تست کد ریشه‌یابی کلمات

با اجرای الگوریتم روی ۱۰۰ داکیومنت اول فایل ir-news-0-2.csv به نتایج زیر رسیدم:

گفت	گفت، می گفت، گفته ام، گفتم
گو	گو، می گوید، می گویند، می گوییم
رود	
رو	رو، برویم، می رود، برود، بروند، می رویم، روهایی
خواه	خواهند، می خواهد، خواهد، نخواهد، خواهیم، نمی خواهند، بخواهد، بخواهیم
سپاس	سپاسی
هنر	هنر، هنرهای
شریف	شریف
دوست	دوستان، دوست، دوستش
یاد	یاد
توان	می توان، بتوانند، نتوانستیم، توانستیم، می تواند، می توانستیم، می توانستند بتوان، نتوانسته، نمی تواند، بتواند
شنو	
کرد	کرد، کردند، کردیم، نکرد، می کردند، کرده اند، نمی کردیم، نکرده، کرده ایم
ساز	ساز، سازی، سازند، می سازد، سازان، بسازیم
دان	نمی دانند، می داند، بدانند، دانستند، دانست، بدانید، می دانند، می دانم، میدانی، نمی دانیم

روش خودکار استخراج کلمات پرتکرار از توکن‌ها

- یادگیری از متن: اگر دیدیم در تعداد زیادی از مواقع بعد از "مع" کلمه‌ی "ذلک" یا "هذا" می‌آید؛ یعنی "مع ذلک" و "مع هذا" کلمه‌های معنی دار هستند.
- اگر بتوانیم متونی پیدا کنیم که از نیم‌فاصله استفاده می‌کنند، می‌توانیم فرض کنیم کسی که از نیم‌فاصله استفاده می‌کند، سواد دارد؛ پس مثلا اگر در متن "ثبت‌نام" را دیدیم، می‌توانیم به این نکته پی ببریم که که "ثبت نام" هم یک کلمه معنی‌دار است که توسط کسی نوشته شده که احتمالا از کمبود سواد ابتدایی یا حوصله‌ی کافی رنج می‌برد.

پیاده سازی

با `python ./generate_inverted_index <csv address> <tokenizer type>` برنامه اجرا می‌شود. `Csv address` همان آدرس فایل داده‌ها است و `tokenizer type` می‌تواند `simple` یا `pro` باشد.

ترتیب اجرای کد به‌صورت زیر است:

۱. خواندن ستون content از فایل داده شده
۲. تکه‌تکه کردن اسناد ورودی (بصورت ساده یا پیشرفته)
۳. نرمال‌سازی و ریشه‌یابی (فقط برای مد پیشرفته)
۴. ساخت دیکشنری
۵. ساخت شاخص معکوس

۱. خواندن ستون content از فایل داده شده

در این مرحله با کمک fetch.py و کتابخانه‌ی pandas ستون content را از فایلی که در ورودی داده شده می‌خوانیم.

```
fetch_column(raw_csv, column_name)
```

این تابع آدرس فایل csv (raw_csv) و نام ستون دلخواه (column_name) را می‌گیرد و ستون داده شده را در قالب دیتافریم pandas پس می‌دهد.

۲. تکه‌تکه کردن اسناد ورودی

در این قسمت بر اساس ورودی داده شده به دو صورت توکن‌های اسناد استخراج شده را پیدا می‌کنیم.

```
tokens = tokenize(tokenize_type, contents)
```

حالت اول | ساده

در این حالت تابع tokenize از tokenizer.py ابتدا simple_tokenizer را صدا می‌زند و سپس فاصله‌ها و توکن‌های تکراری را از اسناد حذف می‌کند.

```
def simple_tokenize(contents):
    contents = cleanup(contents)
    tokens = tokenize_by_space(contents)
    tokens = remove_frequent_tokens(tokens)
    return tokens
```

تابع simple_tokenize اسناد خام را که در مرحله ۱ داشتیم به‌عنوان ورودی می‌گیرد.

اول تابع cleanup را صدا می‌زند که در normalizer.py تعریف شده و وظیفه‌ی حذف کردن اعداد و تگ‌های html را دارد. چون این پروژه برای اخبار فارسی نوشته شده است، برای ساده‌سازی عملیات حذف html و css و js در متن، تمام متون انگلیسی در این تابع از متن حذف می‌شوند. همچنین در cleanup نیم‌فاصله‌ها به فاصله تبدیل می‌شوند، تا نوعی نرمال سازی ساده انجام شود.

در مرحله‌ی بعدی که متون اضافه حذف شدند، تابع tokenize_by_space با کمک فاصله‌ها کلمات را جدا می‌کند و tokens را به‌عنوان خروجی می‌دهد. tokens لیستی از لیست‌ها است که در لیست iام توکن‌های سند iام ذخیره می‌شود.

در مرحله‌ی آخر لیست کلمات پرتکرار که در زیر آورده شده، از tokens حذف می‌شود:

- در
- برای
- چون
- است
- مانند
- باید
- به
- با
- از
- بی
- تا
- این

حالت دوم | پیشرفته

در این حالت تابع tokenize از tokenizer.py ابتدا pro_tokenizer را صدا می‌زند و سپس فاصله‌ها و توکن‌های تکراری را از اسناد حذف می‌کند.

```
def pro_tokenizer(contents):
    contents = cleanup(contents)
    tokens = tokenize_look_ahead(contents)
    tokens = remove_frequent_tokens(tokens)
    return tokens
```

قسمت cleanup و remove_frequent_tokens مانند قبل انجام می‌شود. فقط نحوه‌ی جداسازی پیچیده‌تر است.

الگوریتم کلی tokenize_look_ahead به این شکل است که برای هر کلمه که با فاصله جدا شده است، تا دو کلمه بعد از خود را چک می‌کند. اگر ترکیب این سه کلمه در دیکشنری farDic بود، هر سه کلمه یک توکن محسوب می‌شوند. وگرنه فقط کلمه‌ی بعدی خود را چک می‌کند و اگر ترکیب این دو کلمه در farDic بود، این دو کلمه یک توکن حساب می‌شوند؛ وگرنه این کلمه یک توکن تکی است.
همچنین اگر کلمه‌ی بعدی عضوی از لیست زیر باشد، این دو کلمه باهم یک توکن محصوب می‌شوند:

- ها
- تر
- ترین
- ام
- ات
- اش

نحوه‌ی تولید farDic به این شکل است که افعال چند بخشی پرتکرار که خودم استخراج کردم، از آدرس زیر قابل دسترسی هستند با کلمات چند بخشی، لیست farDic را تشکیل می‌دهند.

PersianStemmerLib/data/FreqVerbList.fa

لیست کلمات چندبخشی:

- بنا براین
- بنابر این
- بنا بر این
- ثبت نام
- چنان چه
- مع ذلک
- هم چنین
- تازه وارد
- غیر ممکن
- خوش حال
- هیچ کدام
- آن گونه
- هم اتاقی
- به صورت
- بی توجهی

- به خاطر
- نا آشنا
- سر و صدا
- تخت خواب
- علی الخصوص
- مع هذا
- به ناچار
- گه گاه
- هیئت گاه
- امیر کبیر
- توافق نامه

۳. نرمال‌سازی و ریشه‌یابی

این مرحله فقط در صورتی اتفاق می‌افتد که مد پیشرفته انتخاب شده باشد.

برای نرمال‌سازی تابع normalize از normalizer.py صدا زده می‌شود. در این تابع ابتدا ك، ي، ة، ؤ، أ به حالت‌های ساده‌ی فارسی نوشته می‌شوند و سپس اِعْرابْ‌ها از توکن‌ها حذف می‌شوند.

برای ریشه‌یابی از کتابخانه‌ی PersianStemmer استفاده شده است. تغییرات کوچکی در این کتابخانه برای تشخیص زمان فعل و فعالسازی تشخیص فعل انجام شده، بخاطر همین این کتابخانه ره هم در فایل پروژه ضمیمه کردم.

۴. ساخت دیکشنری

در این مرحله با توجه به لیست توکن‌ها دیکشنری کلمات ساخته می‌شود و کلمات تکراری حذف می‌شود.

۵. ساخت شاخص معکوس

با کمک دیشکنری مرحله‌ی قبل و لیست tokens که به نوعی شاخص غیرمعکوس است، inv_index ساخته می‌شود. کلیدها کلمات داخل دیکشنری هستند و مقدارشان شامل یک لیست است که عنصر اولش نشان‌دهنده‌ی تعداد تکرار کلید می‌باشد که به دنبالش شماره اسناد آورده شده است.

قوانین Heap و Zipf

۱. قانون Heap

برای بدست آوردن b و k دوبار الگوریتم را روی ۱۰۰۰ و ۵۰۰۰ سند انجام دادم. مقادیر بدست آمده در جدول زیر نشان داده شده است:

	تعداد اسناد	تعداد کل توکن‌ها T	تعداد کلمات یونیک M
ساده	۱۰۰۰	۲۹۱۰۰۷	۱۵۸۵۹
	۵۰۰۰	۱۵۷۵۵۱۱	۳۵۷۵۰
پیشرفته	۱۰۰۰	۲۸۳۳۱۶	۱۲۱۲۷
	۵۰۰۰	۱۵۳۴۲۴۶	۲۵۶۱۴

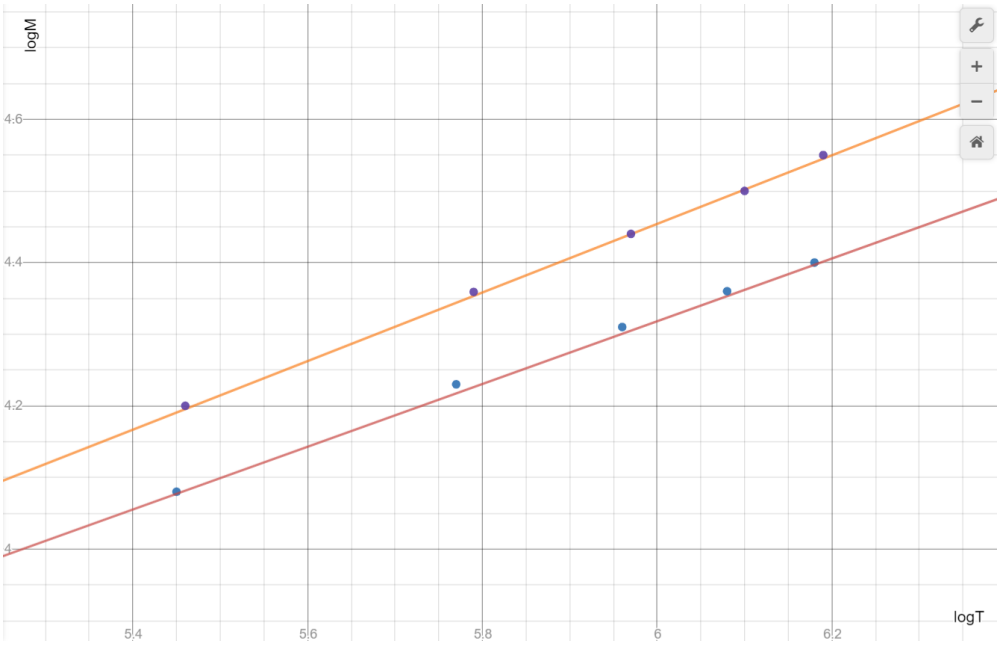
با حل دو معادله استخراج شده از اطلاعات بالا و معادله زیر به نتایج زیر می‌رسیم:

$$\log M = b \log T + \log k$$

```
Simple: k = 38; b = .47
Pro: k = 48; b = .43
```

برای بررسی نتیجه، آزمایش را روی دو، سه و چهارهزار سند امتحان کردم.

نمودار حالت ساده و پیشرفته: دیده می‌شود که حالت ساده خطای کمتری دارد.



خط بالایی برای حالت ساده و خط پایینی برای حالت پیشرفته است. نقاط آبی و بنفش هم به ترتیب نشان‌دهنده‌ی نتایج آزمایشات انجام شده روی الگوریتم پیشرفته و ساده می‌باشد.

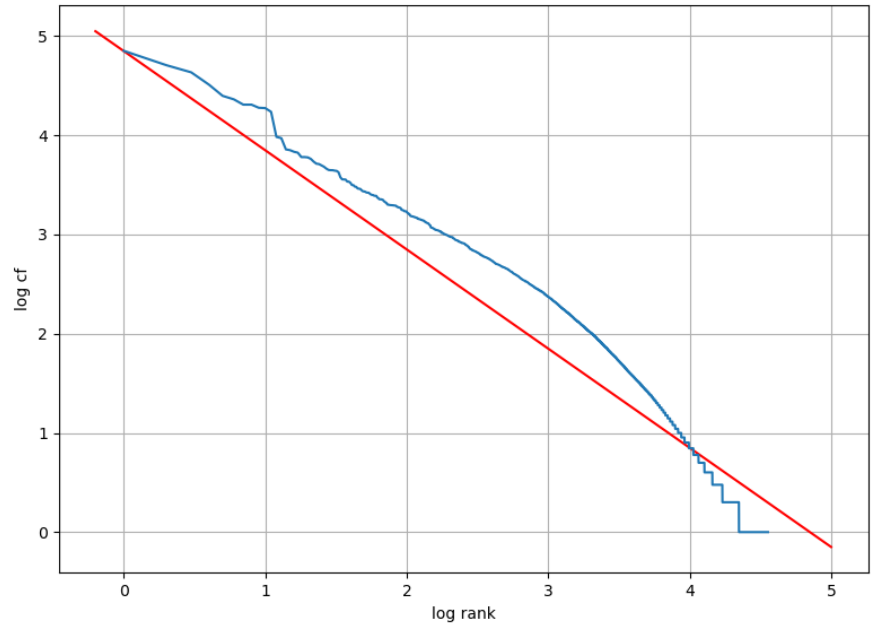
۲. قانون Zipf

برای بررسی این قانون دو روش مختلف را با ۵۰۰۰ سند امتحان کردم و فایل‌های ضمیمه‌شده‌ی `5000pro_zipf.txt` و `5000simple_zipf.txt` در هر سطر به‌صورت نزولی تعداد پرتکرارترین کلمه را نشان می‌دهد.

با توجه به نکات گفته شده، برای تعیین مقادیر به نتایج زیر رسیدم:

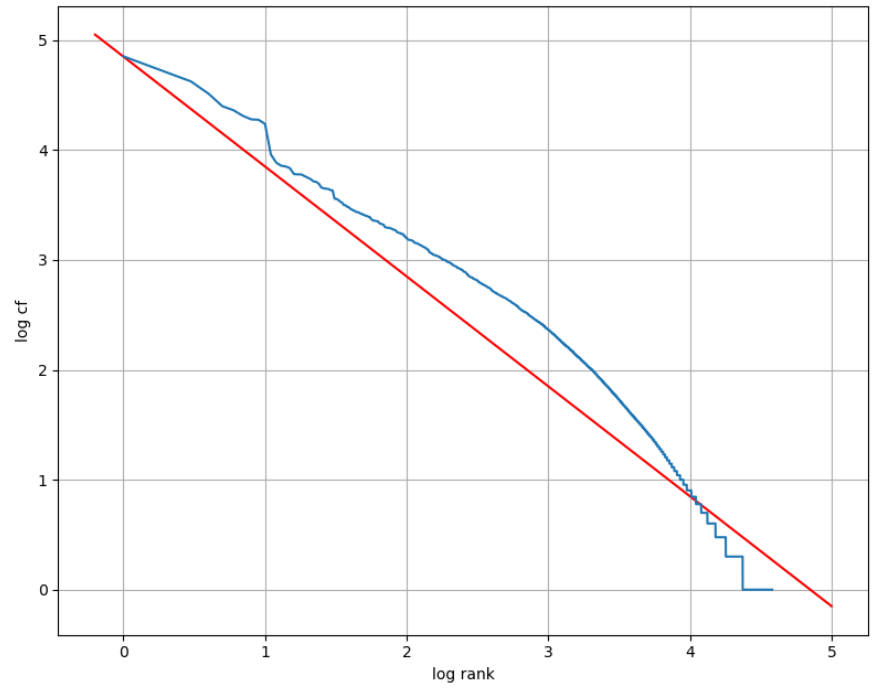
```
Simple: k = 71194
Pro: k = 71181
```

نمودار حالت ساده



دو نمودار خیلی نزدیک به هم هستند. خط پیش‌بینی قانون zipf بدلیل نزدیکی مقدار k در دوحالت تقریبا یکسان هستند.

نمودار حالت پیشرفته:



مقایسه‌ی دو نمودار(آبی ساده و قرمز پیشرفته):