Machine Learning Engineer Nanodegree

Capstone Report

Patrick O'Sullivan
August 01st 2019

Business Public Sentiment

Definition

Project Overview

Businesses are in every aspect of our lives from the moment we are born (and earlier) to the moment we die (and after). They have huge control over us and can manipulate us in many ways (and often do). Several businesses have built platforms that allow their customers rate their one-off experiences with a business but what about the overall sentiment of a business.

Understanding the overall sentiment of a business may help us make a more informed decision about which business we want to use for a given service and hence encourage businesses to be more conscience and pro-active about their public sentiment. It would also allow other businesses to decide which businesses they wish to partner with or provide/receive services to/from.

Problem Statement

The main objective of the project will be to use Machine Learning to decide the sentiment of text. When given a string of text I want to be able to say whether the sentiment of the text is considered positive or negative. If I can build a model that can accurately say whether a string of text is positive or negative, I can then take live data feeds for various companies (from twitter or other sources) and track the public sentiment over time.

I will create models that I will train with data that has already been labeled as positive or negative (the Sentiment140 data-set) using both Naïve Bayes and SVM.

Metrics

Once I have predictions I can evaluate the model to check various scores for my models. There are several ways to evaluate my models.

For this project I don't want businesses with positive sentiment to be labeled with negative sentiment, so I need a high Recall score. I also want to minimize the number of business with negative sentiment been labeled with positive sentiment, so I'd like a good precision score. But I will priorities Recall over Precision in a tradeoff.

The four scores I will calculate for each model are explained below.

Accuracy: is the fraction of predictions our model got right (e.g. what proportion of twits we predicted/classified as positive or negative were actually positive or negative).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision: tells us what proportion of positive identifications was correct (e.g. what proportion of actually positive twits we predicted as positive twits).

$$Precision = \frac{TP}{TP + FP}$$

Recall: tells us what proportion of positives was identified correctly (e.g. what proportion of all positive twits did we predict as positive)

$$Recall = \frac{TP}{TP + FN}$$

F1 Score: is a weighted average of precision and recall ranging from 0 to 1.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

TP: True Positive
TN: True Negative
FP: False Positive
FN: False Negative

Analysis

Data Exploration

For this project I used the data-set called Sentiment140. The data-set is split into both a training set and testing set. The training set contains 1600000 tweets. The test set contains 498 tweets.

Note: The Sendiment140 data-set has 1.6M row of data which was taking too long to load so I reduced the data-set to ~0.5M rows. Also, when creating the frequency matrix, I was getting out of memory errors, so I only use a subset of the loaded data-frame (~18K). I also decided to not use the

test set (498 test tweet) and instead split the ~18K between training and testing data.

The tweets are is a csv file with the following fields:

- labels
- id
- date
- query
- user
- text

I will only use the text and labels fields for this project, so I will drop all other columns.

The tweets are classified as

- 0 = negative
- 2 = neutral
- 4 = positive

Roughly half (8475) of the tweets are classified as negative and half (9477) of the tweets are classified as positive. There are no neutral tweets in this data-set. This means the data-set is balanced so neither class should dominate during training.

Sample of the data-set

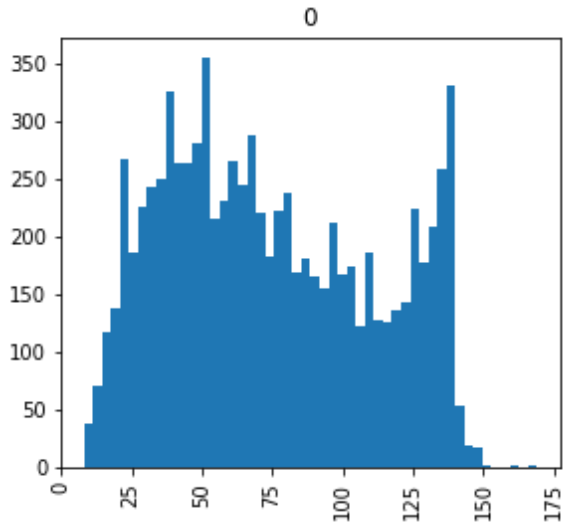
	Labels	Id	Date	Query	User	Text
0	0	2205441133	Wed Jun 17 04:44:48 PDT 2009	NO_QUERY	Julie_oh	@tiedyeina lucky you! 6 more days
1	0	2205441225	Wed Jun 17 04:44:49 PDT 2009	NO_QUERY	KatieBug1112	Morning everyone! :-D not in a good mood righ...
2	0	2205441321	Wed Jun 17 04:44:50 PDT 2009	NO_QUERY	jbh_dc	Back in rainy reston ... With a wife who has t...
3	0	2205441485	Wed Jun 17 04:44:51 PDT 2009	NO_QUERY	aaakritiLove	@jysla :S:S whats wrong, dear?
4	0	2205441608	Wed Jun 17 04:44:52 PDT 2009	NO_QUERY	JackyDouglas	Hates the rain!

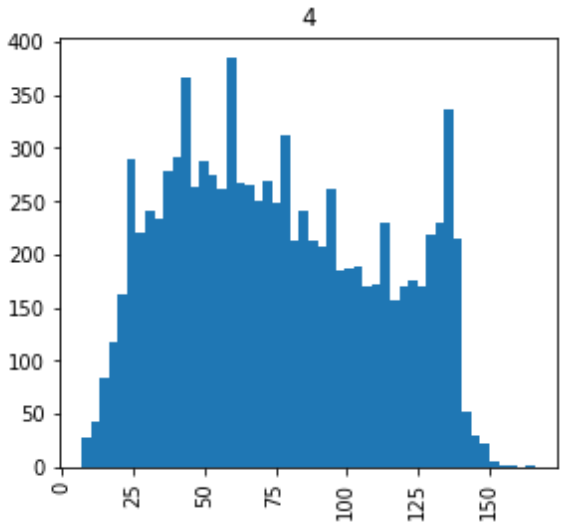
Labels: The sediment (0 for positive and 4 for negative).
Id: The id of the twitter.
Date: The date of the twit.
User: The name of the twitter.
Text: The text of the twit.

The data is fairly balance between the target classes.
4 (positive): 9477
0 (negative): 8475

Exploration Visualization

Below is a histogram plotting the length of both the negative and positive labeled text. The text length varies but is reasonable distributed for both label values (positive and negative).





Algorithms and Techniques

The technique I used for this project is called Bag of Words (described below). I used count vectorization with the baseline model and count and tfidf vectorization for my final model (described below).

Bag of words

The bag of words process is used when you have a collection of text (twit) data that needs to be processed. You take each word and count its frequency within a piece of text. Each word is treated independently, and the order is irrelevant.

I use the sklearn CountVectorizer class to convert the list of texts (twits) into a matrix with each text (twit) been a row and each word been a column. The corresponding row:column value is the frequency of the occurrence of each word in that twit. I've set the stop_words parameter to English to ignore like 'the', 'is', 'are', etc. as they carry less important than other words. It also helps lower the feature space.

Count Vectorizer

I use the sklearn.feature_extraction.text.CountVectorizer to

- Separate the string into individual words and give each word (column) an integer ID.
- Count the occurs of each word (column) in each twit (row).
- Covert all words to lower case.
- Ignore all punctuation.
- Ignore all stop words.

Benchmark

Reason

This is a supervised binary classification problem as the texts (twits) are either positive or negative. I will provide a labeled data-set (sendiment140) to train and test the model.

Model

I used the Naïve Bayes model as my baseline model as it is a quick and easy way to predict classes. It is based on a statistical classification technique known as Bayes Theorem. It is naïve because it assumes that the variables are independent from each other.

It calculates the posterior probability of a certain event A to occur given some probabilities of prior events.

$$P(A|R) = \frac{P(R|A)P(A)}{P(R)}$$

P(A): Prior probability of a class.
P(R|A): Likelihood the probability of predictor given class.
P(R): Prior probability pf predictor.
P(A|R): Posterior probability of class A given the predictor R.

To use the Naïve Bayes algorithm to solve classification problems like deciding if text is positive or negative.

- Split the data.
- Vectorize the data into a frequency table.
- Fit the model with the training data-set.
- Make predictions with the test data-set.
- Score the model.

See the models/BaselineModel class in my project for details.

Strengths

- Easy and quick to implement.
- If the conditional independence holds then it will converge quickly.
- Need less training data.
- Scalable.
- Not sensitive to irrelevant features.

Weaknesses

- The naïve assumption of independent features is unlikely in the real world.

Split Data-set

I split the data-set into four buckets
X_train: Training data for the text (twit) column.
y_train: Training data for the label column.
X_test: Testing data for the text (twit) column.
y_text: Testing data for the label column.

Number of rows in the total set: 17952
Number of rows in the training set: 13464
Number of rows in the test set: 4488

After splitting the data-set, we need to convert the data into the matrix format using CountVectorizer. For the training data we need to fit first, so the model can learn a vocabulary and then transform to the matrix view. For the testing data-set we only need to transform to the matrix view.

Code

I used the multinomial Naïve Bayes implementation for my baseline model because it is suitable for classifications with discrete features like word counts for text classification.

```
def fitNaiveBayes(self):
    self.naive_bayes = MultinomialNB()
    self.naive_bayes.fit(self.training_data, self.y_train)
```

Now that I've trained the model with the training data-set I can use the testing data-set to make predictions.

```
def predict(self):
    self.predictions = self.naive_bayes.predict(self.testing_data)
```

See the models/BaselineModel class in my project for more detail.

Methodology

Data Preprocessing

The data preprocessing was explained above in the Algorithms and Techniques section above. I use the bag of word technique to create a frequency matrix of all the words in my data-set using both the CountVectorizer and the TfidfVectorizer.

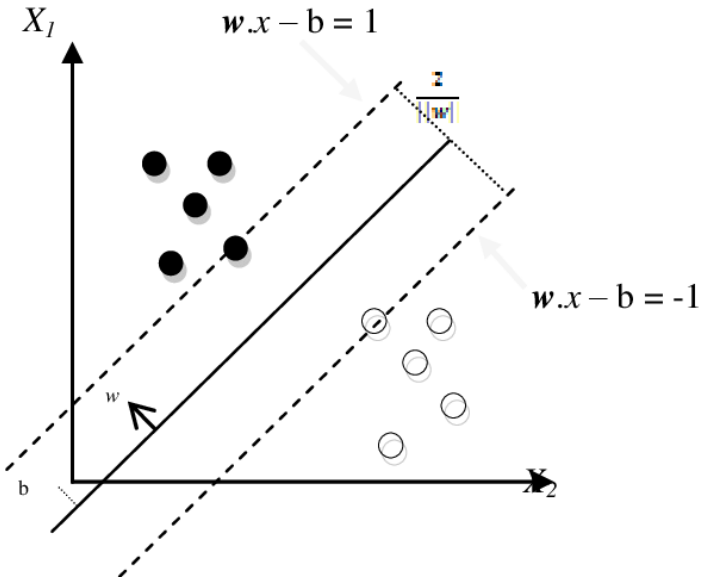
Implementation

Reason

I will use support vector machine (SVM) algorithm for my final model as again it is a good model for classification problems.

Model

The support vector machine algorithm is to find a hyperplane in an N-dimensional space, where N is the number of features that distinctly classify the data points. Our objective is to find a plane that maximizes the distance between the two classes of data (positive and negative).



SVM draws the hyperplane by transforming our data with the help of Kernels (mathematical functions). There are many types of Kernels (linear, sigmoid, rbf, polynomial) but as our problem is to classify data between positive and negative we will use the linear kernel for our model.

To use the SVM algorithm to solve classification problems like deciding if text is positive or negative.

- Gather the data (we will use Sentiment140 data-set).
- Vectorize the data.
    - For comparison reasons we will try both CountVectorizer and TfidVectorizer.
- Train and test the model.
- Measure the scores of the model.

Strengths

- With an appropriate kernel function, we can solve any complex problem.
- It scales relatively well to high dimensional data.

Weaknesses

- They are memory intensive.

- Choosing a "good" kernel function is not easy.
- Long training time for large data-sets.

Split Data-set

Same as the baseline model (see above).

Code

I used the sklean SVM with a linear kernel as my final model because it is suitable for classifications with discrete features like word counts for text classification.

```
def fitNaiveBayes(self):
    self.svn = svm.SVC(kernel='linear')
    self.svn.fit(self.training_data, self.y_train)
```

Now that I've trained the model with the training data-set I can use the testing data-set to make predictions.

```
def predict(self):
    self.predictions = self.svn.predict(self.testing_data)
```

Depending on which Vectorizer I use I get different results. If you look at the scores section below you can see when we use the Tfidf Vectorizer we get better scores across the board.

See the models/Model class in my project for more detail.

Refinement

There are different types of vectorizers you can use with the Bag of Words technique to calculate the frequency of the words. As a refinement I try the TfidfVectorizer with the SVM model.

Tfidf Vectorizer

Term frequency and Inverse Document Frequency (TF-IDF) are word frequency scores that try to highlight words that are more interesting in a piece of text or twit. It looks at the frequency of words in a twit but not across twits.

$$idf(t) = \log\frac{1 + n_d}{1 + df(d,t)} + 1$$

n_d = Number of documents.
$df(d,t)$ = is the term present in number of documents.

Training/Testing Splits

The default training to testing split is 25/75 which is what I used for the values in this report, but I also tried an 20/80 split which gave me better scores for the Naïve Bayes/CountVectorizer and SVM/ CountVectorizer but worse scores for my final model SVM/TfidfVectorizer.

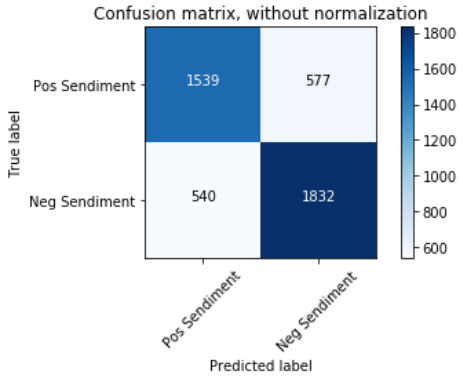
Results

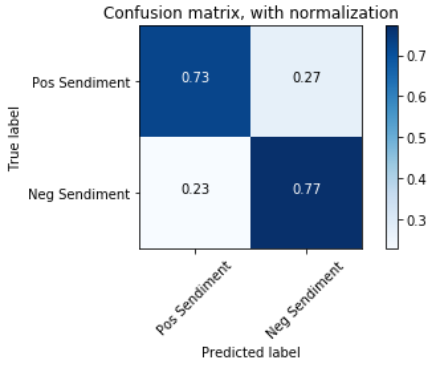
Model Evaluation and Validation

Below is the output of the scores for each of the model/vectorizer combinations with a confusion matrix.

Baseline (Naïve Bayes) Scores with CountVectorizer

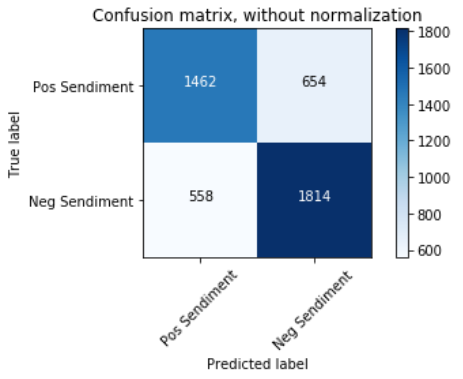
- Accuracy score: 0.7511140819964349
- Precision score: 0.7604815276048152
- Recall score: 0.7723440134907251
- F1 score: 0.7663668688558879

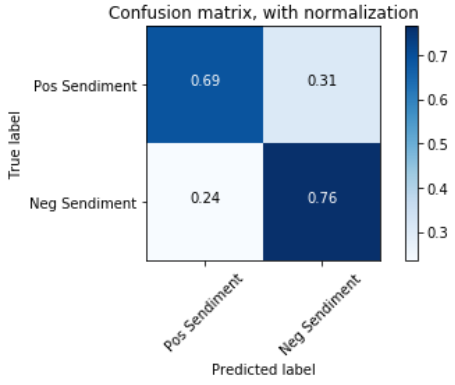




Final (SVM) Scores with CountVectorizer

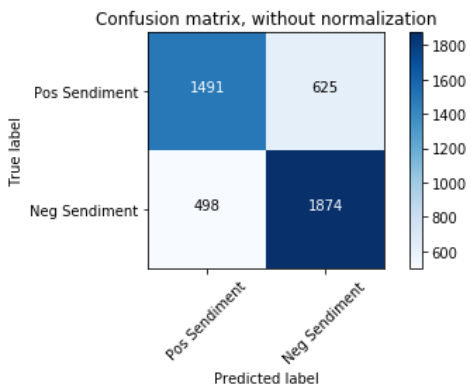
- Accuracy score: 0.7299465240641712
- Precision score: 0.7350081037277147
- Recall score: 0.7647554806070826
- F1 score: 0.749586776859504

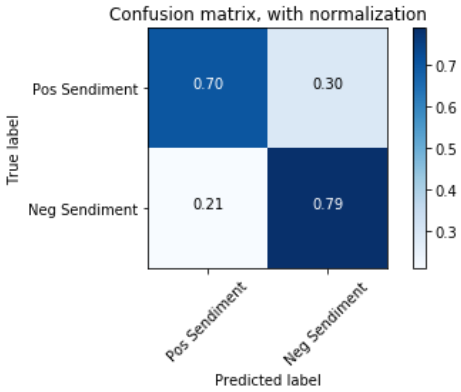




Final (SVM) With TfidfVectorizer

- Accuracy score: 0.7586898395721925
- Precision score: 0.7612484799351439
- Recall score: 0.7917369308600337
- F1 score: 0.7761934283942964





Using the testTwit method I can pass in some real text (twit) and get back the sentiment.

Justification

The table below shows the results of the three models/vectorizer I trained and tested and as you can see the SVM model with the TfidfVectorizer give the best results.

	Naïve Bayes	SVM/CountVectorizer	SVM/TfidfVectorizer
Accuracy	0.7511	0.7299	0.7709
Precision	0.7605	0.7350	0.7786
Recall	0.7723	0.7648	0.7917
F1	0.7664	0.7496	0.7851

Conclusion

Free-Form Visualization

	00	000	0000r0cx	0007	000th	00am	01	01000101	01614948343	018	...	½rmï	½s	½se	½stand	½t	½tï	½ve	½y	½ï	ã¼ã
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

With the large amount of training data and the various characters people use in texts (twits) we have a lot of strange feature names as you can

see from the column name above. It also means we have a lot of column 27387 (even after we use the stop_words parameter).

Reflection

Overall, I think the project went well, my choice of baseline model (Naïve Bayes) and final model (SVM) both works well for this binary classification problem.

The most difficult part of the project was the python code as I don't use it daily. I'm not as familiar with it as I'd like to be and hence some things took longer to code than they should.

The most interesting part was seeing how using different vectorizers (or ways to calculate word frequency) can improve the model scores. I only used two (count and tfidf) but there are others I also need to test with.

The plan was to make a request to the twitter APIs so I can search for twits about a business and use the testTwit() method to see if a twit is positive or negative. This would allow me build up a history of the sentiment of a business over time. I have a request into twitter to use their search APIs, but it's still not approved.

Improvement

The scores for the SVM model with CountVectorizer are in general worse than the baseline model but when I used the TfidfVectorizer I see improvements. I also need to test will other vectorizer like Hash and Dict.

The SVM model with either of the vectorizers take significantly longer (CV: ~20 secs, TFIDF: ~40 secs) to fit the data than the baseline NB model (<1 sec). But time to train the model isn't an issue for this problem so I'm ok with that.

Removing the stop words helped reduce the number of features but I still have over 20k which is to many. I need to add a word cleaning method to remove words that start with certain characters like #, @ and words that contain certain string like http, https.

I was running out of memory when I tried to use the full data-set, but I think if I run the code outside the browser I could use more (if not all) the data-set and hence have a big data-set to train and test the model.

References

1. Sentiment140: http://help.sentiment140.com/for-students/
2. Twitter Search API: https://developer.twitter.com/en/docs/tweets/search/api-reference/get-search-tweets