دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده برق و کامپیوتر

گزارش تمرین شماره ۲

درس NLP

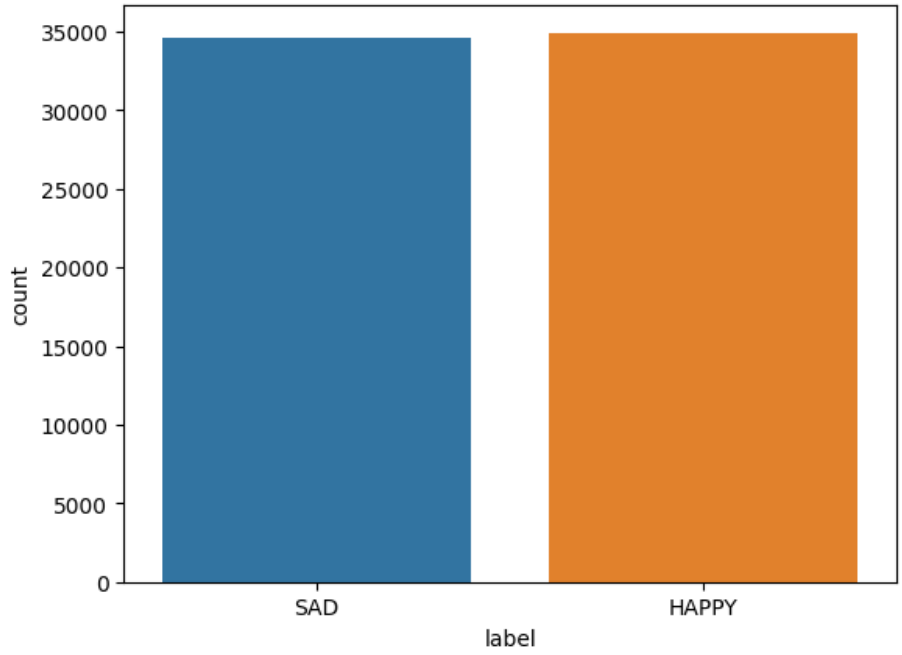
نام و نام خانوادگی
سپهر کریمی آرپناهی

شماره دانشجویی
۸۱۰۱۰۰۴۴۷

o سوال ۱

مرحله صفر :

در این سوال هدف ما تشخیص احساسات در مجموعه دادگان **snappfood** می باشد. برای این کار ابتدا مجموعه دادگان را می خوانیم و در **df** ذخیره می کنیم. سپس تمامی سطر های **null** را حذف می کنیم و سپس چک می کنیم که آیا سطری **null** مانده است یا خیر. در ادامه یک بار دو کلاس سطر **label** را پلات می کنیم که شیوه توزیع دو کلاس را بدست بیاوریم:



شکل ۱ : شیوه توزیع دو کلاس label در دیتاست

همانطور که میبینیم این مجموعه دادگان توزیع **balance** دارد. در ادامه با حفظ توزیع هر کدام از دسته ها ۲۰ درصد از داده ها را نگه داشته و بقیه داده ها را حذف می کنیم.

مرحله ۱:

حال می خواهیم دادگان را پیش پردازش کنیم. برای این کار تابع **text_preprocessing** را تعریف می کنیم در این تابع ابتدا با استفاده از کتابخانه **clean_text** چندین پیش پردازش بر روی متن انجام می دهیم که عبارتند از: ابتدا تمامی متن را به حروغ کوچک نبدیل کرده و تمامی خط فاصله ها، شماره تلفن ها و ایمیل ها را از متن حذف می کنیم. سپس با استفاده از کتابخانه **hazm**، ابتدا متن را نرمالایز کرده و کاراکتر های اضافه را حذف می کنیم. و همه داده های پیش پردازش شده را در ستون **cleaned_comment** ذخیره می کنیم.

مراحل۲:

در این مرحله می خواهیم روش **Tf-idf** را پیاده سازی کنیم. برای این کار ابتدا دادگان را مطابق آنچه در صورت سوال گفته شده به دو دسته **train** و **test** تقسیم می کنیم. سپس **dictionary** را از میان دادگان **train** استخراج کرده و ذخیره می کنیم. حال به سراغ پیاده سازی روش **tf-idf** می رویم:
برای پیاده سازی **tf-idf** یک کلاس تعریف می کنیم و نام آن را **TfIdfClassifier** می گذاریم. این تابع **vocabulary** را به عنوان ورودی از طریق تابع **set_vocab** می گیرد و سپس با دادن **x_train** و **y_train** به آن **tf_idf** را محاسبه می کند.
این کلاس از چند تابع تشکیل شده است که به ازای هر جمله یک بردار ویژگی خواهیم داشت که اندازه سایز آن به اندازه دیکشنری است. در این روش برای محاسبه **tf**، هر کلمه ای که در جمله وجود داشته باشد، در **index** کلمه آن در ماتریس اسپارسی که داریم عدد **idf** ان را قرار می دهیم. شرح دقیق هر کدام از توابع به شرح زیر می باشند:

1. `set_vocab(self, vocab)`:
    - This method sets the vocabulary for the classifier by taking a list of unique words as input.
    - It initializes the `model` attribute to an instance of `MultinomialNB` and the `idf` attribute to an empty dictionary.
2. `fit(self, train_input, train_output)`:
    - This method trains the classifier on the given training data by calculating the IDF values and vectorizing the data.
    - It takes two arguments, `train_input` and `train_output`, which are the input and output data, respectively.
    - It first calculates the IDF values for each word in the vocabulary by calling the `calculate_idf` method.
    - It then vectorizes the input data by calling the `vectorize_data` method, and trains the model on the vectorized data using the `fit` method of the `model` attribute.
3. `predict(self, test_input)`:
    - This method predicts the class labels for the given test data by calling the `predict` method of the `model` attribute on the vectorized test data.
    - It takes a single argument, `test_input`, which is the input data to predict on.
4. `vectorize_data(self, data)`:
    - This method vectorizes the input data by calculating the TF-IDF values for each sentence in the data.
    - It takes a single argument, `data`, which is the input data to vectorize.
    - It returns a 2D numpy array where each row corresponds to a sentence in the input data and each column corresponds to a word in the vocabulary.
5. `calculate_tf_idf(self, sentence)`:
    - This method calculates the TF-IDF values for the given sentence using the IDF values for each word in the sentence and the word counts (TF) for each word in the sentence.
    - It takes a single argument, `sentence`, which is the sentence to calculate the TF-IDF values for.

- It returns a list of TF-IDF values for each word in the vocabulary.

6. `calculate_idf(self, data)`:
   - This method calculates the IDF values for each word in the vocabulary using the input data.
   - It takes a single argument, `data`, which is the input data to calculate the IDF values from.
   - It first creates a dictionary, `word_in_doc_list`, which maps each word in the vocabulary to a list of sentences that contain that word.
   - It then calculates the IDF value for each word in the vocabulary by dividing the total number of sentences in the input data by the number of sentences that contain the word. The IDF values are stored in the `idf` attribute as a dictionary where each key is a word and each value is the IDF value for that word.

حال پس از فراخواندن classifier، و محاسبه y_pred، اطلاعات این طبقه بند را نمایش می دهیم:

```
tf-idf Accuracy: 0.7942446043165468
tf-idf Precision: 0.7905308464849354
tf-idf Recall: 0.7973950795947902
tf-idf F1 Score: 0.7942433263962219
```

مرحله۳:

حال در این قسمت می خواهیم روش PPMI را پیاده سازی کنیم. برای این کار باید ابتدا ماتریس co_occurance را پیاده سازی کنیم. برای این کار ابتدا vocabulary را می سازیم. سپس در ادامه ماتریس co_occurance را ساخته و با اندازه پنجره=۲ تعداد تکرار کلماتی که در این اندازه پنجره از کلمه دیگر امده است را در این ماتریس ذخیره می کنیم.

حال می خواهیم PPMI را محاسبه کنیم برای این کار، می دانیم اگر Z تعداد کل توکن ها و C ماتریس تعداد تکرار میان هر دو کلمه در همسایگی دو باشد، داریم:

$$P(i,j)=C(i,j)\sum_{k,l}C(k,l)=C_{ij}Z$$

$$P(i)=\sum_{k}C(i,k)\sum_{k,l}C(k,l)=Z_iZ$$

$$\text{PMI}(i,j)=\log P(i,j)P(i)P(j)=\log C_{ij}\cdot ZZ_i\cdot Z_j$$

$$\text{PPMI}(i,j)=\max(0,\text{PMI}(i,j))$$

پس ازمحاسبه PPMI، با استفاده از طبقه بند بیزین، دقت های این طبقه بند را با استفاده از روش PPMI می سنجیم که نتایج به شرح زیر است:

```
ppmi Accuracy: 0.7352092352092352
ppmi Precision: 0.7193211488250653
ppmi Recall: 0.7193211488250653
ppmi F1 Score: 0.73593259649943
```

مقایسه نتایج: خب همانطور که دیدیم روش tf-idf پیاده سازی شده مقداری بهتر از روش PPMI بهتر عمل کرده است که دلایل آن ممکن است به خاطر overfit کردن ppmi بر روی دیتاست ترین باشد که یکی از اشکال های شایع این روش است(ضعف generality). اما به طور کلی هر دو روش به نتایج قابل قبولی رسیدند که نزدیک به مدل های آماده می باشد.

o سوال ۲

در این سوال هدف تولید بردار های معنا (**Vector Semantic**) برای جانمایی کلمات یک دادگان با روش مشابه **word۲vec** است. مدلی که از آن برای تولید این بردار های معنا استفاده میکنید **Skipgram** است.

• الف

در قسمت اول سوال ابتدا دادگان را دریافت و ان را در **text** ذخیره می کنیم. سپس دادگان ا به عنوان ورودی به تایع پیش پردازش می دهیم. این تابع پیش پردازش های زیر را بر روی دادگان انجام می دهد:

تابع **preprocess** ابتدا کل دادگان را به عنوان ورودی دریافت می کند. سپس تمامی حروف را به حروف کوچک تبدیل کرده و اعداد و **stopword** ها را از دادگان حذف می کند. این کار باعث افزایش دقت مدل ما خواهد شد. سپس دادگان را جمله به جمله جدا کرده و **punctuation** ها را از تمامی جملات حدف می کنیم و توکن های ساخه شده از جملات را به عنوان خروجی بر میگردانیم.
همانطور که می بینیم پس از پردازش بر روی داده ها تعداد جملات و کلمه ها به شرح زیر خواهد بود:

```
Number of sentences: 78026
Vocabulary size: 28863
```

در قسمت دوم سوال مدل **skipgram** را پیاده سازی می کنیم. این مدل اندازه پنجره، تعداد نمونه های منفی، تعداد جملات و جملات را به عنوان ورودی می گیرد و جملات منفی و مثبت(**negative_sentences, positive sentences**) را به عنوان خروجی می دهد.
در ادامه اندازه پنجره را برابر با ۲ و **k** را برابر با ۴ قرار می دهیم. سپس تابع **skipgram** را صدا زده و نمونه های مثبت و منفی را تولید می کنیم.
در ادامه ماتریس های جانمایی و زمینه را می سازیم و در ۱۳ ایپاک، و با نرخ یادگیری برابر با ۰.۰۰۱، هر بار ماتریس ها را اپدیت می کنیم و در انتها دو ماتریس را با هم جمع کرده و بردار ویژگی را می سازیم.

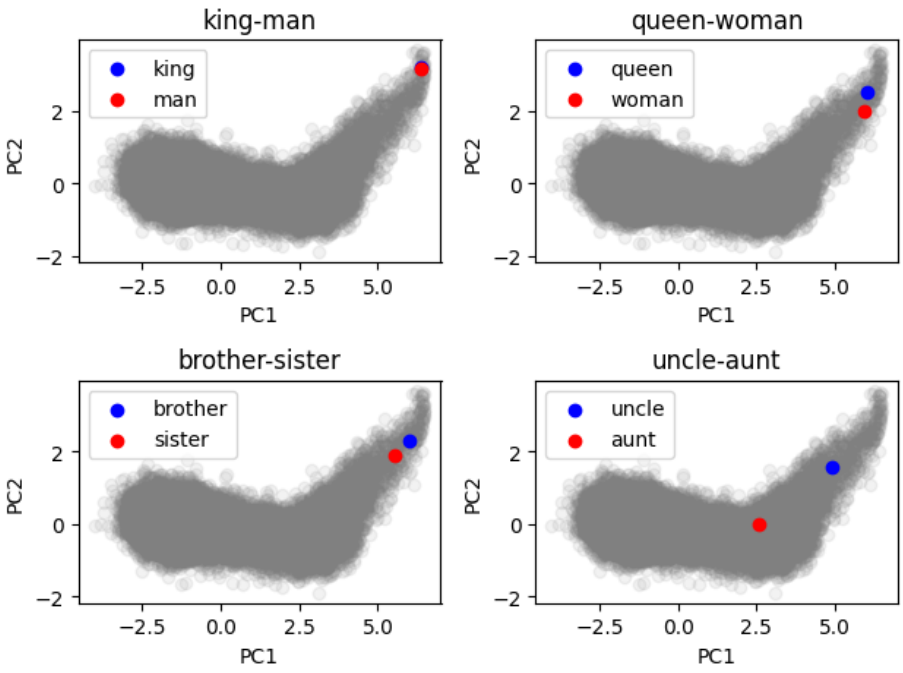
• ب

در این قسمت می خواهیم با استفاده از بردار ویژگی بدست آمده در قسمت قبل، با استفاده از تبدیل **PCA** و **fit_transform** بردار ویژگی کلمات را در دو بعد تصویر کنیم و سپس چهار بردار تفاضل گفته شده را رسم کنیم.

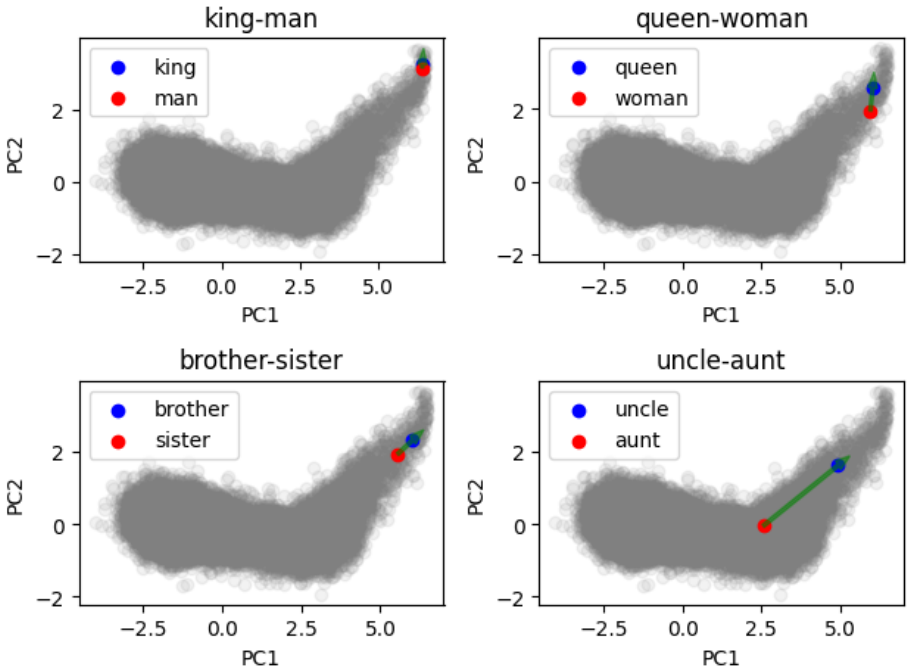
همانطور که دیده می شود، دو بردار تفاضل **king-man** و **queen-woman** و همچنین دو بردار تفاضل **brother-sister** و **uncle-aunt** تقریبا موازی هستند که همان هدفی بود که ما در این سوال دنبال آن بودیم. زیرا همانطور که می دانیم مثلا برای ساختن **queen** به صورت زیر عمل می کنیم:

Queen = (king-man) + woman

و این موجب می شود که دو بردار تفاضل اول با هم موازی باشند که همان نتیجه ای است که ما در این قسمت به آن رسیدیم.



شکل ۲ : نمایش بردار ویژگی کلمات گفته شده



شکل ۳ : نمایش بردار تفاضل به همراه کلمات گفته شده

o سوال ۳

همانطور که در صورت سوال گفته شده، در این سوال می خواهیم یک مسئله تشخیص إحساس برای مجموعه دادگان داده شده را حل کنیم.

- **الف:**

مرحله اول:

در مرحله اول ابتدا باید پیش پردازش های لازم را بر روی این مجموعه دادگان انجام دهیم. برای این کار ابتدا در میان مجموعه دادگان فابل **allData.csv** را میخوانیم. و در دیتاست **df** ذخیره می کنیم. همانطور که می بینیم این دیتاست از ۴۸۴۶ سطر تشکیل شده است. همچنین این دیتاست از ۲ ستون تشکیل شده است که ستون اول **sentiment** و ستون دوم خود متن جمله(**text**) میباشد که هرکدام از این سطر ها به یک جمله و احساس مربوط به آن مربوط می باشد.
سپس به سراغ انجام پیش پردازش های لازم بر روی دیتاست می رویم.

پیش پردازش های لازم را در تابع **clean_text** تعریف کردیم. این تابع هرکدام از سطر های دیتاست را به عنوان ورودی گرفته و ابتدا تمامی **URL** ها، منشن ها و هشتگ ها را از جملات حذف می کنیم. سپس کاراکتر های **non-alphabetic** را از جملات حذف می کنیم. در ادامه جملات را توکنایز کرده و **stopword** ها را از آن ها حذف می کنیم. حال دوباره تمامی متن پیش پردازش شده را در ستون **text** باز نویسی می کنیم.

مرحله دوم:

در این مرحله میخواهیم با استفاده از بردار های معنای **GloVe** کلماتمان را بازنمایی کنیم.
GloVe در نسخه **۶B** بر روی **۵ Gigaword** و **۲۰۱۴ wikipedia** ساخته شده است که شامل ۶ بیلیون توکن و ۴۰۰ هزار **vocab** می باشد و ما در اینجا از نسخه **۱۰۰d** که به معنی بازنمایی های با ابعاد ۱۰۰ می باشد استفاده کردیم.

پس ابتدا با استفاده از دستور **wget** و **unzip** آن را دانلود کرده و بازنمایی ها را استخراج می کنیم. سپس یک ارایه از کل **embedding** ها می سازیم و نام آن را **glv_embedding** قرار می دهیم. حال برای آن که برای هر جمله یک بازنمایی استخراج کنیم تابع زیر را تعریف می کنیم:

get_sentence_embedding(**sentence**, **embeddings**):

این تابع با گرفتن هر جمله و وکتور های بازنمایی (در اینجا **glove.۶B**) برای آن جمله یک بازنمایی استخراج می کند. این تابع به این صورت عمل می کند که در یک حلقه برای تمام کلمات موجود در جمله که در وکتور های بازنمایی اماده وجود دارند، باز نمایی ها را استخراج

کرده و با بازنمایی کلمات قبلی جمع می کند و در انتها از میان تمامی بازنمایی ها میانگین می گیرد و آن را به عنوان بازنمایی جمله به خروجی می دهد.

در ادامه در یک حلقه بر روی همه سطر های دیتافریم، کل داده های هر سطر را توکتایز کرده و به حروف کوچک تبدیل می کنیم. سپس هر سطر را به همراه بازنمایی های آماده در تابع **get_sentence_embedding** فراخوانی می کنیم و خروجی بازنمایی آن سطر را یک آرایه **vec_embeddings** ذخیره می کنیم.

مرحله سوم:

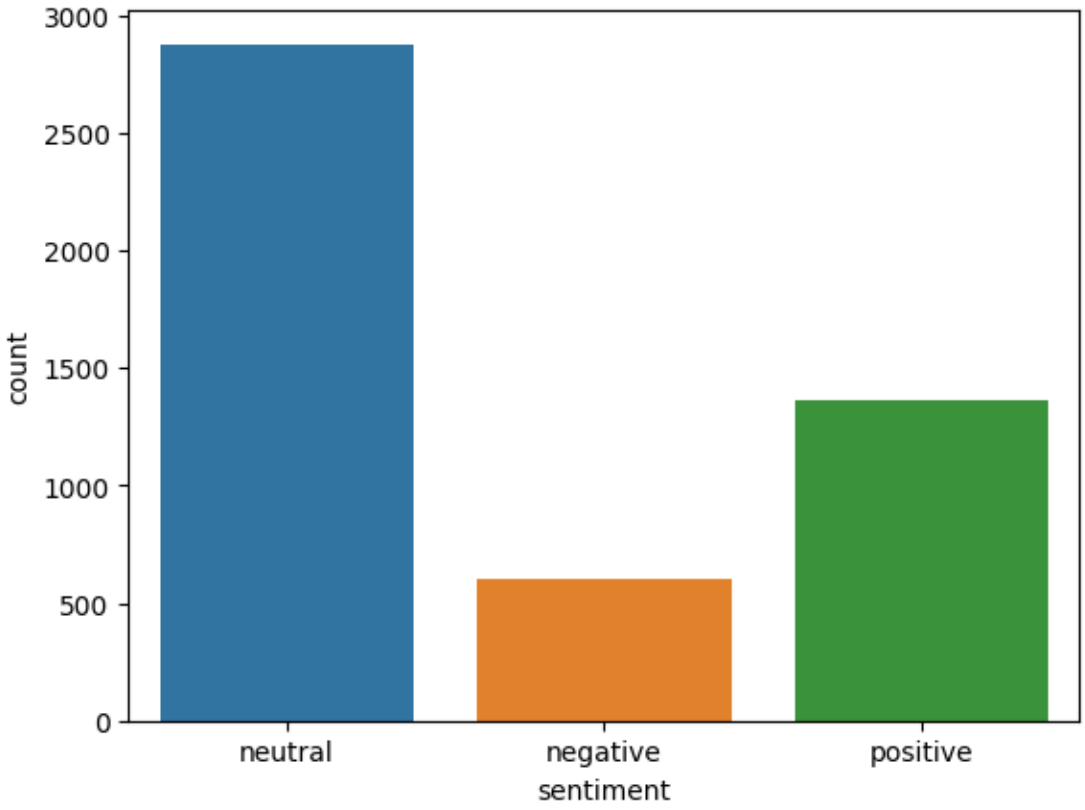
حال در این مرحله ابتدا داده های بدست امده را به دو دسته **train** و **test** تقسیم می کنیم.(مطابق انچه در صورت سوال گفته شده ۹۰ درصد به ترین اختصاص می دهیم.) سپس با استفاده از کتابخانه آماده **sklearn**، داده های آموزش را به **logisticRegression** فیت کردیم. سپس آمتیاز های بدست آمده را نمایش دادیم :

	precision	recall	f1-score	support
-1	0.68	0.45	0.54	55
0	0.75	0.89	0.81	287
1	0.66	0.50	0.57	143
accuracy			0.72	485
macro avg	0.70	0.61	0.64	485
weighted avg	0.71	0.72	0.71	485

دقت ما در این مدل برابر با ۷۲ درصد شده است که دقت خیلی خوبی برای این دادگان می باشد. همچنین همانطور که می بینیم این مدل در همه معیار ها(**precision, recall and f1 score**) برای کلاس ۰ یا همان داده های خنثی بهتر عمل می کند. علت این امر آن است که تعداد داده های یادگیری این کلاس بسیار بیشتر بوده و مدل ما بر روی آن بهتر عمل یادگیری را انجام می دهد.

- ب:

برای توضیح این قسمت ابتدا توزیع دادگان در کلاس های مختلف را رسم می کنیم:



شکل ۴ : توزیع کلاس ها در دادگان

همانطور که می بینیم کلاس خنثی یا ۰، تعداد نمونه های بسیار بیشتری نسبت به سایر کلاس ها دارد و همانطور که در قسمت قبل نیز دیدیم، توزیع نامتوازن کلاس ها باعث می شود عمل یادگیری بر روی کلاسی که تعداد نمونه های بیشتری دارد بهتر انجام شود و مدل به دقت بالاتری در آن کلاس دست یابد.
پس نتیجه می گیریم در دادگان با کلاس های نامتوازن، باعث می شود که تشخیص کلاس ها برای مدل سخت شود زیرا مدل کلاس با تعداد بیشتر را با دقت بالایی یاد گرفته است و به همین علت بر روی کلاس اقلیت عملکرد ضعیفی خواهد داشت و دلیل آن بالاس شدن مدل به سمت کلاس اکثریت می باشد.

- **ج:**

در کلاس **naïve bayes** از آن جایی که بر روی ویژگی ها فرض **independence** دارد باعث می شود مدل ما توزیع ویژگی ها را بهتر یاد بگیرد(حتی در دادگان نامتوازن). اما در مقابل در مدل **logistic regression** چون این فرض را نداریم کمی نسبت به دادگان نامتوازن حساس تر خواهیم بود. البته برای اینکه نظر دقیق بتوانیم در این باره بدهیم باید پیاده سازی کنیم و نتایج دو مدل را با هم مقایسه کنیم.

در انتهای نوت بوک این سوال روش **naïve bayes** نیز پیاده سازی شده است و همانطور که پیشبینی می شد کمی دقت آن بهتر از روش **logistic regression** بود.