma_encode_number(number):
96          code = ""
97          for _ in range(floor(log2(number))):
98              code += '1'
99          binary_num = bin(number)[2:]
100         code += "0"
101         for i in range(1, len(binary_num)):
102             code += binary_num[i]
103         return code
```

.Variable byte

برای این قسمت سه تابع اصلی وجود دارد؛ تابع encode و decode و تابعی برای numbers ها. مطابق قسمت گاما، encode کردن این قسمت به صورت زیر است.

```python
    def variable_byte_encode(numbers):

        gaps = Compressor.calculate_gaps(numbers)
        res = ""
        for n in gaps:
            res += Compressor.variable_byte_encode_number(n)
        return Compressor.bin_to_byte(res)

    @staticmethod
    def variable_byte_encode_number(number):
        s = ""
        bytes_list = []
        while True:
            binary_num = bin(number % 128)[2:]
            bytes_list.append('0' * (8 - len(binary_num)) + str(binary_num))
            if number < 128:
                break
            number = number // 128
        low_byte = list(bytes_list[0])
        low_byte[0] = '1'
        bytes_list[0] = "".join(low_byte)

        for i in range(len(bytes_list) - 1, -1, -1):
            s += bytes_list[i]

        return s
```

بخش چهارم.

در این بخش هدف اصلاح کوئری‌های داده شده توسط کاربر می‌باشد. دو عنصر اصلی این بخش توابعی هستند که فاصله‌ی levenshtein و jaccard بر اساس بایگرام‌ها را محاسبه می‌کنند.تابع levenshtein بصورت زیر می‌باشد.

```python
def levenshtein_distance(word1, word2):
    if len(word1) > len(word2):
        word1, word2 = word2, word1

    distances = range(len(word1) + 1)
    for i2, c2 in enumerate(word2):
        distances_ = [i2 + 1]
        for i1, c1 in enumerate(word1):
            if c1 == c2:
                distances_.append(distances[i1])
            else:
                distances_.append(1 + min((distances[i1], distances[i1 + 1], distances_[-1])))
        distances = distances_
    return distances[-1]
```

این تابع دو کلمه را به عنوان ورودی دریافت کرده و edit distance را مطابق الگوریتم فوق به ما می‌دهد.

تابع jaccard similarity هم مشابها دو کلمه دریافت می‌کند و شباهت جاکارد را با استفاده از bigram های این دو ترم به ما می‌دهد. مورد توجه است که مقدار این تابع با زمانی که بدون بایگرام اقدام به محاسبه می‌کنیم، متفاوت خواهد بود. قسمت کامنت شده محاسبه‌ی جاکارد بدون استفاده از بایگرام می‌باشد. مورد توجه است که بایگرام‌ها در یک مجموعه ذخیره می‌شوند و نه لیست.

استفاده از بایگرام دقت کار را تا حدی زیادی کم می‌کند و استفاده از قسمت کامنت شده پاسخ مطلوب‌تری به ما می‌دهد.

```python
def jaccard_similarity(word1, word2):
    # intersection = len(list(set(word1).intersection(word2)))
    # union = (len(word1) + len(word2)) - intersection
    # return float(intersection) / union

    bigrams1 = [word1[i:i + 2] for i in range(len(word1) - 1)]
    bigrams2 = [word2[i:i + 2] for i in range(len(word2) - 1)]
    intersection = len(list(set(bigrams1).intersection(set(bigrams2))))
    union = len(set(bigrams1)) + len(set(bigrams2)) - intersection
    return float(intersection) / union
```

سپس با استفاده از تابع جاکارد می‌توانیم لغات نزدیک به هم را بیابیم و در یک لیست ذخیره کرده و مقدار آن را به عنوان خروجی تابع تحویل دهیم. می‌توانیم برای این کار دو رویکرد در نظر بگیریم:

1. برای این کار یک دیکشنری، یک ترشهولد و لغتی که می‌خواهیم در دیکشنری بررسی کنیم را به عنوان ورودی تابع در نظر می‌گیریم. در بدنه‌ی تابع روی اعضای دیکشنری پیمایش کرده و برای هر ترم مقدار کلمه‌ی داخل دیکشنری + معیار جاکارد را به عنوان تاپل ذخیره می‌کنیم. سپس اعضای این لیست را بر اساس معیار جاکارد بصورت نزولی مرتب می‌کنیم (چون هر چه مقدار به ۱ نزدیک‌تر باشد، دو عدد به هم شبیه‌تر هستند) و اینگونه لیستی از کلمات مشابه به ترتیب بیشترین شباهت به کلمه‌ی مورد نظر خواهیم داشت. باید در نظر بگیریم که مقادیر جاکارد را با ترشهولد دریافتی ورودی تابع مقایسه کنیم و در صورتی که از مقدار معین شده بیشتر باشند به لیست نهایی آن‌ها را اضافه کرده و خروجی بدهیم.

2. به جای استفاده از ترشهولد می‌توانیم ۱۰ تا از بهترین اعضای لیست را انتخاب کنیم.

بطور مشابه می‌توانیم یک لیست با استفاده از تابع levenshtein بسازیم. (استفاده از ۱۰ بهترین کلمه)

برای تابع نهایی که کوئری را تصحیح می‌کند، باید ترم‌های موجود در کوئری را بررسی کنیم اگر در دیکشنری وجود داشتند آن‌ها را به یک لیست اضافه می‌کنیم. در غیر اینصورت کلمات مشابهی که با jaccard similarity پیدا کردیم را در یک لیست می‌ریزیم، در صورتی که این لیست خالی بود (در صورت استفاده از ترشهولد)، کلمات مشابه اولیه و در غیر این صورت خود این لیست را به تابع levenshtein اضافه می‌کنیم. خروجی تابع نهایی بهترین انتخاب توسط تابع levenshtein خواهد بود.

```
58  def correct_query(q, dictionary, threshold):
59      modified_query = []
60      for word in q.split():
61          if word in dictionary:
62              modified_query[len(modified_query):] = [word]
63          else:
64              result_j = similar_words_j(dictionary, word, threshold)
65              if len(result_j) == 0:
66                  result_l = similar_words_l(dictionary, word)
67              else:
68                  result_l = similar_words_l(result_j, word)
69              modified_query.append(result_l[0])
70      return modified_query
```

بخش پنجم.

.TF IDF

برای این بخش و روش همانطور که توضیح داده شده بود از رابطه. lns-ltc استفاده می‌کنیم برای پیاده سازی این بخش نیز از فرمت class-based استفاده کردیم. (کلا approach ما اینه که یک سری کلاس داشته باشیم و در کلاس اصلی و main ما instance ها این کلاس به عنوان property وجود داشته باشند و استفاده‌ها و بخش‌ها مختلفی که از ما خواسته شده را در آن پیاده کنیم برای دسترسی آسان).

در این بخش ما برای داکیومنت و کوئری هر کدام جداگانه tf و idf و norm را محاسبه می‌کنیم (بنظرم توضیح خاصی این بخش‌ها ندارد و کد و الگوریتم ساده و گویا می‌باشد.) و در نهایت با for زدن روی همه داکیومنت‌ها تعداد k مد نظر (یعنی چندتا مشابه برتر در رنکینگ را یوزر می‌خواهد ببیند؟) بر می‌گردانیم.

البته کار اینجا تمام نمی‌شود و ما حالا doc_id را برای برترین‌ها داریم برای برگرداندن خود داکیومنت در فایل information_retrieval_system.py که نقش همان کلاس main ما را دارد یک فانکشن برای نمایش کوئری و پروسس درخواست کوئری کاربر قرار داده‌ایم که در آن خود سند خام و پیش‌پردازش نشده را بر می‌گردانیم.

.Proximity Search

تابع این بخش ۳ ورودی دریافت می‌کند؛ لیست داکیومنت‌آیدی‌ها، کوئری، لیست positional index ها و پنجره‌ی نام برده شده در داک پروژه. در این روش مطابق مطالب گفته شده اسنادی که تمام کلمات کوئری با حداکثر فاصله‌ی پنجره‌ی داده شده را دارا می‌باشند پیدا شده و به ترتیب امتیازشان بر اساس جستجوی ترتیب‌دار در فضای بردار tf-idf نمایش داده می‌شوند. خروجی این تابع به تابع جستجوی قبلی پاس داده می‌شود و خروجی‌ای مانند آن به ما می‌دهد و بقیه‌ی کار مطابق توضیحات قبلی انجام می‌شود.