دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس پردازش زبان طبیعی

تمرین دوم

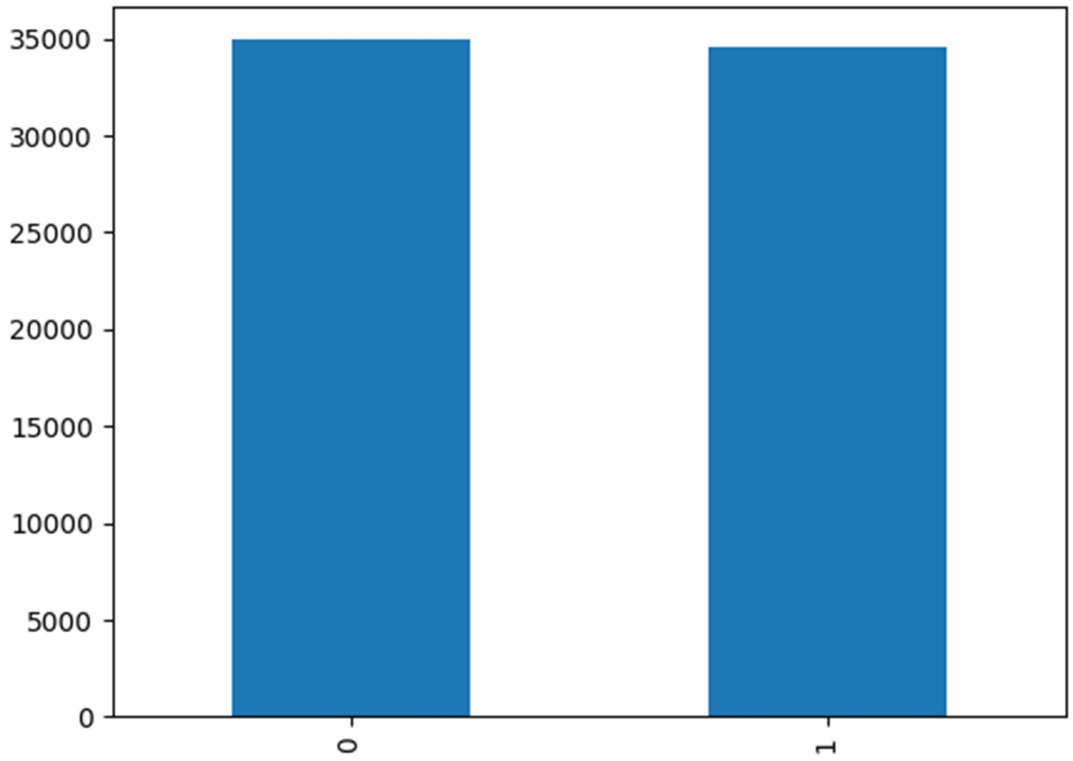
نام و نام‌خانوادگی	پرهام بیچرانلو
شماره‌دانشجویی	۸۱۰۱۰۰۳۰۳
تاریخ ارسال	۱۴۰۲/۰۲/۰۸

۱. سوال ۱- تحلیل احساسات دادگان اسنپ فود

۱-۱. پیش پردازش

ابتدا دیتا را با کتابخانه pandas خواندیم. ستون‌های اضافه را حذف کردیم. همچنین null بودن را برسی کردیم که تعدادی از سطرها در ستون label_id مقدار null داشتند پس آن سطرها را حذف کردیم. حدود ۵۲۰ سطر از ۷۰۰۰ سطر حذف شدند. سپس ستون label_id را از float تبدیل به int کردیم که لیبل‌ها استاندارد شوند. بعد ستون label را حذف کردیم چون همان label_id کافی است.

حالا نمودار توزیع لیبل دیتاست را رسم کردیم که در ادامه آمده است:



شکل ۱. نمودار توزیع لیبل‌ها

که تقریبا توزیع لیبل داده‌ها متوازن هستند.

برای ادامه کار همانطور که در سوال خواسته شده فقط ۲۰ درصد داده‌ها را برمی‌داریم و بقیه را دور می‌ریزیم. برای این که توزیع داده‌ها حفظ بشه ابتدا روی label_id گروه بندی می‌کنیم و بعد 0.2 از هر لیبل را برمی‌داریم. در این مرحله تعداد داده‌ها به 13896 کاهش پیدا می‌کند.

حال نوبت به مرحله اصلی این گام می‌رسد یعنی تمیزسازی نظرات مشتریان. برای این کار از کتابخانه قدرتمند hazm بهره می‌گیریم. در تابع preprocess(text) عملیاتی مثل اطلاح فاصله گذاری‌ها، حذف اعراب، حذف کاراکترهای بی‌استفاده، تبدیل ارقام انگلیسی به فارسی و ... را روی متن انجام می‌دهد و یک متن استاندارد و تمیز تحویل می‌دهد.

در نهایت هم دادگان را با ماژول train_test_split کتابخونه Sklearn داده‌ها را به دو بخش train و test تقسیم می‌کنیم. که نسبت تقسیم داده‌ها ۹۰ به ۱۰ درصد است.

۱-۲. پیاده سازی روش Tf-idf

ایده کلی: هر کامنت شامل چندین کلمه است، ما می‌توانیم برای هر کدام از کلمات داخل داده‌های train به روشی که در ادامه ذکر خواهیم کرد مقدار Tf-idf مرتبط آن را بدست بیاوریم. اما سوال اینکه آیا طول نظرات و تعداد کلمات آن‌ها یکسان است؟ اگر یکسان نباشد که نمی‌توان آن‌ها را به عنوان feature به naïve bayes بدهیم. راه حل اینکه برای تمام کامنت‌ها یک بردار با طول ثابت به اندازه تعداد کل کلمات داده‌های train درنظر بگیریم. و برای یک کامنت مقدار tf-idf کلمات آن را بگذاریم و برای بقیه کلمات که ظاهر نشده اند صفر بگذاریم.

سوال دیگه که مطرح میشه اینکه Tf-idf را از کدوم داده‌ها بدست آوریم؟ آیا می‌تونیم از داده‌های test استفاده کنیم؟ جواب خیر است چون اصولا ما در دنیای واقعی به داده test دسترسی نداریم. و اینجا هم برای اینکه درست عمل کنیم باید با همین فرض جلو بریم. پس فقط از داده‌های train برای بدست آوردن tf-idf استفاده می‌کنیم. نکته مهم دیگه اینکه اگه کلمه‌ای در test بود اما در train نبود مقدار tf-idf متناظر آن را باید صفر بگذاریم.

حالا بریم سراغ توضیح مراحل کد و پیاده سازی:

در گام اول هر جمله را tokenize کنیم و البته در همین حین کلمات را به مجموعه کلمات اضافه می‌کنیم تا در آخر تمام کلمات train در آن باشند. و برای هر کدام از این کلمات یک index در نظر می‌گیریم تا یک دیکشنری شامل تمام کلمات train همراه با ایندکس آن‌ها داشته باشیم.

در گام بعد می‌خواهیم بشماریم که در داکیومنت یک کلمه چندبار تکرار شده است که در تابع count_dict پیاده شده است. از خروجی آن می‌توان برای بدست آوردن inverse_doc_freq استفاده کرد. یعنی تعداد کل کامنت‌ها تقسیم بر تعداد وقوع کلمه در کل داکیومنت.

$$IDF(word) = log\ \frac{total\ number\ of\ document}{1 + number\ of\ comments\ containing\ word}$$

برای tf تعداد وقوع یک کلمه تقسیم بر تعداد کل کلمات کامنت را می‌توان درنظر گرفت که طول نظرات کمتر مهم باشد. البته در اسلایدهای درس اینطور حساب می‌کند که از تعداد وقوع کلمه در کامنت لگاریتم می‌گیرد. ما از هر دو روش استفاده کردیم. و روش اول یکی دو درصد در دقت بهتر عمل کرد پس همان را نگه داشتیم و اون یکی رو کامنت کردیم. این فرآیند در تابع termfreq پیاده سازی شده است.

$$TF(word, comment) = \frac{number\ of\ times\ word\ appears\ in\ comment}{total\ number\ of\ words\ in\ comment}$$

در نهایت ضرب tf در idf را برای کلمات جمله بدست می‌آوریم. سپس ایندکس کلمه را بدست آورده و در خانه متناظر آن در بردار ویژگی را برابر مقدار بدست آمده قرار می‌دهیم. این فرآنید در تابع tf_idf پیاده شده است.

$$TF_{IDF(word,comment)} = TF(word, comment) * IDF(word)$$

حال برای تمام جملات داده‌ی train تابع tf_idf را فراخوانی کرده و embedding کلمات آن‌ها را در یک لیست ذخیره می‌کنیم. دقت شود برای هر جمله یک وکتور به سایز کل کلمات داخل دیتاست داریم. یعنی این وکتورها اسپارس هستند چون خیلی از کلمات در یک نظر نیامدند پس tf_idf متناظر با آن‌ها را صفر در نظر می‌گیریم.

برای داده test هم همین کار را می‌کنیم. نکته مهم اینکه ممکن است کلماتی در تست باشند که در train نباشند. برای این کلمات هم tf_idf را صفر درنظر می‌گیریم. در پیاده سازی برای هندل کردن این موضوع از روش try,except در تابع tf_idf استفاده می‌کنیم. یعنی اگر دید index_dict[word] موجود نیست کاری انجام نمی‌دهد و به سراغ کلمه بعدی می‌رود.

در آخر هم از یک طبقه بند multinomial naïve bayes برای پیش بینی لیبل کامنت‌های داده test استفاده می‌کنیم. فیچرهایی که به مدل می‌دهیم همان بردار tf_idf مربوط به جملات است. ابتدا مدل را با داده‌های train آموزش می‌دهیم که فیچرهای بردارهای tf_idf مروبط به کامنت‌های train است و تارگت آن لیبل‌های train است. بعد از آن بردارهای tf_idf مربوط به کامنت‌های test را می‌دهیم تا سنتیمنت را پیش بینی کند. نتایج را با classification_report بدست می‌آوریم که در بخش ۱-۴ گزارش شده است.

.

۱-۳. پیاده سازی روش PPMI

ایده کلی: روش PMI می‌آید به طور کلی میزان هم رخداد بودن دو پدیده که می‌تواند کلمه با کلمات دیگر یا بقیه مثال‌ها را محاسبه کند. برای این سوال ایده‌ای که به ذهنم رسید و در برخی منابع مشابه آن را دیدم این است که میزان هم رخداد بودن هر کلمه با لیبل مثبت و لیبل منفی را بدست آوریم. که برای تمام کلمات کامنت این کار را کنیم و کلماتی که در آن کامنت نیست مقدار صفر قرار دهیم. در آخر یک بردار ویژگی اسپارس داریم که می‌توانیم به classifier به عنوان ورودی بدهیم تا آموزش ببینند.

$$PMI(word, happy) = \frac{P(word, happy)}{P(word)P(happy)}$$

$$PMI(word, sad) = \frac{P(word, sad)}{P(word)P(sad)}$$

شاید سوال پیش بیاد که دوتا بردار PPMI برای هر کامنت داریم چطور این را به عنوان feature به classifier می‌دهیم؟ جواب سادست ما در انتها این دو بردار را concat می‌کنیم تا یک بردار واحد داشته باشیم. البته روش‌های دیگه هم به ذهنم رسید ولی این رو مناسب‌تر دیدم. روش‌های دیگه مثل میانگین گرفتن از PPMI کلمه با لیبل مثبت و کلمه با لیبل منفی. یا روش تفاضل بردارها از هم.

اینجا هم مانند قبل برای بدست آوردن PPMI از داده train استفاده می‌کنیم و بعدا روی داده test اعمال می‌کنیم.

نحوه پیاده سازی:

برای بدست آوردن P(happy) و P(sad) باید نسبت کامنت‌های مثبت و منفی را نسبت به کل داده‌ها حساب کنیم. برای این کار ابتدا روی داده‌های train دستور value_counts روی ستون label_id می‌زنیم تا تعداد کامنت‌های مثبت و منفی را حساب کند. حالا برای بدست آوردن احتمالات P(happy) و P(sad) تعداد آن‌ها را تقسیم بر کل کامنت‌ها می‌کنیم. که این احتمالات برابر شد با:

$$P(happy) = 0.50055$$
$$P(sad) = 0.49944$$

برای بدست آوردن P(word) کافی است تعداد کامنت‌هایی که شامل کلمه word هستند را بدست آوریم. این کار را در تابع doc_freq انجام می‌دهیم که از دیکشنری word_count که قبلا ساختیم استفاده می‌کند. که این احتمال برای هر کلمه باید حساب شود.

برای بدست آوردن P(word, happy) و P(word,sad) ابتدا باید داده‌های این دو کلاس را جدا کنیم. و بعد هر کامنت آن‌ها را توکنایز کنیم. برای هر کدام از این مجموعه‌ها باید یک دیکشنری بسازیم که تعداد کلمات تکرار شده در آن‌ها را بشمارد. برای این کار از تابع count_dict استفاده می‌کنیم که در مرحله tf_idf پیاده کردیم. حال برای بدست آوردن احتمالات P(word,happy) و P(word,sad) کافی است از دیکشنری مربوط به هر کدام از لیبل‌ها تعداد تکرار کلمه word را دربیاریم. بعد آن را تقسیم بر تعداد کل تعداد کامنت‌ها کنیم. این کار در تابع sad_word و happy_word انجام می‌شود.

حالا مرحله آخر تابعی است که با استفاده از این احتمالاتی که تا اینجا بدست آوردیم PPMI را حساب کند. این عمل در تابع calculate_PPMI انجام می‌شود. که ورودی آن کلمه است. ابتدا برای آن کلمه تابع doc_freq را فراخوانی کرده تا P(word) را برگرداند. بعد happy_word و sad_word را فراخوانی می‌کنیم تا احتمالات P(word,happy) و P(word, sad) را حساب کنند. در آخر PMI(word, happy) و PMI(word, sad) را از فرمولی که اول گفته شد بدست می‌آوریم. اما این با PPMI فرق دارد چون ممکن است مقادیر منفی دهد و این بامعنا نیست ما آن‌ها را صفر درنظر می‌گیریم. که با یک شرط ساده این کار را انجام دادیم. در آخر این تابع PPMI مربوط به کلمه و مثبت و منفی را برمی‌گرداند.

حال برای اینکه برای تمام کلمات جمله این مقادیر را بدست آوریم در تابع PPMI که ورودی آن جمله است ابتدا دو وکتور با سایز تعداد کلمات کل داده با مقدار اولیه صفر می‌سازم. سپس برای هر کلمه جمله تابع calculate_PPMI را فرخوانی کرده و خروجی آن را در دو وکتور ساخته شده می‌ریزیم. در آخر هم این وکتورها را به عنوان خروچی تابع برمی‌گردانیم. استفاده از تکنیک try, except برای هندل کردن کلماتی است که در داده train نبودند تا این دستورات را نادیده بگیریم.

در نهایت برای تمام جملات تابع PPMI را فراخونی می‌کنیم. و مقادیر خروجی آن را در یک لیست قرار می‌دهیم.

برای کلمه "ممنون" خواستیم مقدار PPMI مثبت و منفی را مشاهده کنیم تا حس آن را درک کنیم. نتایج در زیر آمده است:

$$P\left('\text{ممنون}', sad\right) = 0.1386$$

$$P\left('\text{ممنون}', happy\right) = 1.8633$$

خب همانطور که مشاهده می‌کنید نتایج آن منطقی است. یعنی کلمه "ممنون" خیلی بیشتر در داکیونت‌های مثبت(خوشحال) تکرار شده است. پس به نظر الگوریتم ما درست کار می‌کند.

برای انیکه وکتور PPMI جملات داده train دو بعدی شود از reshape اسفاده می‌کنیم تا وکتور PPMI مربوط به sad و happy را کنار هم قرار دهد. تعداد featureها 21824 می‌شود.

برای داده test هم این کارها را تکرار می‌کنیم. در نهایت یک naïve bayes با استفاده از کتابخانه scikit learn می‌سازیم و با featureهای بدست آمده از داده train و لیبل مربوط به آن‌ها، آن را آموزش می‌دهیم.

حال با این classifier آموزش دیده شده لیبل کامنت‌های test را با دادن فیچر مربوط به آن‌ها پیش بینی می‌کنیم. ار دستور predict استفاده می‌کنیم. بعد با دستور classification_report نتایج را بدست می‌آوریم که در بخش بعدی آورده شده است.

۱-۴. نتایج

برای بدست آوردن precision, recall, f1-score و accuracy از دستور classification_report که به مقایسه لیبل واقعی و لیبل پیش بینی شده می‌پردازد استفاده می‌کنیم.

نتایج مربوط به TF-IDF:

	precision	recall	f1-score	support
0	0.87	0.76	0.81	737
1	0.76	0.88	0.82	653
accuracy			0.81	1390
macro avg	0.82	0.82	0.81	1390
weighted avg	0.82	0.81	0.81	1390

نتایج مربوط به PPMI:

	precision	recall	f1-score	support
0	0.87	0.80	0.83	737
1	0.79	0.87	0.83	653
accuracy			0.83	1390
macro avg	0.83	0.83	0.83	1390

```
weighted avg          0.83        0.83        0.83
                                                1390
```

۵-۱. برسی نتایج

خب دقت روش PPMI بهتر از روش tf_idf شده است. برای PPMI دقت ۸۳ درصد داریم و برای tf-idf دقت ۸۱ درصد. دلیل بهتر بودن روش PPMI ما احتمالا برای این بوده که در بدست آوردن embedding آن لیبل را هم درگیر کردیم. برای همین اطلاعات بهتری داریم. اما در روش tf-idf اصلا به لیبل(سنتیمنت) کاری نداشتیم و این embedding اطلاعتش کمتر اهمیت دارد نسبت به PPMI.

نکته بعدی این است که در هر دو روش recall برای سنتیمت sad بالاتر بود اما precision برای کلاس happy بیشتر بوده. یعنی کلاس اگر کلاسی sad بوده به احتمال زیاد آن را درست تشخیص داده اما اگر کلاسی happy باشد با احتمال کمتری تشخیص می‌دهد. از آنطرف اگر مدل گفته باشد کلاسی happy است ما بیشتر مطمئن هستیم تا اینکه بگوید کلاسی sad است.

حالا چرا دقت این دو روش بیشتر نیست؟ یکیش اینکه شاید دیتا کافی نباشد، یک احنمال محتمل دیگر این است که مدل overfit کرده باشد. جون تعداد featureها زیاد است این امکان واقعا وجود دارد. اما نکته مهم دیگر خود لیبل گذاری داده‌ها است. ما برای نمونه لیبل چند داده تست را پیش بینی کردیم و کنار لیبل واقعیش گذاشتیم و به نکته جالبی برخودریم. در ادامه چند نمونه آورده شده است:

نمونه‌های روش tf-idf:

```
sentence:  [' ','خوب', 'اصلا', 'شده', 'استفاده', 'سوسیس
''نبود']
real label:  1
predict_label:  1
---***---
sentence:  ['کـه', 'نیست', 'اولی', 'دفعه', 'من
','مطمئنا', 'ولی', 'میگیرم', 'غذا', 'رستوران', 'ازاین'
'بود', 'بارم', 'آخرین'']
real label:  1
predict_label:  1
```

```
---***---
sentence:  [' ','شیرینی', 'خیلی', 'و', ''اصلا', 'طعم',
'بود', 'شیره', 'پر', 'نداشت', 'خوبی'']
real label:  1
predict_label:  1
---***---
sentence:  [' ','بار', ''اولی', 'بود', 'که', 'سفارش',
'میدادم', 'از', 'این', 'رستوران', 'لازانیا', 'خیلی',
'خیلی', 'معمولی', 'بود', 'گوشت', 'بو', 'میداد', 'غذا',
'فوق', ''العاده', 'چرب', 'بود', 'تا', 'حدی', 'که',
'وقتی', 'سرد', 'شد', 'دیگه', 'نمیشد', 'خوردش', 'کلا',
'کیفت', 'غذا', 'بد', 'نبود', 'ولی', ''اونقدرم', 'خوب',
'نبود', 'که', 'مجدد', 'به', 'این', 'رستوران', 'سفارش',
'درصد', 'تخفیف', 'تو', 'حتی', 'بدم', 'غذا'']
real label:  1
predict_label:  1
---***---
sentence:  [' ','با', 'یکساعت', 'و', 'دقیقه', 'تاخیر',
'شد', 'تحویل', 'جنس'']
real label:  0
predict_label:  1
---***---
sentence:  [' ','با', 'وجود', ''اینکه', 'تخفیف', 'داشت',
'بود', 'خوشمزه', 'و', 'کیفیت', 'با', 'ولی'']
real label:  0
predict_label:  0
---***---
sentence:  [' ','کیفیت', 'غذا', 'نسبت', 'به', 'قبلی',
'فرق', 'میکرد', 'برنج', 'تغییر', 'و', 'مرغ', 'مثل',
'گذشته', 'طعم', 'دار', 'نبود', 'متاسفانه', 'رستوران',
'هم', 'نزدیک', 'بود', 'و', 'هزینه', 'پیک', 'به',
'واقعا', 'نظرم', 'معقول', 'نبود', 'حیف'']
real label:  1
predict_label:  1
---***---
sentence:  [' ','هر', 'دو', 'عاالی', 'بود', 'سسی', 'که',
'همراه', 'بال', 'بود', 'و', 'با', 'کرفس', 'سرو',
'میشد', 'چندان', 'مطلوب', 'نبود', 'و', 'میتوانستن',
'خیلی', 'بیشتری', 'در', 'این', 'امر', 'خرج', 'دهند',
```

```
'در', 'کل', 'بسیار', 'مطلوب', 'بود', 'مزه', 'پیتزا',
'و', 'بال']
real label:  0
predict_label:  0
---***---
sentence:  [' ', 'غذای', 'از', 'تشکر', 'با', 'سلام',
'سپاسگزارم', 'نداشت', 'حرف', 'عالیتون']
real label:  0
predict_label:  0
---***---
sentence:  [' ', 'خوشمزه', 'و', 'تازه', 'خیلی', 'خیلی',
'زود', 'هم', 'خیلی', 'بودن', 'بودن', 'مخصوصا', 'عالی',
'ممنون', 'رسید', 'دستم', 'به']
real label:  0
predict_label:  0
```

نمونه‌های روش PPMI:

```
sentence:  [' ', 'شیرینیا', 'کردم', 'تاکید', 'اینکه', 'با',
'مونده', 'و', 'کهنه', 'کاملا', 'ولی', 'باشه', 'تازه',
'بود', 'بود', 'کیلو', 'ولی', 'گرم', 'توضیحات', 'تو',
'بود', 'خالی', 'جاهاییش', 'جعبه', 'و', 'شیرینی',
'چپه', 'که', 'چجوری', 'نیس', 'معلوم', 'هم', 'اسنپ',
'من', 'جعبه', 'ب', 'مالیده', 'تزئیناتش', 'و', 'و',
'روی', 'شیرینی', 'ببرم', 'میخاستم', 'مهمونی', 'برای',
'دوباره', 'شدم', 'مجبور', 'ولی', 'دادم', 'پول', 'تومن',
'اصلا', 'نبودم', 'راضی', 'اصلا', 'بخرم']
real label:  1
predict_label:  1
---***---
sentence:  [' ', 'سفارش', 'که', 'دستکشی', 'و', 'کروسان',
'من', 'که', 'آوردن', 'برند', 'از', 'رو', 'دو', 'هر',
'نداشتم', 'دوست']
real label:  0
predict_label:  1
---***---
sentence:  [' ', 'توضیحات', 'قسمت', 'است', 'قرار', 'اگر',
'و', 'کاربردش', 'پس', 'نادیده', 'من', 'همیشه', 'سفارش',
'از', 'سفارش', 'هربار', 'من', 'چیه', 'برای', 'وجودش',
```

'مختلف', 'در', ''این', 'قسمت', 'تاکید', 'که', 'نوشابه',
'اسپرایت', 'یا', 'زرد', 'باشد', 'اما', 'همیشههههههه',
'مشکی', 'کوکا', 'کولا', 'یا', 'پپسی', 'و', 'همیشه',
'باز', 'نکرده', 'در', 'سطل', 'لطفا', 'لطفا', 'لطفا',
رو', 'پیگیری'', 'کنید', ''این', 'موضوعات',
real label: 1
predict_label: 1
---***---
sentence: [' ', 'غذا'', 'روی', 'جدا', 'که', 'مالیاتی',
نمیشم', 'متوجه', 'رو', 'میشه', 'گرفته'']
real label: 0
predict_label: 1
---***---
sentence: [' ', 'ارزش'' 'اصلا', 'سالاد', 'قیمت', 'ب', 'نسبت',
ندارہ', 'خرید'']
real label: 1
predict_label: 1
---***---
sentence: [' ', 'با', 'نظرم', 'به', 'باید', 'بیشتری', 'سس',
'غذا'', 'فرستاده', 'بشه', 'در', 'کل', 'غذا'', 'عالیه',
'ولی', 'ارسال', 'کمی', 'زیاد', 'طول', 'کاملا',
رو', 'پیشنهاد'', 'این', 'رستوران',
real label: 0
predict_label: 0
---***---
sentence: [' ', 'عالی', 'و', 'بود', 'خوشمزه', 'خیلی',
'تازه', 'ساندویچش', 'چیپس', 'هم', 'داره', 'سسش', 'هم',
بود', 'عالی', 'بسیار', 'است', 'خوشمزه'']
real label: 0
predict_label: 0
---***---
sentence: [' ', 'هویج', 'بستنی', 'تاخیر', 'افتضاح',
'سفارش', 'دادم', 'اما', 'به', 'جاش', 'چیز', 'فرستادن',
'سیگار', 'اولترا', 'میخواستم', 'اما', 'چیزی', 'دیگه',
شد', 'ارسال''']
real label: 1
predict_label: 1
---***---
sentence: [' ', 'شکلاتی', 'شیری', 'بستنی', 'جای', 'به',
فرستادن', 'شیری', 'بستنی'']

```
real label:  1
predict_label:  1
---***---
sentence:  [' ,'سیر' ,'کوچولوعه' ,'منتها' ,'عالیه
''نمیشی']
real label:  0
predict_label:  0
```

برای اینکه بفهمیم چرا دقت مدل بیشتر نیست رفتیم سراغ جملاتی که لیبل آن‌ها اشتباه پیش بینی شده است. مثل جملات زیر که در بالا آورده شده است:

```
sentence:  [' ,'سفارش' ,'که' ,'دستکشی' ,'و' ,'کروسان
'من' ,'که' ,'آوردن' ,'برند' ,'از' ,'رو' ,'دو' ,'هر'
''دوست' ,'نداشتم']
```

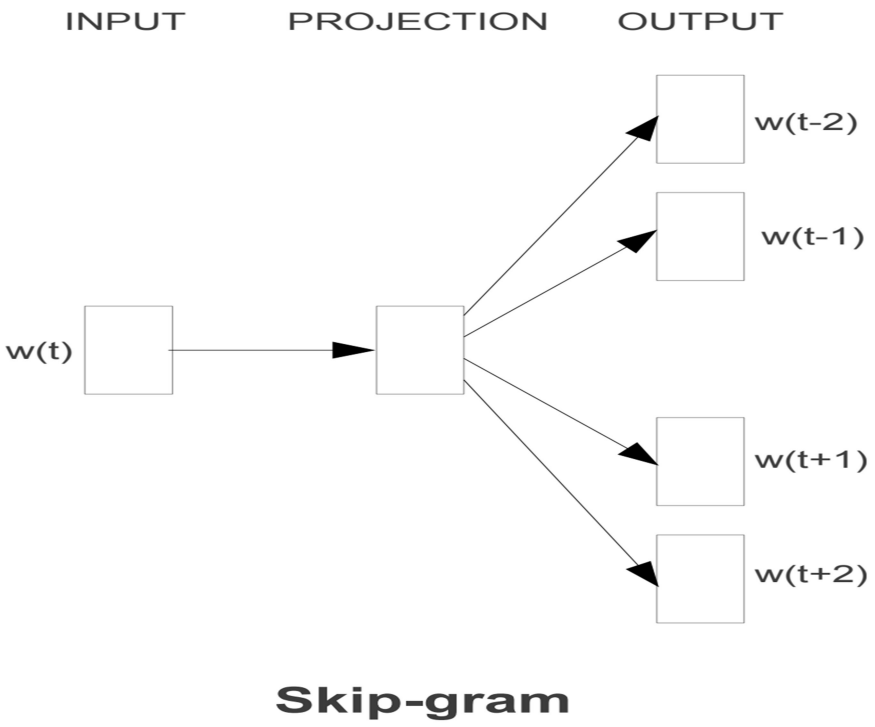
```
sentence:  [' ,'تاخیر' ,'دقیقه' ,'و' ,'یکساعت' ,'با
''جنس' ,'تحویل' ,'شد']
```

این دو جمله را ما sad پیش بینی کردیم اما لیبل واقعی آن‌ها صفر یعنی happy بوده است. اگر خود شما بخواهید لیبل بزنید حتما sad را انتخاب می‌کنید. این یعنی اینکه تعداد زیادی از داده‌ها اشتباه برچسب گذاری شده‌اند ولی مدل ما درست عمل کرده است اگرچه در نگاه اول دقت مدل گمراه کننده است. پس بخش از دلیل کامل نبودن دقت ما مربوط به برچسب گذاری اشتباه دادگان است.

۲. سوال ۲- پیاده سازی Skipgram

۲-۱. الف)

در این تمرین ما برای پیاده سازی word2vec از معماری skip-gram با negative sampling استفاده می‌کنیم.



شکل ۲. معماری skip-gram

حسن این روش نسبت به CBOW در سریع‌تر بودن آموزش آن است. در این روش ما کلمه‌ای را به مدل می‌دهیم و باید کلمات اطراف آن را پیش بینی کند. با این روش می‌تونیم شبکه را برای بازنمایی کلمات آموزش دهیم که کلمات با context یکسان، بردار embedding مشابه داشته باشند.

پیش پردازش: تابع preprocess

تمام کلماتی که کمتر از پنج بار در دیتاست آمده‌اند حذف می‌کنیم. این کار تاثیر نویز را کاهش می‌دهد و کیفیت بازنمایی بردار را افزایش می‌دهد. همچین حروف بزرگ را به کوچک تبدیل می‌کنیم.

تعداد کلمات قبل پیش پردازش: ۳۲۲۶

تعداد کلمات بعد پیش پردازش: ۸۰۳۹

دیکشنری‌ها: تابع create_lookup_tables

دو دیکشنری در این تابع پیاده سازی شده است. یک دیکشنری کلمه را می‌گیرد و ایندکس آن را برمی‌گرداند و یکی دیگر برعکس. از این تابع برای ساخت این دو دیکشنری برای کلمات corpus استفاده می‌کنیم.

Subsamplimg:

برخی کلمات مثل ’for‘ یا ’a‘ برای کلمات همسایه خیلی context خوبی فراهم نمی‌کند. پس حذف آن‌ها به بالا رفتن کیفیت تعبیه سازی کلمات کمک می‌کند و سرعت یادگیری را افزایش می‌دهد. به این فرآیند subsampling می‌گویند. برای هر کلمه w_i ما آن را با احتمال زیر حذف می‌کنیم:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

که $f(w_i)$ در اینجا تعداد تکرار کلمه در دیتاست است و t پارامتر حد آستانه است. پس در پیاده سازی برای تمام کلمات، تعداد تکرار هر کلمه را حساب می‌کنیم و از فرمول بالا احتمال

حذف آن را حساب می‌کنیم. بعد با کمک تابع random اگر عدد تصادفی تولید شده کمتر از این احتمال باشد آن را حذف می‌کنیم.

ساخت target ها: تابع get_target

حال دیتای ما مناسب برای دادن به مدل شده است. در معماری skip-gram برای هر کلمه در متن باید context اطراف آن را تعریف کنیم و تمام کلماتی که در پنجره به سایز C اطراف آن قرار دارند را بدست آوریم. یعنی $\frac{C}{2}$ کلمات قبلی و $\frac{C}{2}$ کلمات بعدی را به عنوان همسایگان کلمه داده شده در نظر می‌گیریم. که ما سایز C را ۴ درنظر گرفتیم. پس دو کلمه قبل و بعد جزو context حساب می‌شوند. به طور مثال:

Input: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Target: [3, 4, 6, 7]

که Input رشته ورودی است و Target کلمات همسایه که با کلمه داده شده که اینجا 5 است دوتا فاصه دارند.

ساخت batchها: تابع get_batches

حالا باید تابعی بسازیم که برای مدل batch از ورودی و تارگت بسازد. که ورودی کلمات داخل corpus هستند و تارگت کلمات همسایه آن هستند که می‌توان برای بدست آوردن آن‌ها از تابع get_target کمک گرفت. برای این کار اولا فقط batchهای کامل را در نظر می‌گیریم یعنی batch آخر اگه سایز کافی را نداشته باشد حذف می‌شود. بعد برای هر batch به ازای تمام کلمات آن همسایگان آن را بدست می‌آوریم.

مثال: اگر ورودی متن ما از ۰ تا ۹ باشد، سایز batch برابر ۴ باشد و سایز پنجره ۴ باشد داریم:

Batch 1:
X: [0, 0, 1, 1, 1, 2, 2, 2, 3, 3]
Y : [1, 2, 0, 2, 3, 0, 1, 3, 1, 2]

Batch 2:
X : [4, 4, 5, 5, 5, 6, 6, 6, 7, 7]
Y: [5, 6, 4, 6, 7, 4, 5, 7, 5, 6]

دقت شود که batch سوم چون سایزش ۲ است دور انداخته شد. همچنین context فقط درون batchها و مستقل از هم درنظر گرفته می‌شود. مثلا عدد ۳ همسایگی ۲و ۱ را فقط در نظر می‌گیریم و ۴، ۵ چون در batch دیگری هستند در نظر نمی‌گیریم. شاید فکر کنید این خیلی بد است اما چون سایز batchها خیلی بزرگ یعنی ۵۱۳ درنظر گرفته شده است و سایز پنجره هم کوچک است این عیب بسیار ناچیز و قابل چشم پوشی است و نسبت به پیچدگی‌هایی که برای درست کردن آن لازم است انجام شود قابل تحمل است. به طور دقیق کلا ۲ کلمه انتهایی و ابتدایی در ۵۱۳ کلمه حداکثر نصف context را از دست می‌دهند ولی ۵۰۸ کلمه دیگر context کامل دارند.

ارزیابی: تابع cosine_similarity

برای اینکه ببینیم مدل ما در هر مرحله چقدر پیشرفت داشته علاوه بر loss می‌توان در هر مرحله شباهت کلمات corpus را با یک سری از کلمات سنجید و چندتای نزدیک آن را چاپ کرد. برای نزدیکی از کسینوس دو بردار استفاده می‌کنیم. یک راه خوب و جانبی برای ارزیابی مدل ما است.

$$similarity = \cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{\left|\vec{a}\right| \cdot \left|\vec{b}\right|}$$

Negative Sampling: تابع forward_noise در کلاس SkipGramNeg

در skip-gram که فقط از کلمات context استفاده می‌کنیم و بقیه کلمات را non-context در نظر می‌گیریم دو مشکل وجود دارد:

۱. برای هر نمونه training، فقط وزن‌های مربوط به کلمات context ممکن است تغییر معناداری کنند. درحالیکه در فرآیند back-propagation ما تلاش می‌کنیم تمام وزن‌های لایه‌های مخفی را آپدیت کنیم. وزن‌های مربوط به کلمات non-context بسیار اندک یا اصلا تغییر نمی‌کنند یعنی در هر مرحله آپدیت وزن‌ها اسپارس است.

۲. برای هر نمونه training، محاسبه احتمالات نهایی با استفاده از softmax هزینه‌بر است چون مخرج کسر شامل جمع امتیازات کل کلمات vocabulary برای نرمالیز کردن می‌شود.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}} \quad for\ j = 1, \dots, k$$

برای غلبه بر این دو مشکل به جای اینکه brute force عمل کنیم یعنی همه وزن‌ها را بخواهیم آپدیت کنیم، باید سعی کنیم تعداد وزن‌های آپدیت شده برای هر نمونه آموزشی را کاهش دهیم. که تکنیک مورد نظر برای این کار negative sampling است. بجای اینکه احتمال همسایه بودن را پیش بینی کنیم، سعی می‌کنیم احتمال اینکه دو کلمه همسایه هستند یا نیستند را پیش بینی کنیم.

مثلا در روش عادی احتمال P(word1| word2) را پیش بینی می‌کردیم اما در روش negative sampling احتمال P(1|<word1, word2>) را حساب می‌کنیم. همچنین برای ساده‌تر شدن مسئله به طور تصادفی تعداد محدودی از کلمات را به عنوان کلمات

negative برای آپدیت وزن‌ها انتخاب می‌کنیم که k بربر کلمات positive هستند. پس loss فقط برای کلمات context و negative عمل back progate رو انجام می‌دهد.

محاسبه loss مدل: کلاس NegativeSamplingLoss

تابع هدف:

$$-\log \sigma({u_{w_O}}^\top v_{w_I}) - \sum_{i}^{N} \mathbb{E}_{w_i \sim P_n(w)} \log \sigma(-{u_{w_i}}^\top v_{w_I})$$

که u_{w_O} همان بردار جانمایی است و v_{w_I} بردار زمینه است. عبارت اول تارگت‌های درست است و عبارت دوم تارگت‌های نویزی یا negative است. پس مدل سعی میکند برای کلمات که context هستند ۱ را پیش بینی کند و برای کلمات negative که non-context هستند ۰ را پیش بینی کند.

آموزش مدل: بخش Trainng

برای آموزش مدل از کتابخونه پایتورچ کمک می‌گیریم. اول کورپوس را می‌دهیم به get_batches تا برای ما همانطور که قبلا توضیح دادیم داده مناسب train تولید کند. و روی آن حلقه می‌زنیم. هر بار محاسبات را روی یک batch انجام می‌دهیم. ابتدا برای بردار ورودی embedding را با کمک تابع forward_input در کلاس SkipgramNeg حساب می‌کنیم. بعد برای بردار خروجی embedding را با کمک تابع forward_output از همان کلاس حساب می‌کنیم. و آخر بردار کلمات negative را به forward_noise می‌دهیم تا embedding آن را برگرداند. حالا loss را با کمک تابع forward کلاس NegativeSamplingLoss و با دادن embedding سه بردار بدست می‌آوریم. سپس عمل backward propagation و optimization را انجام می‌دهیم تا وزن‌ها آپدیت شوند.

هایپرپارامترها و سایر تنظیمات:

نام هایپرپارمتر	مقدار هایپرپارامتر
تعداد ایپاک	20
بهینه ساز	Adam
سایز embedding	300
نرخ یادگیری	0.003
سایز پنجره همسایگی	4
تعداد نمونه منفی برای هر نمونه	4

نکته: برای تولید کلمات negative از توزیع unigram استفاده می‌کنیم اما برای اینکه کلمات نادر تاثیرشان ناچیز نباشد از فرمول زیر برای smoothing احتمالات استفاده می‌کنیم تا شانس انتخاب کلمات نادر کمی بیشتر شود:

$$P_n(w) = (\frac{U(w)}{Z})^{3/4}$$

که U(w) توزیع یکنواخت کلمات است.

همچنین بعد از هر چند نمونه یعنی ۱۵۰۰تا به طور تصادفی برای برخی کلمات ۶ کلمه نزدیک به آن‌ها که با فاصله کسینوسی حساب میشه را نمایش می‌دهیم. در کنار آن مقدار loss را هم نشان می‌دهیم.

اجرا:

بعد از ایپاک ۲۰ام مقدار loss برابر 1.67 شده است. که در مقایسه با ایپاک اول که 6.62 است به طور قابل ملاحظه‌ای کاهش یافته و این یعنی یادگیری اتفاق افتاده است.

ترکیب وزن‌های ماتریس جانمایی و زمینه برای embedding:

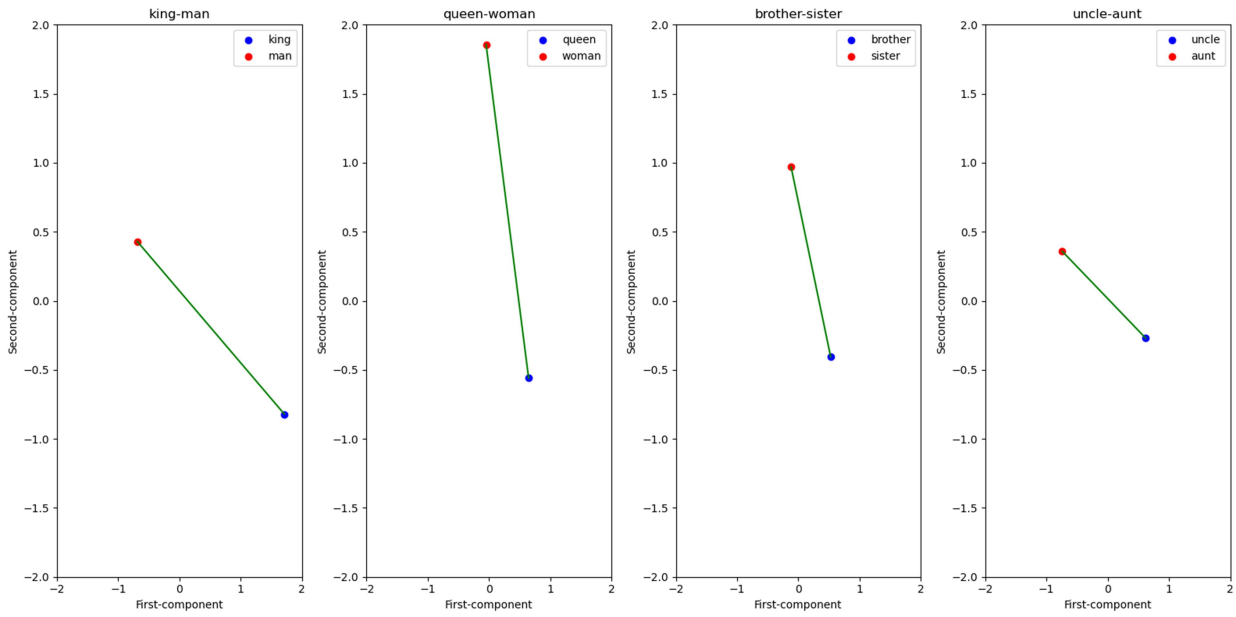
بعد از اینکه مدل ما آموزش دید دو ماتریس جانمایی و زمنیه که در کد به نام‌های in_embed و out_embed تعریف شده‌اند با هم جمع می‌شوند که بردار embedding نهایی را بسازند.

۲-۲. ب)

در این مرحله طبق خواسته سوال می‌خواهیم بردار تفاضل جفت کلمات مشخص شده را در نمودار دو بعدی نمایش دهیم.

خب الان هر کلمه ما شامل ۱۰۰تا مقدار است. پس آن را نمی‌توان در نمودار دو بعدی نمایش داد. پس لازم است که برای هر کلمه عمل کاهش ابعاد را انجام دهیم. یکی از محبوب‌ترین روش‌ها برای این کار روش PCA است. رویکرد کلی آن هم این اینکه ترکیب‌های خطی از فیچرهای مختلف داده را جستجو می‌کنه و اونی را انتخاب میکنه که بیشترین واریانس را در داده‌ها پوشش دهند. برای استفاده از PCA از ماژولی به همین نام در کتابخونه Sklearn استفاده می‌کنیم. و به عنوان ورودی تعداد کامپوننت به آن ۲ را می‌دهیم و بعد داده‌ها را transform می‌کنیم به فضای دو بعدی.

حال برای هر جفت کلمه مقدار embedding کاهش یافته آن‌ها را حساب کرده و از هم کم می‌کنیم تا بردارهای تقاضل آن‌ها بدست آید. بعد بردار تفاضل‌ها را همراه خود کلمات در فضای دو بعدی جانمایی می‌کنیم. که در شکل زیر نتیجه آورده شده است:



شکل ۳. بصری سازی بردار تفاضل کلمات

تحلیل:

شاید این نمودار دقیقا شبیه چیزی که انتظار داریم نباشد یعنی مثلا فاصله پادشاه با مرد دقیق اندازه فاصله ملکه با زن باشد. دو علت برای این موضوع می‌توان حدس زد:

۱. اندازه corpus کم است، کلا نزدیک ۸۰۰۰ کلمه یکتا داریم. درحالیکه مدل‌های embedding که استفاده می‌شود روی چندین میلیون کلمه آموزش می‌بینند و نباید انتظار داشت embedding ما روی دیتاست خیلی کوچک کلمات بتواند مانند embedingهای موجود کلمات را به صورت عالی تفکیک کند.

۲. علاوه بر کوچک بودن corpus نوع متن شکسپیر هم ادبی است. این یعنی کلمات خیلی کم تکرار می‌شوند نسبت به متن‌های رایج. خب این باعث می‌شود مدل سخت‌تر context مشابه برای کلمات پیدا کند. و این مشکل کوچک بودن دیتا را وخیم‌تر می‌کند.

حالا البته باز هم مدل ما یکسری ویژگی‌های خوب دارد که از روی شکل می‌توان دریافت که در ادامه به آن‌ها اشاره می‌کنیم:

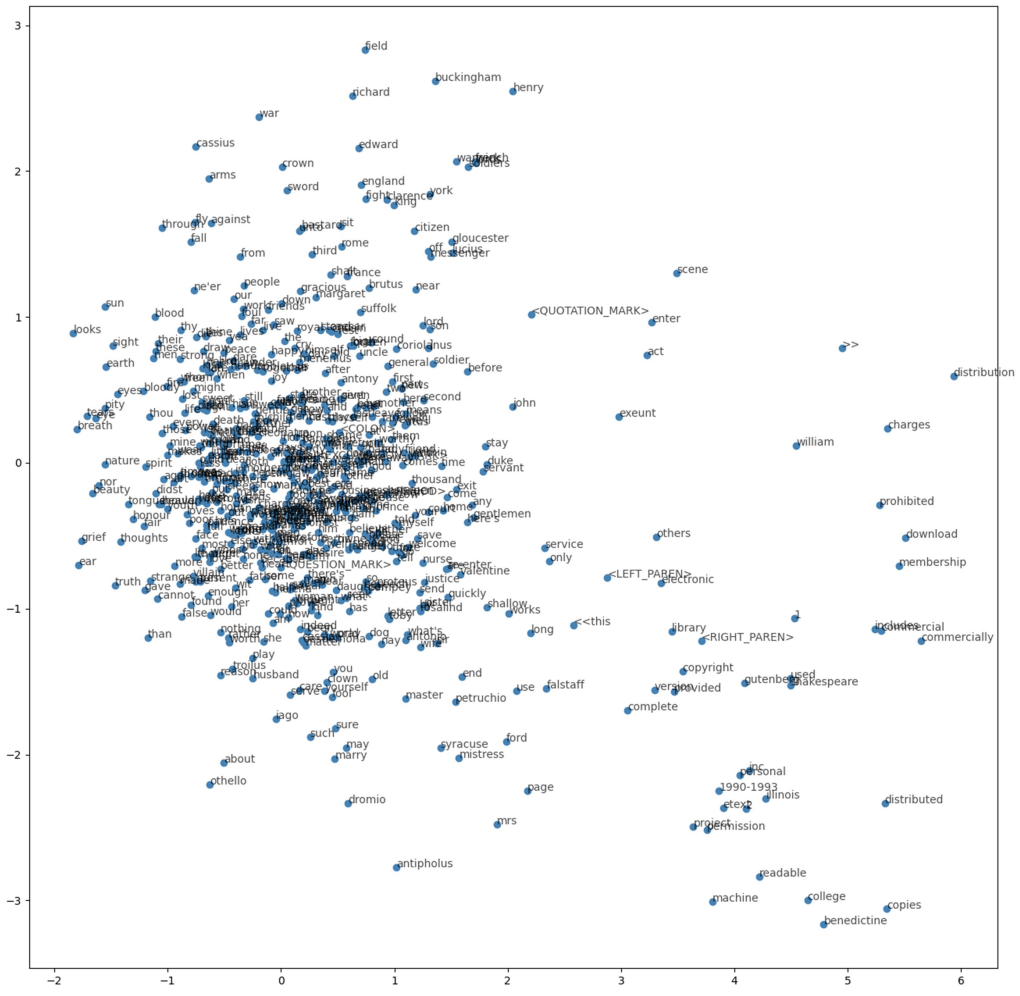
اگر به نمودارها دقت کنید جنس مونث پایین و جنس مذکر بالا قرار می‌گیرد. یعنی خواهر و عمه بالاتر از برادر و عمو قرار گرفته اند.

نکته بعدی اینکه جهت فاصله‌ی پادشاه با مرد و ملکه و زن یکسان است یعنی مرد بالاتر از پادشاه قرار دارد و زن هم بالاتر از ملکه قرار دارد. پس جهت هر دو بردار رو به پایین است. گرچه این جهت و اندازه بردار تفاضل دقیقا یکسان نیست.

مورد بالا برای نسبت جفت‌های خواهر-برادر و عمه-عمو هم صدق می‌کند.

یه استنباط دیگر که البته شاید درست نباشد این است که گفتیم جانمایی کلمه پادشاه و ملکه نسبت به مرد و زن پایین‌تر است. و خود پادشاه هم از ملکه پایین‌تر است و مرد هم از زن پیایین‌تر است. شاید بشه استدلال کرد که هرچه طبقه اجتماعی بالاتر باشد بازنمایی کلمه پایین‌تر قرار بگیرد. یعنی به ترتیب: پادشاه – ملکه – مرد – زن. با توجه به اینکه این متن در قرون وسطا نوشته شده است و معمولا آن زمان تبعیض جنسیتی بیشتر بوده است. ممکن است این تصادفی باشد چون گفتیم که دیتا به اندازه کافی بزرگ نیست.

در آخر هم embedding کل کلمات را در فضای دو بعدی نمایش دادیم که در ادامه آمده است:



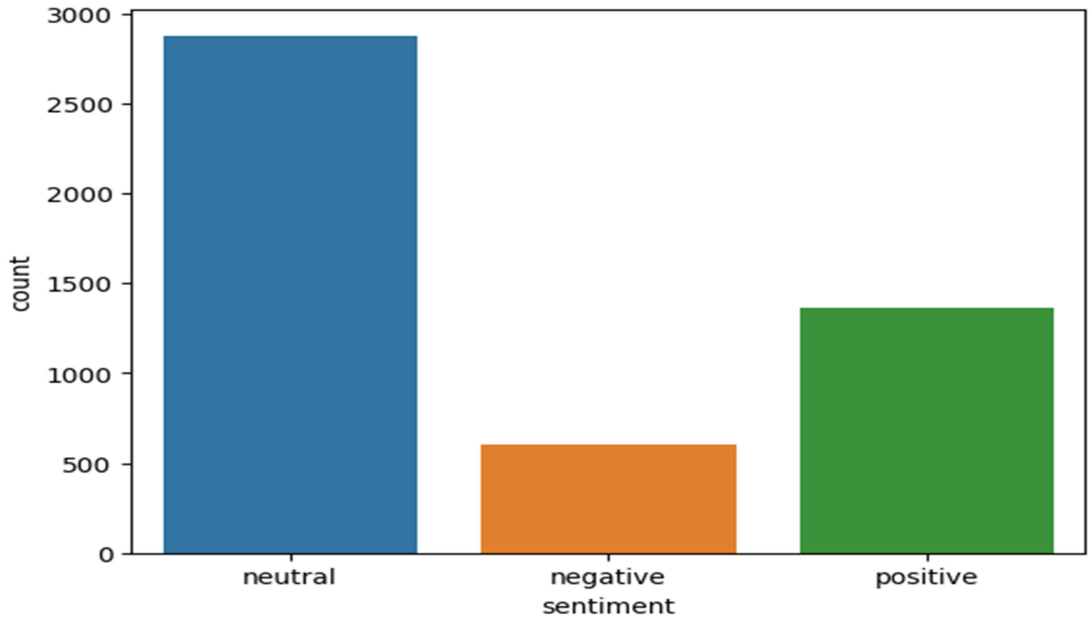
شکل ۴. بازنمایی کلمات در دو بعد

از این شکل هم می‌توان نتیجه گرفت که مدل به خوبی کلمات را تفکیک نکرده و کلمات بیشتر در گوشه سمت چپ متمرکز شدند. در حالیکه که اگر بهتر عمل می‌کرد احتمالا پراکندگی باید بیشتر می‌شد. که دلیل این ضعف همان موارد گفته شده قبل‌تر است.

۳. سوال ۳- تحلیل احساسات برای دادگان FinancialPhraseBank

۳-۱. پیش پردازش:

ابتدا داده را با کمک کتابخانه pandas می‌خوانیم. سپس با استفاده از کتابخانه seaborn نمودار تعداد لیبل‌ها(مثبت، منفی، خنثی) را رسم می‌کنیم.



شکل ۵. نمودار تعداد کلاس‌ها

سپس سطر شماره ۲۹۸۳ را به دلیل اینکه قسمت متنش خالی است از دیتاست حذف می‌کنیم. و بعد از آن در تابع clean_text متن را به حالت استاندار و تمیز تبدیل می‌کنیم. این کار شامل حذف علائم غیر الفبا، تبدیل کاراکترها به شکل کوچک، جدا کردن کلمات، حذف کلمات stopword و متصل کردن کلمات باقی مانده به هم می‌شود.

۳-۲. الف)

۳-۲-۱. GloVe: یک روش تعبیه سازی کلمات است که آن را دانلود می‌کنیم. سپس از حالت zip خارج کرده. بعد از آن یک دیکشنری embedding از کلمات می‌سازیم. که شامل چهارصد هزار کلمه می‌شود.

حال جملات را ابتدا توکنایز می‌کنیم و بعد برای تمام جملات دیتاست باید بردار glove مربوط به کلمات آن را بدست آوریم. که برای این کار کافی است کلمات جمله را در دیکشنری embedding بخوانیم و مقدار آن را ذخیره کنیم. بعد از اینکه glove را برای همه کلمات جمله بدست آوردیم از آن‌ها میانگین می‌گیریم و آن را برمی‌گردانیم تا به عنوان feature برای classification استفاده کنیم.

۳-۲-۲. تقسیم دیتا: قبل از اینکه بخواهیم دیتا را به train و test تقسیم کنیم باید لیبل‌ها که از جنس categorical هستند را به مقادیر صحیح تبدیل کنیم تا بتوان به عنوان ورودی به مدل بدهیم. برای این کار از کتابخونه LabelEncoder استفاده می‌کنیم. که به لیبل neutral مقدار 1، به لیبل negative مقدار 0 و به لیبل positive مقدار 2 را نسبت داده است.

در مرحله بعد با ماژول train_test_split کتابخونه Sklearn داده‌ها را به دو بخش train و test تقسیم می‌کنیم. و البته این کار را با شافل انجام می‌دهیم. نسبت را هم ۹۰ به ۱۰ درصد قرار می‌دهیم.

۳-۲-۳. Logistic Regression: از کتابخونه Sklearn استفاده می‌کنیم و logisticRegression را فراخوانی می‌کنیم. و به آن دادگان train را می‌دهیم تا آموزش ببینند. پس از آن داده X_test را می‌دهیم تا لیبل آن‌ها را حدس بزند.

بعد از آن از classification_report برای مقایسه لیبل پیش بینی شده با لیبل واقعی استفاده می‌کنیم که این تابع precision, recall, f1-socre و accuracy را برمی‌گرداند.

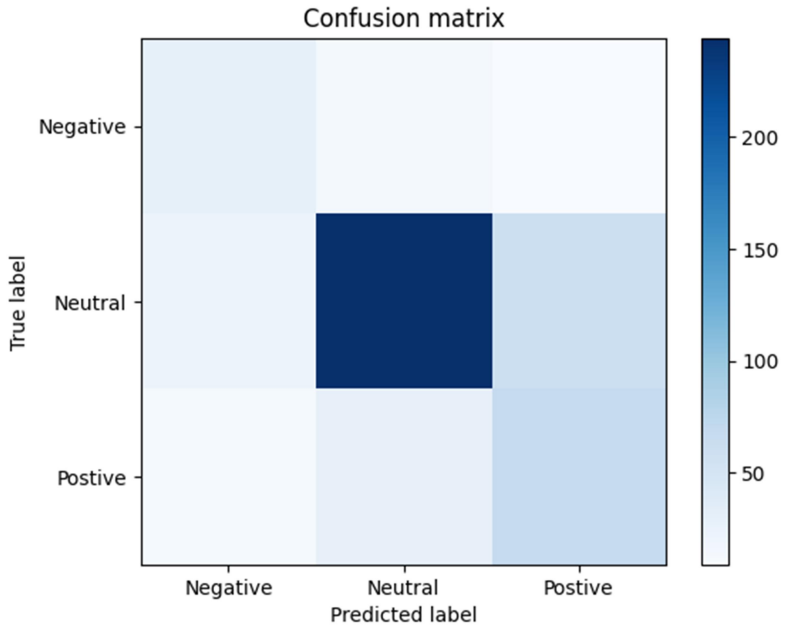
دقت پیش بینی مدل ما ۷۰ درصد شده است. که از baseline یعنی رندوم پیش بینی کردن که دقت آن ۳۳ درصد است خیلی بهتر است. اما دقت کنید چون کلاس‌ها نامتوازن هستند این معیار به تنهایی می‌تواند گمراه کننده باشد. باید به معیارهای ارزیابی در تک تک کلاس‌ها هم دقت کنیم. همچنین می‌توان از ROC هم استفاده کرد که چون سوال نخواسته انجام ندادیم اما معیار خوبی برای ارزیابی مدل در زمانی است که دیتا نامتوازن است.

خروجی این تابع در ادامه آورده شده است:

	precision	recall	f1-score	support
0	0.56	0.46	0.50	63
1	0.75	0.85	0.80	286
2	0.63	0.49	0.55	136
accuracy			0.70	485
macro avg	0.64	0.60	0.62	485
weighted avg	0.69	0.70	0.69	485

مقدار precision، recall و f1-score برای کلاس ۱ یا لیبل خنثی بیشتر است. دلیل آن احتمالا داده‌های بیشتر است که در بخش بعد بیشتر توضیح خواهیم داد.

برای اینکه بهتر بتوانیم عملکرد مدل را ارزیابی کنیم از confusion matrix هم استفاده کردیم که به شکل زیر است:



شکل ۶. ماتریس آشفتگی

همانطور که مشخص است داده‌های خنثی از همه بهتر پیش بینی شده‌اند. سپس داده‌های مثبت و در آخر داده‌های منفی. و مدل بیشتر داده ها را خنثی و بعد مثبت و در آخر منفی پیش بینی کرده است.

۳-۳. ب) مدل logistic regression هدفش این است که loss خودش را کمینه کند و این اتفاق زمان می‌افتد که بتواند فاصله احتمال پیش بینی‌اش را با خروجی واقعی کمینه کند. وقتی که دیتا ما نامتوازن می‌شود برای مدل بهتر است که روی داده‌های مربوط به لیبل فرآوان‌تر تمرکز کند. چون احتمال اینکه درست پیش بینی کند بیشتر می‌شود. مثلا تعداد داده‌های لیبل خنثی نزدیک پنج برابر داده‌های لیبل منفی است. خب طبیعی است که اگر مدل سعی کند تمایل داشته باشد که خنثی پیش بینی کند احتمالا loss کمتری داشته باشد.

۳-۴. ج) مدل naïve bayes به دلیل مستقل فرض کردن فیچرها از هم در مقابل دیتای نامتوازن غیر حساس است. همچنین چون سعی می‌کند احتمال کلاس‌ها را مستقل از هم پیش بینی کند به جای اینکه بخواد از مرز جدا کننده مشترک استقاده کند. برخلاف logistic regression که به دیتای متوازن حساس است. اما باید توجه شود که همین فرض فرض مستقل در نظر گرفتن فیچرها از قدرت مدل کم می‌کند. الان مدل ما برای هر ورودی یک بازنمایی ۱۰۰ تایی دارد که آن را به مدل می‌دهیم و این ۱۰۰ فیچر کنار هم معنا می‌دهند و زمانی که این‌ها را مستقل از هم فرض کنیم دیگر embedding معنای زیادی ندارد. برای همین در کل انتظار داریم که naïve bayes بدتر عمل کند با وجود اینکه در مقابل دیتای نامتوازن حساس است.

دقت کنید که پیاده سازی در جواب این سوال خیلی تاثیر دارد و ممکن است در نوع دیگر پیاده سازی مدل naïve bayes بهتر عمل کند.