Sentiment Analysis with SVM

Lizzy Burl

CSE 5522

4/24/2014

1. Abstract

The purpose of the experiment was to implement sentiment analysis using support vector machines. The data that was analyzed were movie reviews from Rotten Tomatoes. The Support Vector Machine was trained with 700 positive and 700 negative reviews. It was then tested with 692 negative and 694 positive reviews. The model performed well under test, getting accuracies above 99% for different kernel function (i.e. linear, polynomial, etc.).

1. Introduction

The goal of the research project was to see how effective support vector machines are at implementing sentiment analysis of Rotten Tomato movie reviews. Essentially, it was necessary to see how accurately a SVM could predict if a given movie review was positive or negative. A support vector machine (SVM) is a supervised learning model that is used for classification. In order to be used, it must first be trained with training data. The goal is to find separator with the widest margin between training data sets. The reviews were separated into two classes: positive and negative reviews. The SVM was trained with 700 positive and negative reviews; then tested with 700 positive and 700 negative reviews. This report will cover the methodology use, the results from this methodology as well as a discussion of the results, an analysis of possible future improvements, and a conclusion.

1. Method

**Data Acquisition**

Movie reviews were available as text files in folders that separated them into positive and negative reviews [2]. The reviews were somewhat cleaned, meaning that a space had been added between words and common punctuation elements and all words were lowercase. The review set that was used for training is labeled “polarity dataset v1.0” and the review set that “polarity dataset v1.1.”

**The Support Vector Machine**

Actually implementing a Support Vector machine is incredibly difficult. Therefore, a pre-existing implementation, LibSVM was used [1]. LibSVM is an implementation of a SVM that is offered for multiple programming languages.

**Training the SVM:**

1. **N-Length Reference Dictionary:**

All entries of training data were read, which consisted of 700 positive and 700 negative reviews. Each word was added as an entry to the dictionary, where the word was the key and the number of times it appeared in every entry was the value. Keys such as “the,” “a,” “.,” and other common punctuation were removed. Furthermore, any entry contained non-ascii characters were removed. Lastly, words that only appeared once out of all 700 negative and 700 positive reviews were removed. This significantly reduced the length of the dictionary, or *n*, speeding up later parts of the code. Furthermore, since the word only appeared once in the 1400 training data reviews, it had such a low significance it was negligible.

The significance scores were not normalized. At first, normalization was attempted, but since to the large total number of words, most values simply became zero, which was not useful.

The following is an example of the format of the final dictionary:

{"woods": 55, "spiders": 9, "hanging": 30, "comically": 7}

The dictionary was stored as a json object to be used in a separate file. The code used to create it can be seen in Attachment 1, as “InitialDict2Faster.py.”

1. **Finding the N-Dimension Vector for Each Review:**

It was then necessary to find the vectors, and that would be given to LibSVM. The vector was of size *n* for the number of unique word entries in the dictionary. Each word in the training data was given a count for how many times it appeared. This count was multiplied by the score given to that word by the dictionary.

Thus if a positive review contained the following values:

{"woods": 0, "spiders": 1, "hanging": 0, "comically": 1}

After the words were multiplied by their significance score, it would look like this:

{"woods": 0, "spiders": 9, "hanging": 0, "comically": 7}

Words that appeared in the *n* length dictionary vector that did not appear in the review simply had zero values.

1. **Preparing the Review for LibSVM**

Each review needed to be in the following formal for LibSVM

<label> <index1>:<value1> <index2>:<value2> ...

Where:

**Label:** +1 or -1 for positive of negative reviews, respectively

**Index#:** Must be an integer value. A map called indexMap was used to assign every value in the dictionary to a numerical index.

**Value#:** Number of times the word appeared in the review Significance score of the word.

The index map would simply map each dictionary value to a unique index. It might look like this:

{"woods": 1, "spiders": 2, "hanging": 3, "comically": 4}

Fortunately, LibSVM allowed dictionary entries with a zero label to be ignored. Thus, adhering to the example of a positive review used above, the output would look like this.

+1 2:9 4:7

The code that executes this can be found in the file “preSVM.py,” in the attachments. A function prepareOutput(outputFileName, posPath, negPath) was used in order to separately put both the training and testing data in the format needed for LibSVM.

1. **Training the SVM**

Now that the data was ready to go, LibSVM could be used. At this point, it was quite simple. A SVM model was created based on training data loaded from the training output. That model was then used to test the testing data. The file SentimentAnalysis.py, seen in the attachments contained the following code:

from svmutil import \*

y, x = svm\_read\_problem('trainingOutput');

# When this was run, the third parameter consisted of a string # with a flag specifying what type of kernel function to use

# and what kind of SVM to use

m = svm\_train(y, x, ‘-t 0 –s 0’);

yTest, xTest = svm\_read\_problem('testingOutput');

p\_label, p\_acc, p\_val = svm\_predict(yTest, xTest, m);

print p\_acc;

The value p\_acc printed a tuple containing: accuration, mean squared error, square correlation coefficient

1. Results and Discussion

With 700 positive and 700 negative reviews to train the model, the results were incredibly accurate. The testing data contain 694 positive movie reviews and 692 negative reviews. Two different types of SVMs were observed:

1. C-SVC: Regularized support vector classification
2. Nu-SVC: Automatically regularized support vector classification

Table 1 shows the results for different kernel functions using C-SVC and Table 2 shows the results using -SVC. Both tables are results from 700 positive and 700 negative reviews

**Table 1:** SVM Results for Different Kernel Functions using C-SVC for Multi-Class Classification (700 pos, 700 neg)

|  |  |  |  |
| --- | --- | --- | --- |
| Kernel Function | Linear | Polynomial  (gamma\*u'\*v + coef0)^degree | Radial Basis Function:  exp(-gamma\*|u-v|^2) |
| Accuracy | 99.1342% | 99.2063% | 99.2063% |
| Mean Square Error | 0.034632 | 0.031746 | 0.031746 |
| Iterations | 10,000,000 (Max) | 3,738,521 | 794 |

**Table 2:** SVM Results for Different Kernel Functions using -SVC for Multi-Class Classification (700 pos, 700 neg)

|  |  |  |  |
| --- | --- | --- | --- |
| Kernel Function | Linear | Polynomial  (gamma\*u'\*v + coef0)^degree | Radial Basis Function:  exp(-gamma\*|u-v|^2) |
| Accuracy | 86.7965% | 84.4877% | 99.2063% |
| Mean Square Error | 0.528138 | 0.620491 | 0.031746 |
| Iterations | 10,000,000 (Max) | 10,000,000 (Max) | 2088 |

Thus all SVM models showed a high degree of accuracy. The radial basis function was by far the most efficient model to reach a high degree of accuracy. The best kernel function to use was the radial basis function, because it was by far the most efficient algorithm. It reached the desired accuracy level with 794 iterations in the C-SVC model, and 2088 iterations in the -SVM model, while the linear model used the maximum allowed iterations in both models, and the polynomial used 3,738,521 and the maximum, respectively.

The accuracy was due to the large sample size of positive and negative reviews in the training data. The SVM did not perform nearly as well with 1/10 of the amount of training data. The same tests were performed on training data sets with 70 positive and 70 negative reviews. The testing data still contained 694 positive movie reviews and 692 negative reviews.

Table 3 shows the results for different kernel functions using C-SVC and Table 4 shows the results using -SVC. Both tables are results from 70 positive and 70 negative reviews

**Table 3:** SVM Results for Different Kernel Functions using C-SVC for Multi-Class Classification (70 pos, 70 neg)

|  |  |  |  |
| --- | --- | --- | --- |
| Kernel Function | Linear | Polynomial  (gamma\*u'\*v + coef0)^degree | Radial Basis Function:  exp(-gamma\*|u-v|^2) |
| Accuracy | 62.987013% | 55.2251% | 54.7619% |
| Mean Square Error | 1.480519 | 1.670996 | 1.809524 |
| Iterations | 208,322 | 31,986 | 70 |

**Table 4:** SVM Results for Different Kernel Functions using -SVC for Multi-Class Classification (70 pos, 70 neg)

|  |  |  |  |
| --- | --- | --- | --- |
| Kernel Function | Linear | Polynomial  (gamma\*u'\*v + coef0)^degree | Radial Basis Function:  exp(-gamma\*|u-v|^2) |
| Accuracy | 60.9668% | 56.349206% | 54.7619% |
| Mean Square Error | 1.561327 | 1.746031 | 1.809524 |
| Iterations | 468,440 | 10,000,000 (Max) | 70 |
|  |  |  |  |

Thus it is conclusive that the SVM does a good job at classifying data as long as it has a large enough training data set. A training data set of 700 positive and 700 negative reviews was a really good size to achieve a high level of accuracy.

1. Future Improvements

**Normalize Dictionary:**

An original plan for the initial dictionary that provided significance values for each word was to normalize it such that all significance values would have added to equal 1. However, this created an issue, since the data set was incredibly large. Most of the values simply evaluated to zero because they were so small.

In the future, I would make further attempts to normalize the dictionary. The removal of values that only appeared once may make it possible to normalize the dictionary. Another thing that could be tried would be to divide the value by the total values separate for each review, only after it is multiplied by the number of times a word appears in said review (therefore, if it appears more than once it would have a higher value and be less likely to equate to zero—This approach would favor words that appear many times overall or words that appear many times in a single review).

**Efficiency**

There is much that could have been done to increase the efficiency of the code.

* **Creating the significance dictionary (InitialDict2Faster.py)**

The runtime was already drastically improved by making a separate dictionary every 30 entries and merging it with the all-encompassing dictionary. This was done for positive reviews and negative reviews and these dictionaries were consequently merged. However, I believe this could be further improved by merging a 30 review dictionary with another 30 review dictionary, and then merging the resulting dictionary with a 60 review dictionary, and so on.

* **Finding and formatting the vectors to use the SVM model (preSVM.py)**

The running time of the function prepareOutput could absolutely be improved upon. As it is, it iterates through each review checks if the word in the *n* length vector. If it is not, the word is ignored. If it is, a count is kept for the amount of times that said word appears. Checking to see if each word appears takes a considerable amount of time. There are a few things that would be interesting to try:

* Keeping a dictionary of dictionaries where the key of the dictionary is the first character and the value is a dictionary containing entries that start with that same first character
* Instead of having a dictionary for significance scores (word, score) and a dictionary for indices (word, index), I would have one dictionary with a tuple containing the score and the index, represented as (word, <score, index>)
* Read up more about python and python optimizations, especially for dictionaries. I used this project to introduce myself to python, so I’m sure there are many things that a seasoned python programmer would do to improve the efficiency.

**Different Calculation of Significance Score**

A pre-existing dictionary would be used, such as SentiWord [3]. These dictionaries give positive and negative associations to words. If this were to be used, significance scores would be replaced with a “strength score.” Essentially, the score would be higher if there was either a strong positive association or a strong negative association.

**Different Approaches to Sentiment Analysis**

The initial plan was to compare the accuracy of SVM versus Bayes for correctly identifying positive and negative reviews. One thing I would add is a Bayes model to compare the two approaches side by side.

**Attempt and Creating an SVM**

If time allowed for it, it would have been a cool experiment to implement a version of a support vector machine myself and compare it with the pre-existing libraries.

**Comparing Training Data Set Sizes**

A comparison was done between using 70 positive and 70 negative reviews vs. 700 positive and 700 negative reviews. It would have been nice to graph the increase in accuracy as the size of the training set increases.

1. Conclusion

The goal was to see how effective support vector machines are at determining the sentiment of movie reviews. With a training sample set of 700 positive and 700 negative reviews, the data was prepared for LibSVM. The testing data performed well under the SVM model that was trained. Multiple kernel functions were attempted, all scoring with above 99% accuracy under a C-SVC support vector machine (the standard model). It would have been really interesting to compare other types of sentiment analysis classification to the accuracy of the SVM, but time did not allow for that.

Attachment: InitialDict2Faster.py, 3

Attachment: preSVM.py, 2

Attachment: SentimentAnalysis.py, 1

Attachment: References, 1

**InitialDict2Faster.py**

import os;

import json;

# This was added to improve the efficiency of the program. Combines two dictionaries into one

def combineDictionaries (bigDict, miniDict):

for word in miniDict.keys():

if word in posDict.keys():

bigDict[word] = posDict[word] + miniDict[word];

else:

bigDict[word] = miniDict[word];

return bigDict;

# Gets rid of entries that should not be in the dictionary to

# (1) Decrease the size

# (2) Get rid of weird unicode entries that don't like being converted to json and back

def clearBadEntries(dictionary):

badEntries = ["the", "and", "a", "that", ".", ",", "they", ";", "[", "]", "(", ")", "\n", "\t"];

for entry in badEntries:

if entry in dictionary.keys():

del dictionary[entry];

for key in dictionary.keys():

if not all(ord(c)<128 for c in key):

del dictionary[key];

print key;

continue;

if dictionary[key] == 1:

del dictionary[key];

continue;

return dictionary;

# Initialize the dictionary for positive reviews

posDict = {};

totalWords = 0;

# Get all words from positive reviews

posPath = "C:/Users/Lizzy/Documents/AI2/trainingTokens/tokens/pos/";

# Mini dict gets filled every 30 reviews and is then joined with the big dictionary

miniDict = {};

dictMax = 30;

dictCount = 0;

for filename in os.listdir(posPath):

f = open(os.path.join(posPath, filename), 'r');

# Combining reviews

if dictCount == dictMax:

dictCount = 0;

posDict = combineDictionaries(posDict, miniDict);

miniDict = {};

for word in f.read().split():

totalWords = totalWords + 1;

if word in miniDict.keys():

miniDict[word] = miniDict[word] + 1;

else:

miniDict[word] = 1;

f.close();

print filename;

dictCount = dictCount + 1;

posDict = combineDictionaries(posDict, miniDict);

negPath = "C:/Users/Lizzy/Documents/AI2/trainingTokens/tokens/neg/";

# Get all words

negDict = {};

miniDict = {};

dictCount = 0;

for filename in os.listdir(negPath):

f = open(os.path.join(negPath, filename), 'r');

if dictCount == dictMax:

dictCount = 0;

negDict = combineDictionaries(negDict, miniDict);

miniDict = {};

for word in f.read().split():

totalWords = totalWords + 1;

if word in miniDict.keys():

miniDict[word] = miniDict[word] + 1;

else:

miniDict[word] = 1;

f.close();

print filename;

dictCount = dictCount + 1;

negDict = combineDictionaries(negDict, miniDict);

# Combine the dictionaries made from positive and negative values

dictionary = combineDictionaries(posDict, negDict);

dictionary = clearBadEntries(dictionary);

print(len(dictionary))

#Dumb dictionary into a json object

with open("dictionary.json", "wb") as fp:

json.dump(dictionary ,fp, encoding='latin1') ;

**preSVM.py**

import json;

import os;

# creates an output file ready for use by libSVM

# outputFileName: the output file name

# posPath: The path for the folder containing positive movie reviews

# negPath: The path for the folder containing negative movie reviews

def prepareOutput(outputFileName, posPath, negPath):

with open("dictionary.json", "rb") as fp:

dictionaryScore = json.load( fp, encoding="latin1")

indexMap = {};

counter = 1;

# The index cannot be a word; it must be a number, so we map each word to a unique number

for word in dictionaryScore.keys():

indexMap[word] = counter;

counter = counter + 1;

output = open(outputFileName , 'wb');

for filename in os.listdir(posPath):

f = open(os.path.join(posPath, filename), 'r');

reviewMap = {};

for word in f.read().split():

# Only care about words that are in the giant dictionary of words.

if not word in dictionaryScore.keys():

continue;

if word in reviewMap.keys():

reviewMap[word] = reviewMap[word] + 1;

else:

reviewMap[word] = 1;

# Format the output as it asked

output.write('+1 ');

for word in reviewMap.keys():

value = reviewMap[word] \* dictionaryScore[word];

output.write(str(indexMap[word]) + ":" + str(reviewMap[word]\*dictionaryScore[word]) + " ");

output.write("\n");

print filename;

f.close();

for filename in os.listdir(negPath):

f = open(os.path.join(negPath, filename), 'r');

reviewMap = {};

for word in f.read().split():

if not word in dictionaryScore.keys():

continue;

if word in reviewMap.keys():

reviewMap[word] = reviewMap[word] + 1;

else:

reviewMap[word] = 1;

output.write('-1 ');

for word in reviewMap.keys():

value = reviewMap[word] \* dictionaryScore[word];

output.write(str(indexMap[word]) + ":" + str(reviewMap[word]\*dictionaryScore[word]) + " ");

output.write("\n");

f.close();

print filename;

output.close();

# Prepares the training output to go to a file called "trainingOutput" and the test output to go to a file called "testingOutput"

prepareOutput("trainingOutput", "C:/Users/Lizzy/Documents/AI2/trainingTokens/tokens/pos/", "C:/Users/Lizzy/Documents/AI2/trainingTokens/tokens/neg/");

print "Moving on to testing output";

prepareOutput("testingOutput", "C:/Users/Lizzy/Documents/AI2/testTokens/pos/", "C:/Users/Lizzy/Documents/AI2/testTokens/neg/");

**preSVM.py**

from svmutil import \*

y, x = svm\_read\_problem('trainingOutput');

m = svm\_train(y, x, '-t 0 -s 0');

yTest, xTest = svm\_read\_problem('testingOutput');

p\_label, p\_acc, p\_val = svm\_predict(yTest, xTest, m);

print p\_acc;

**References**

[1] Lins, C. (n.d.). LIBSVM -- A Library for Support Vector Machines. Retrieved from http://www.csie.ntu.edu.tw/~cjlin/libsvm/

[2] Pang, B., & Lee, L. (n.d.). Movie Review Data. Retrieved from <http://www.cs.cornell.edu/People/pabo/movie-review-data/>

[3] SentiWordNet. (n.d.). Retrieved from <http://sentiwordnet.isti.cnr.it/>