Batuhan Erden

batuhan.erden@ozu.edu.tr

CS 452/552 – Data Science with Python

Project 2 Report

14.12.2017

Sentiment Analysis Tool for Tweets

1 Introduction

This is a tool that reads tweets and sentiments from a .csv (Excel) file and trains a model using these tweets. After I read these tweets, I stored them in my own data structure. Since the data had too many hash tags and links in it, I have also cleaned these from the tweet to have a better performance. I had 2 numpy arrays; **X** and **y**. **X** was the tweets and they were vectorized since tweets are bunch of strings and **y** was the sentiments. There are 4 sentiments categories: *Positive*, *Negative*, *Neutral* and *Irrelevant*. The **X** was basically the input while the **y** was basically the output. To successfully test the classifiers, I made use of the *10-Fold Cross Validation* which means that train data is *randomly chosen 90% of the data* and test data is r*andomly chosen 10% of the data*. It also means that testing the model part is repeated 10 times. I have implemented 5 different classifiers from **scikit-learn** library and sorted them with respect to their performances. These classifiers are: *LinearSVC*, *MLPClassifier*, *MultinomialNB*, *BernoulliNB* and *SVC.*

2 Excel Data (Tweets)

As can be seen in the *Figure 1*, the rows in the excel data is like this. I only use Sentiment and TweetText columns. In the data, there are 5113 rows. As described in the introduction, there are 4 sentiments categories: *Positive*, *Negative*, *Neutral* and *Irrelevant*.

Topic	Sentiment	TweetId	TweetDate	TweetText
apple	positive	1.26416E+17	Tue Oct 18 21:53:25 +0000 2011	Now all @Apple has to do is get swype on the iphone and it will be crack. Iphone that is
apple	positive	1.26405E+17	Tue Oct 18 21:09:33 +0000 2011	@Apple will be adding more carrier support to the iPhone 4S (just announced)
apple	positive	1.26403E+17	Tue Oct 18 21:02:20 +0000 2011	Hilarious @youtube video - guy does a duet with @apple 's Siri. Pretty much sums up the love affair! http://t.co/8ExbnQjY
apple	positive	1.26397E+17	Tue Oct 18 20:40:10 +0000 2011	@RIM you made it too easy for me to switch to @Apple iPhone. See ya!
apple	positive	1.26396E+17	Tue Oct 18 20:34:00 +0000 2011	I just realized that the reason I got into twitter was ios5 thanks @apple
apple	positive	1.26395E+17	Tue Oct 18 20:30:50 +0000 2011	I'm a current @Blackberry user, little bit disappointed with it! Should I move to @Android or @Apple @iphone
apple	positive	1.2638E+17	Tue Oct 18 19:30:39 +0000 2011	The 16 strangest things Siri has said so far. I am SOOO glad that @Apple gave Siri a sense of humor! http://t.co/TWAeUDBp via @HappyPlace
apple	positive	1.26378E+17	Tue Oct 18 19:22:35 +0000 2011	Great up close & personal event @Apple tonight in Regent St store!
apple	positive	1.26374E+17	Tue Oct 18 19:07:11 +0000 2011	From which companies do you experience the best customer service aside from @zappos and @apple?

Figure 1: Excel Data (Tweets)

3 Pre-processing the data

My first way of approaching to this data was checking the character variation in the tweets. I realized that there were too many hash tags and links, which I thought that they might decrease the performance. After doing some experiments, my thesis on this seemed correct and cleaning the data actually improved my classifiers' performances. I performed the cleansing operation by just simply removing characters like **'@'** and **'#'** if they appeared as the first element of the words. Besides, I omitted the words if their length were smaller than or equal to 1 and if they contained **"http://"**. Removing links from the tweets resulted me the best gain in terms of performance.

4 Preparing the dataset

As mentioned above, I had 2 numpy arrays; **X** which were the tweets and **y** which were the sentiments. Since my input array must have consisted of floats not strings, I have used **TfidfVectorizer** to vectorize the tweets into a sparse 2D matrix suitable for feeding into a classifier. For example, the string "*Now all Apple has to do is get swipe on the iphone and it will be crack. Iphone that is*" was converted into an array that can be seen in the *Figure 2*. In addition, the shape of the **X** becomes (5113, 13468) while its first shape is (5113,).

```
(0, 7946)     0.206302625562
(0, 686)      0.214064802311
(0, 945)      0.11054860177
(0, 5141)     0.226545934069
(0, 11442)    0.126393529016
(0, 3337)     0.209378725316
(0, 5929)     0.301098802635
(0, 4760)     0.204551456599
(0, 10964)    0.4053213556
(0, 8136)     0.146022686837
(0, 11282)    0.12109543942
(0, 5888)     0.371188338262
(0, 799)      0.14674675826
(0, 5960)     0.161215773988
(0, 12461)    0.232465986237
(0, 1347)     0.203700348341
(0, 2666)     0.373563655103
(0, 11275)    0.187330587863
```

Figure 2: Vectorized version of a string

5 KFold Cross Validation (k = 10)

In this project assignment, I used 10-Fold Cross Validation which basically means that k is **10** in KFold Cross Validation. As can be seen in the *Figure 3*, KFold is initialized with **10** splits and shuffle is set to **True** which basically means that the data is shuffled each time before testing the classifier. Classification operation occurs inside the loop. For each iteration, train data and test data are both chosen randomly. 90% of the randomly chosen data becomes the train data and the rest becomes the test data. The classification operation inside the loop is repeated **10** times and scores are stored at each iteration. In conclusion, the average score accumulated is printed to console.

```
def test_classifier_using_kfold(name, classifier, X, y, k=10):
    scores = []
    kfold = KFold(n_splits=k, shuffle=True)

    for train_index, test_index in kfold.split(X):
        classifier.fit(X[train_index], np.take(y, train_index))
        score = classifier.score(X[test_index], np.take(y, test_index))

        scores.append(score)
        print("..", end="")

    print("\n{} Average Score: {:.2f}%".format(name, np.mean(scores) * 100))
    # draw_scores(name, scores)
```

Figure 3: Implementation of KFold Cross Validation (k=10)

6 Classifiers

As can be seen in *Figure 3* above, performing classification with **scikit-learn** is pretty simple. First, you define a classifier. Then, you use **fit** function with the train data (*X[train_index] and y[train_index]*) to train the classifier. Finally, you check its performance by using **score** function with the test data (*X[test_index] and y[test_index]*). In this assignment, I have used 5 different classifiers of **sklearn** library which are: *LinearSVC*, *MLPClassifier*, *MultinomialNB*, *BernoulliNB* and *SVC*. In the next steps, I will demonstrate the results of each classifier starting from the best and ending with the worst.

6.1 LinearSVC (81.32%)

It is a Linear Support Vector Classification. It is also same as SVC (Support Vector Classification) but kernel is linear in this classifier. The results and implementation of this classifier can be seen in the *Figure 4* below. There are 2 arguments given. The 1st one is the penalty parameter **C** of the error term. I set this one to **1.48** after doing significant number of experiments. The 2nd one is the loss function. The default loss function was **squared_hinge**, the square of the **hinge** loss but I decided to use hinge, which is the standard SVM loss, as it gives the best performance.

```
test_classifier_using_kfold("LinearSVC", svm.LinearSVC(C=1.48, loss="hinge"), X, y)
```

```
.....................
LinearSVC Average Score: 81.32%
```

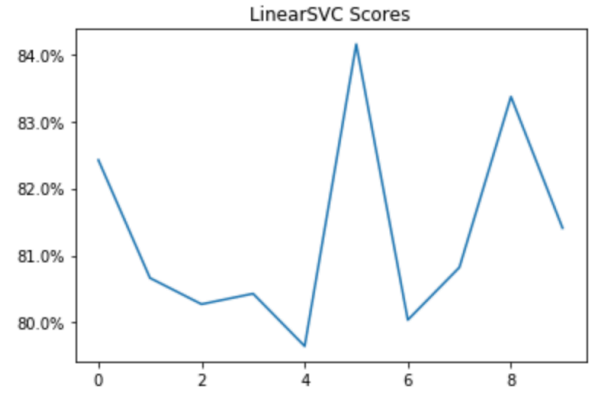

Figure 4: Results and Implementation of LinearSVC Classifier

6.2 MLPClassifier (77.66%)

MLPClassifier is a multi-layer perceptron algorithm that trains using back-propagation. It is a supervised learning algorithm. The results and implementation of this class classifier can be shown in the *Figure 5* below. I used 4 arguments in the implementation of MLPClassifier. The 1st one is the hidden layer sizes and I decided to make it **(30, 30)**. It might be too much for our use case but I experienced great results with this parameter. The 2nd one is the solver for weight optimization. I used **Adam** which is a stochastic gradient-based optimizer and which is a great optimizer in machine learning. The 3rd one is the activation function for the hidden layers. I decided it to be **ReLu (Rectified Linear Unit Function)** that returns **f(x) = max(0, x)**. For the last one, I made the number of max iterations 2000 to never stop in the middle of training. This is just for being non-risky.

```
test_classifier_using_kfold("MLPClassifier", MLPClassifier(
    hidden_layer_sizes=(30, 30), solver="adam", activation="relu", max_iter=2000), X, y)
```

```
.....................
MLPClassifier Average Score: 77.66%
```

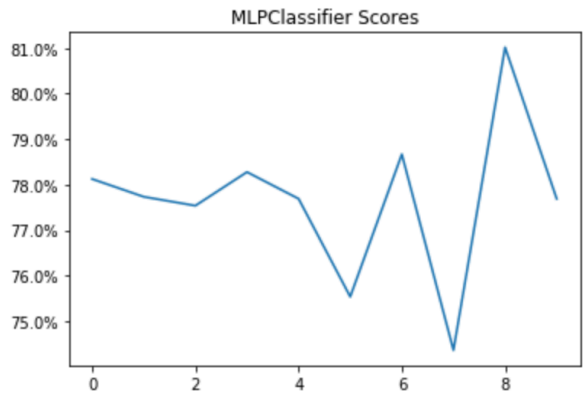

Figure 5: Results and Implementation of MLPClassifier

6.3 MultinomialNB (72.89%)

It is a multinomial Naïve Bayes classifier that is suitable for classification with discrete features. The results and implementation of this classifier can be seen in the *Figure 6* below. I changed 1 argument for the implementation of this classifier. It's Alpha, Additive smoothing parameter. After a significant number of experiments, I set it to 0.61 as it gives the best performance.

```
test_classifier_using_kfold("MultinomialNB", MultinomialNB(alpha=.61), X, y)
```

```
.....................
MultinomialNB Average Score: 72.89%
```

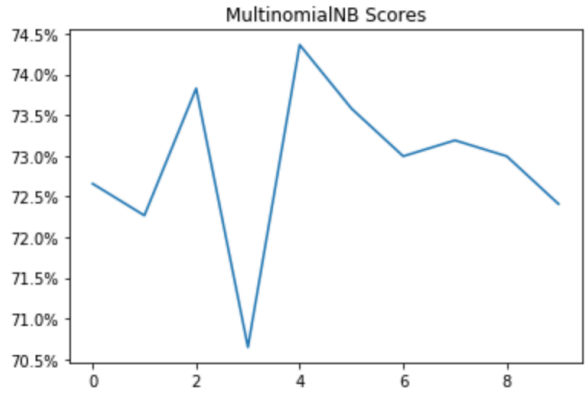

Figure 6: Results and Implementation of MultinomialNB

6.4 BernoulliNB (72.25%)

It is also a Naïve Bayes classifier. Like MultinomialNB, it is also suitable for classification with discrete features. The only difference is bernoulli is designed for binary/Boolean features while multinomial works with occurrence counts. The results and implementation of this classifier can be seen in the *Figure 7* below. I have also used Alpha and set it to 1.08 after doing a significant number of experiments.

```
test_classifier_using_kfold("BernoulliNB", BernoulliNB(alpha=1.08), X, y)
```

```
.....................
BernoulliNB Average Score: 72.25%
```

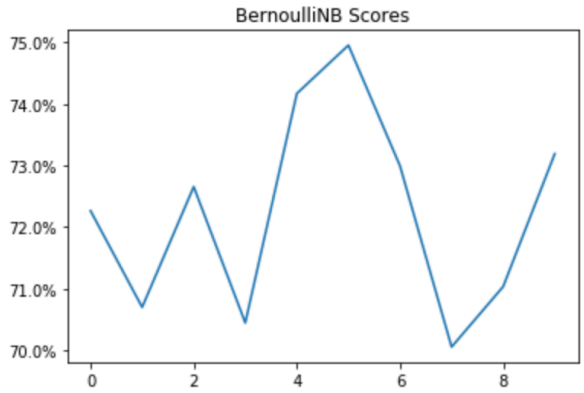

Figure 7: Results and Implementation of BernoulliNB

6.5 SVC (45.63%)

SVC is Support Vector Machines as described in the *Section 2.1* but without a linear kernel. The results and implementation of this classifier can be shown in the *Figure 8* below.

```
test_classifier_using_kfold("SVC", svm.SVC(), X, y)
```

```
.....................
SVC Average Score: 45.63%
```

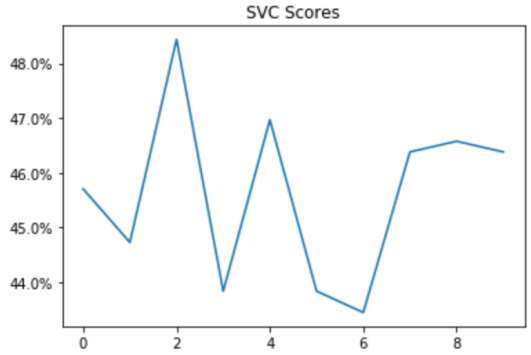

Figure 8: Results and Implementation of SVC

7 Conclusion

In conclusion, after testing these 5 different classifiers, I have examined some changes on performances after applying different parameters. I have also realized that each classifier is strong on some cases and weak on another. In this case, LinearSVC performed the best among 5 classifiers. At first, I expected MLPClassifier to perform better than LinearSVC since it's neural network. However, LinearSVC tended to be slightly better than MLPClassifier. I think that it's because of the size and complexity of the data. If the data were bigger and more complex, maybe MLPClassifier would perform better. To conclude, I think all classifiers except SVC performed well. The overall results can be seen in the *Figure 9* below.

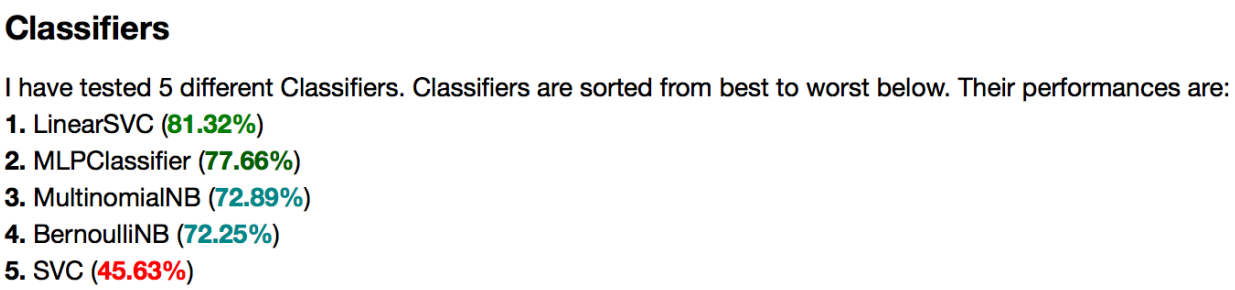

Figure 9: Overall Results

References

- http://scikit-learn.org