دانشگاه صنعتی امیرکبیر

دانشکده‌ی علوم‌کامپیوتر

گردآورنده:

شیده هاشمیان

شماره دانشجویی:

۹۶۱۳۴۲۹

تمرین دوم درس مباحثی در علوم‌کامپیوتر

عنوان : تشخیص هرزنامه ها

استاد درس: دکتر اکبری

پاییز ۹۹

این پیاده‌سازی متشکل از چهار فایل `.py` و یک فایل `.ipynb` است که هریک از آن‌ها و توابع موجود در آن‌ها در زیر توضیح داده شده است.

برای اجرای برنامه، آدرس محل پوشه‌ی `test` (شامل دو پوشه‌ی `hamtesting` و `spamtesting`) و `train` (شامل دو پوشه‌ی `hamtraining` و `spamtraining`) را در `given_data_root_path` که در فایل `constant.py` وجود دارد وارد کنید. همچنین تمام فایل‌های ساخته شده در برنامه، در پوشه‌ای که آدرس آن رشته‌ی `document_root_path` (در فایل `constant.py`) است قرار دارند.

۱. **ثابت‌ها (`constant.py`):**

این فایل شامل متغییرهایی است که در دیگر فایل‌ها مورد استفاده قرار می‌گیرند و در میان آن‌ها یکسان است که شامل آدرس پیکره‌های اولیه، آدرسی که برنامه فایل‌هایی که در طول اجرا تولید می‌کند در آن آدرس ذخیره کند هست. همچین مجموعه‌ای از علائم نگارشی و علائم غیر الفبایی که در پیکره‌ی آموزش موجود بود هست که در مرحله‌ی نرمال‌سازی متن از آن‌ها استفاده شود. علاوه بر این‌ها شامل ثابت عددی محدودسازی تعداد کلمات مهم در نظر گرفته شده زمان استفاده از χ^2 هست.

۲. **ابزارهای پردازش زبان**(LP_toolkits.py):

این فایل متشکل از سه تابع است.

- تابع sub_alphabets که برگرفته شده از تابعی با همین نام در پکیج parsivar هست باتوجه به نیاز در این برنامه در برخی از قسمت‌ها عوض شده است که با گرفتن یک رشته در آن تمام حروفی که در این برنامه برای ما معنی‌دار هستند را به یک مجموعه حروف مشخص map می‌کند تا کلماتی که یک نگارش دارند یکسان شناسایی شوند.
- تابع normalizer نرمال‌سازی ابتدایی که شامل اجرای تابع sub_alphabets بر روی رشته‌های ورودی، حذف علائم نگارشی، علائم غیر الفبایی و اعداد است و نهایتا رشته‌ی نهایی را به عنوان خروجی بازمی‌گرداند.

۳. **پیش‌پردازش داده‌ها (`preprocessing.py`):**

ابتدا یک نمونه از کلاس‌های `findStems` (به نام `Stemmer`)از کتابخانه‌ی `parsivar` و `Lemmatizer` (به نام `Lemmatizer`)از کتابخانه‌ی `hazm` را می‌سازیم و همچنین یک مجموعه از ایست‌واژه‌های موجود در لیست `()stopwords_list` (به نام `stopwords`)از کتابخانه‌ی `hazm`.

- تابع `stemmer(email)`: با ترکیبی از ترتیب اعمال توابع `Lemmatizer.lemmatize(word)` و `Stemmer.convert_to_stem(word)` (بدست آمده با آزمایش چند حالت روی داده و دیدن نتیجه، بهترین ترکیب آورده شده) ریشه‌ی افعال موجود در ایمیل ورودی داده‌شده را یافته و جایگزین افعال می‌کند. سپس رشته‌ی نهایی را خروجی می‌دهد.
- تابع `prepare_data(ham_dir, spam_dir, train=True)`: مسیر داده‌های متنی داده‌های `ham`، مسیر داده‌های متنی داده‌های `spam` و متغیر `Boolean` را که در صورت `True` بودن نشان دهنده‌ی اجرای این تابع برای داده‌ی آموزش و در صورت `False` بودن نشان‌دهنده‌ی اجرای این تابع روی داده‌های آزمایش است را ورودی می‌گیرد. سپس باتوجه به نوع اجرای آن (آموزش یا آزمایش) نام مناسب برای فایل‌های متنی را می‌سازد. پس از آن در یک حلقه تمام ایمیل‌ها را ابتدا به تابع `stemmer` داده و سپس در خروجی آن کلمات موجو در مجموعه‌ی `stopwords` را حذف کرده و رشته‌ی نهایی را در کلاس متناسب خود در لغت‌نامه‌ای ذخیره می‌کند. در انتها این لغت‌نامه را خروجی می‌دهد.

```
{'ham': [processed_1, …], 'spam': [processed'_1, …]}
```

- تابع `chi_square_calculator(processed_emails_dict)`: این تابع مانند تابع هم‌نام خود در تمرین قبلی گرفته، با گرفتن لغت‌نامه‌ای مانند خروجی تابع قبل، معیار χ^2 را برای تمامی داده‌ها متنسب با کلاسشان محاسبه کرده، سپس ماتریس آن، و دو لغت‌نامه که `index` شده‌ی کلاس‌ها و کلمات است تا بتوان با آن‌ها مهم‌ترین کلمات را استخراج کرد.
- تابع `most_important_tokens(processed_emails_dict)`: این تابع لغت‌نامه‌ای مانند خروجی تابع `prepare_data` را ورودی می‌گیرد و با استفاده از تابع قبل، ۵۰۰ مهم‌ترین کلمه‌ها را استخراج و آن‌ها را `index` می‌کند و لغت‌نامه‌ی آن را خروجی می‌دهد.
- تابع `construct_vocabulary(processed_emails_dict)`: این تابع لغت‌نامه‌ای مانند خروجی تابع `prepare_data` را ورودی می‌گیرد و دو لغت‌نامه، یکی ساده و با استفاده از تمام لغت‌ها و دیگری پیشرفته و تنها با استفاده از لغت‌های مهم، را می‌سازد و در حافظه با نام‌های `vocabulary_simple.pickle` و `vocabulary_advance.pickle` ذخیره می‌کند.
- تابع `vectorize_email(processed_email, vocabulary)`: این تابع لغت‌نامه و متن یک ایمیل را گرفته و بردار متناظر با آن ایمیل را خروجی می‌دهد.
- تابع `vectorize_data(processed_emails_dict, vocabulary)`: این تابع لغت‌نامه‌ای مانند خروجی تابع `prepare_data` را ورودی می‌گیرد و با استفاده از تابع قبل هر یک از ایمیل‌ها را بردار کرده و دیکشنری بردارها را خروجی می‌دهد.

  `{'ham':[vec1, …], 'spam':[vec'1, …]}`

۴. **دسته‌بند** (`model.py`):

- تابع `cosine_similarity(vec1, vec2)`: با گرفتن دو بردار، شباهت کسینوسی میان آن دو را حساب کرده و خروجی می‌دهد.
- تابع `idf_calculator(vectorized_train_dict)`: با گرفتن لغت‌نامه‌ای مانند خروجی تابع `vectorize_data`، معیار `idf` را برای هر کلمه‌ی متناظر با خانه‌ی `i`ام از بردارها محاسبه کرده و نهایتا لیستی که درایه‌ی `i`ام آن نشان‌دهنده‌ی مقدار `idf` برای کلمه‌ای با همین `index` در لغت‌نامه است خروجی می‌دهد.
- تابع `tf_calculator(vectorized_train_dict)`: با گرفتن لغت‌نامه‌ای مانند خروجی تابع `vectorize_data`، معیار `tf` را به‌ازای هر ایمیل و کلمه محاسبه کرده و در ساختاری مانند زیر ذخیره کرده و خروجی می‌دهد. (در صورتی که `index` کلمات استفاده شده برای بردار کردن مجموعه بردار ورودی `0` تا `m` باشد)

  $\{'ham': [[\log(v_0+1), \dots, \log(v_m+1)], \dots], 'spam': [[\log(v_0+1), \dots, \log(v_m+1)], \dots]\}$
- تابع `tf_idf_calculator(non_zero_indices, tf, idf)`: لیست `tf` مانند داخلی‌ترین لیست موجود در ساختاری مانند خروجی تابع محاسبه‌گر آن و لیست `idf` مانند خروجی تابع محاسبه‌گر آن است. اندیس‌های غیر صفر (`non_zero_indices`) هم آرایه‌ای از اندیس‌هایی است که در بردار ایمیلی که این معیار را برای آن می‌خواهیم محاسبه کنیم مقدار ناصفری (لزوما مثبت زیرا شمارش کلمات متناظر با آن اندیس در متن ایمیل را در آن قرار می‌دهیم) را اخذ کرده‌اند (که معادل با اندیس کلمه‌های موجود در این ایمیل است). حال به‌ازای اندیس‌های موجود در لیست `non_zero_indices` درایه‌ی متناظر این اندیس در دو لیست `tf` و `idf` را در هم ضرب کرده و مجموع این مقادیر را خروجی می‌دهد.

- تابع `knn(new_email_vec, similarity_func, vectorized_train_dict=None, idf=None, tf_dict=None)`: این تابع دو متغیر اجباری دارد. یکی از آن‌ها بردار ایمیلی ایت که می‌خواهیم دسته‌ی آن را با استفاده از این الگوریتم بیابیم (`new_email_vec`) است و دیگری (`similarity_func`) یک عدد است که در صورتی که مقدار آن `0` باشد به این معنا است که از معیار شباهت کسینوسی استفاده می‌شود و در صورتی که برابر `1` باشد، از امتیاز `tf-idf`. سه متغییر اختیاری بعدی را باتوجه به این که از کدام معیار استفاده کنیم ورودی می‌دهیم.
  - شباهت کسینوسی: دیکشنری که مانند خروجی تابع `vectorize_data` برای داده‌های آموزش است، ورودی داده می‌شود. سپس برای هر کدام از بردارهای موجود در آن، شباهت کسینوسی با بردار ایمیل ورودی را با استفاده از تابع محاسبه‌گر آن حساب کرده و دو دیکشنری `index` شده برای ذخیره‌سازی کلاس متناظر با هر امتیاز و خود امتیاز را می‌سازیم.
  - امتیاز `tf-idf`: لیست و دیکشنری `idf` و `tf` که مانند خروجی تابع محاسبه‌گر آن‌ها هستند را ورودی می‌دهیم. سپس برای هر داخلی‌ترین لیست موجود در دیکشنری `tf`، امتیاز `tf-idf` را بری آن و بردار ایمیل ورودی با استفاده از تابع محاسبه‌گر آن محاسبه کرده و دو دیکشنری `index` شده برای ذخیره‌سازی کلاس متناظر با هر امتیاز و خود امتیاز را می‌سازیم.

  در نهایت با استفاده از این دو دیکشنری و داده‌ساختار `heap`، اندیس‌های ۷ داده‌ای که بیشترین امتیاز برای آن‌ها بوده‌است را گرفته و در بین کلاس‌های متناظر با آن‌ها، کلاسی که بیشترین تکرار را دارد خروجی می‌دهد.

- تابع `all_probabilities_calculator(vectorized_train_dict)`: دیکشنری ای مانند خروجی تابع `vectorize_data` برای داده‌های آموزش را می‌گیرد و دیکشنری‌ای که کلیدهای آن نام دو کلاس و مقادیر هرکدام آرایه‌ای از احتمالات است که به‌طور مثال اندیس `i`ام این آرایه برای کلید `ham` نشان دهنده‌ی مقدار احتمال زیر است را ساخته و خروجی می‌دهد. (که در آن w_i کلمه‌ای که اندیس آن در فرهنگ لغت `i` است، هست)

$$p(w_i|'ham') = \frac{tf(w_i|'ham')}{|F| + \sum_{w_j \in F} tf(w_j|'ham')}$$

که در آن F مجموعه‌ی تمام لغات آمده در دسته‌ی $'ham'$ است. همچنین دلیل ضرب نکردن احتمال هر کلاس برابر بودن این مقدار برای دوکلاس (باتوجه به داده‌های اموزش) و سادگی محاسباتی است.

۵. **ارزیابی مدل رو ی ده دا های آموزش**(دو تابع در فایل `model.py`):

- تابع `evaluation(vectorized_test_dict,vectorized_train_dict,similarity_func=0, naive=False)`: دو ورودی مجموعه بردار آموزش و مجموعه بردار آزمایش همانند خروجی تابع `vectorize_data` این برای داده‌ی آموزش و داده‌ی آزمایش است. و باتوجه به دیگر ورودی‌ها عملکرد این تابع در زیر توضیح داده‌شده است.
  - حالت `similarity_func=0, naive=False`: در این حالت تابع برای یافتن کلاس احتمالی هر ایمیل از مدل `knn` و تشابه کسینوسی استفاده می‌کند. (صدا زدن تابع `knn` با ورودی‌های متناسب با این حالت برای هر بردار ایمیل آزمایش) پس از آن لیست تخمین زده شده توسط مدل و لیست دسته‌های واقعی ایمیل‌های آموزش (هر درایه متناظر با یک ایمیل آموزش) را ذخیره می‌کند.
  - حالت `similarity_func=1, naive=False`: در این حالت تابع برای یافتن کلاس احتمالی هر ایمیل از مدل `knn` و امتیاز`tf-idf` استفاده می‌کند. (صدا زدن تابع `knn` با ورودی‌های متناسب با این حالت برای هر بردار ایمیل آزمایش) پس از آن لیست تخمین زده شده توسط مدل و لیست دسته‌های واقعی ایمیل‌های آموزش (هر درایه متناظر با یک ایمیل آموزش) را ذخیره می‌کند.
  - حالت `naive=True`: در این حالت تابع برای یافتن کلاس احتمالی هر ایمیل از مدل `naïve bayes` استفاده می‌کند. به این صورت که ابتدا تابع `all_probabilities_calculator` را با دادن مجموعه بردار داده‌ی آموزش صدا می‌زند و از خروجی آن برای محاسبه‌ی احتمال قرار گیری هر ایمیل آزمایش در هر دسته استفاده می‌کند و ایمیل آزمایش را در دسته‌ای که این احتمال برای آن بیشتر است قرار می‌دهد (ابتدا اندیس‌های غیر صفر بردار ایمیل آزمایش را پیدا کرده و سپس با جمع لگاریتم‌های اندیس‌های متناظر با این اندیس‌ها از آرایه احتمالات هر دسته، لگاریتم احتمال تعلق به آن دسته را می‌یابد.).پس از آن لیست تخمین زده شده توسط مدل و لیست دسته‌های واقعی ایمیل‌های آموزش (هر درایه متناظر با یک ایمیل آموزش) را ذخیره می‌کند. بازهم به‌دلیل برابر بودن احتمالات هر کلاس و وارد نکردن آن در رابطه‌ی $p(w_i|c_j)$ ، مخرج کسر در رابطه‌ی محاسبه‌ی $p(c_j|d_i)$ برای تمامی کلمات یکسان بوده و از ضرب آن برای سادگی محاسبات (و بی‌تاثیر بودن آن در نتیجه‌ی مدل) صرف نظر شده است.

  در انتها لیست تخمین زده شده توسط مدل و لیست دسته‌های واقعی ایمیل‌های آموزش همراه با دیکشنری‌ای که کلاس‌ها اندیس گذاری شده‌اند را به تابع زیر داده و آن ارزیابی مدل استفاده شده را نمایش می‌دهد.

- تابع `evaluation_metrics_calculator(predicted_ls,true_ls,indexed_classes)`: در این تابع با گرفتن دو آرایه‌ی `true_ls` و `predicted_ls` این که به‌ترتیب در آن دسته‌های حقیقی و دسته‌های انتسابی توسط مدل در آن قرار داردن (با حفظ ترتیب) و دیکشنری اندیس گذاری شده‌ی دسته‌ها، معیارهای ارزیابی را محاسبه کرده و با پیامی مناسب نمایش می‌دهد.

  و احتمال متناظر با آن را در فایل `probabilities.txt` ذخیره می‌کنیم.

۶. **نتایج ارزیابی**(`main.ipynb`): در این جا تمام حالات مدل‌ها صدا زده شده است و نتایج آن در زیر آمده است.

Use all terms as vocabulary

	KNN (k=7)				Naïve Bayes	
	cosine similarity		tf-idf			
	ham	spam	ham	spam	ham	spam
Confusion matrix	196 59 4 141	141 4 59 196	193 68 7 132	132 7 68 193	193 21 7 179	179 7 21 193
Precision	0.7050	0.9800	0.6600	0.9650	0.895	0.965
Recall	0.9724	0.7686	0.9496	0.7395	0.9624	0.9019
F1	0.8174	0.8615	0.7788	0.8373	0.9275	0.9324
accuracy	0.8425		0.8125		0.93	
f1 macro	0.8395		0.8080		0.9299	
f1 micro	0.8425		0.8125		0.93	

Use important terms as vocabulary

	KNN(k=7)				Naïve Bayes	
	cosine similarity		tf-idf			
	ham	spam	ham	spam	ham	spam
Confusion matrix	195 66 5 134	134 5 66 195	191 22 9 178	178 9 22 191	192 17 8 183	183 8 17 192
Precision	0.6700	0.9750	0.8900	0.9550	0.915	0.96
Recall	0.9640	0.7471	0.9519	0.8967	0.9581	0.9187
F1	0.7906	0.8460	0.9200	0.9249	0.9361	0.9389
accuracy	0.8225		0.9225		0.9375	
f1 macro	0.8183		0.9224		0.9375	
f1 micro	0.8225		0.9225		0.9375	

همان‌طور که در جدول می‌توان دید برای حالت `knn` با استفاده از `tf-idf` و `naïve bayes`،استفاده کردن از تنها ۵۰۰ کلمه‌ی تاثیرگذار به‌عنوان لغت نامه نتیجه‌ی بهتری را می‌دهد (که برای `knn` با استفاده از شباهت کسینوسی این‌طور نیست) البته این تاثییرگزاری در `tf-idf` محسوس‌تر است. و نهایتا می‌توان دید که بهترین نتیجه برای حالتی‌است که از مدل `naïve bayes` و تنها ۵۰۰ کلمه‌ی تاثیرگذار به‌عنوان لغت نامه برای دسته‌بندی استفاده کنیم.