Grant Nelson A10821659
Tamara Zubatiy A11904774
9/4/17
Anjum Gupta
COGS188 SS2 2017

Assignment 2: Sentiment Classification

1. Introduction

There are many comparisons that can be made between the different methods of sentiment classification that we used to predict whether a movie review was positive or negative. Five supervised machine learning models were used: K-Nearest-Neighbor, Perceptron, SVM, Linear Logistic Regression, and Fisher Linear Discriminant Analysis. Each classification algorithm has different pros and cons as they relate to sentiment classification.The **K-NN** model can be summarized as classifying a query point based on a majority vote of its nearest *k* neighbors. K-NN classification is simple with no distribution assumptions, has a great, diverse, task performance, is good for large training sets, and is non-parametric. **Perceptron** is a form of supervised learning where classifications are based on a linear predictor function that combines a set of weights with an object's feature vector. The perceptron algorithm is a compact, parametric, linear separator. The perceptron will not converge for non-linearly separable data, and it will also not search for the best linear decision boundary, merely the first that classifies correctly. The **Support Vector Machine**, or SVM, is a model that maps objects in space to classify objects in categories that have wide gaps in classification boundary. It is a non-probabilistic binary classifier, that can perform many non-linear classifications using higher dimensional hyperplanes to group the given categorical data(SVM). We used **Linear Logistic Regression** as a binary classifier to calculate probabilities of a binary outcome based on predictor variables, specifically our feature vectors. In this the dependent variable is categorical. Logistic regression is used to predict the probability of a binary outcome in order to classify an object. **Fisher's linear discriminant analysis** works by making a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule. The goal in LDA is to project the data into the dimension that minimizes the variance within classes while also maximizing the variance between classes. LDA is interesting because it can simultaneously perform dimensionality reduction and classification. This report will explore how these methods do in comparison with one another, in classifying movie reviews based on whether they were positive or negative.

2. Input Data

Two .txt data files are involved: one training set and one testing set. Each line in the datasets start with a label of the vector, either a 1 (positive) or a 0 (negative), followed by the word2vec vector. The word2vec model is an efficient method for computing machine-learned word embeddings. The data files given:

Name of Files and Data	Dimensions of File
testFile	10,000X101, becomes 10,000X100 after labels removed
testLabels (data extracted from testFile)	10,000X1
trainFile	40,000X101, becomes 40,000X100 after labels removed
trainLabels (data extracted from trainFile)	40,000X1

Each consists of feature vectors that start with a label (first column) and are followed by the word2vec vector that represents relevant words embedded in each review (remaining 100 columns). The vector length in total is 101, including the label and word2vec vectors. Input parameters for KNN consisted of trainFile, testFile, k, and distanceMeasure. Input parameters for Perceptron consisted of trainFile, testFile, and n. Input parameters for

SVM, Logistic Regression, and Fisher Linear Discriminant Analysis consisted of trainFile and testFile.

3. Results from KNN with different K values and distance measure

The table below summarizes how the accuracy of our KNN classifier changed as we varied the value of k, which corresponds to the number of neighbors whose vote is taken when KNN is classifying a query point. When k=1 and only one nearest neighbor is consulted, the accuracy is pretty bad (68.59%), but as more neighbors are consulted, a better picture of the query point is created and the accuracy goes up. The default k value in the classifier was k=5, but this was not enough to exceed our goal of 80% classification accuracy. As such, we continued to increase the k value until we got 80%. This actually only takes a k of 24. The test code runs a k of 100. The rest of our results are summarized below and the optimum k, 100, is highlighted.

Value of K	Accuracy %	Time to run
1	68.59%	2 minutes 45 seconds
3	72.05%	2 minutes 36 seconds
5	74.55%	2 minutes 19 seconds
7	75.6%	2 minutes 7 seconds
9	76.49%	2 minutes 5 seconds
11	77.26%	2 minutes 7 seconds
13	77.81%	2 minutes 8 seconds
15	77.97%	2 minutes 7 seconds
23	79.49%	2 minutes 8 seconds
24	80.08%	2 minutes 10 seconds
30	80.34%	2 minutes 20 seconds
40	81.04%	2 minutes 14 seconds
100	81.85%	2 minutes 20 seconds
200	80.82%	2 minutes 10 seconds

Below is a chart of how accuracy varies with different distance measures in the KNN classifier. We tried Euclidean distance, which is the default and the parameter entered in the test, and got 81.85%, the highest accuracy that we achieved in KNN. We tried other distance measures, but none of them exceeded Euclidean distance. See the table below for the rest of the results:

Distance Measure	Accuracy %	Time to run
Euclidean	81.85%	2 minutes 20 seconds
Manhattan	80.65%	2 minutes 30 seconds
Chebyshev	74.26%	2 minutes 19 seconds
Minkowski	81.85%	2 minutes 7 seconds

4. Results from Perceptron with different values of n.

In our perceptron model, we tried to increase the accuracy by optimizing the parameter n, which corresponds to the maximum number of iterations the perceptron will loop through the data points. Intuitively, we thought that increasing the number of iterations would increase accuracy, but apparently it decreases it. This may be because points that were classified correctly earlier get misclassified in subsequent iterations. Additionally, we kept getting different values every time we ran the model. Sometimes it would not clear the assertion with just the default settings. We optimized the n_iter parameter. The highest accuracy we were able to get is highlighted, and the results of the other trials are recorded below:

Value of n	Accuracy %	Time to run
5	81.01%	3 seconds

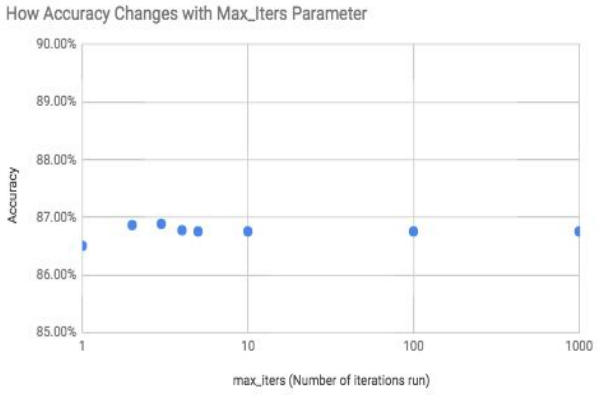
10	81.04%	3 seconds
100	80.75%	3.73 seconds
1000	79.04%	14 seconds

5. Results of our experiments

a. *Linear Logistic regression:*

We chose this model because we ran linear regression and got an accuracy of 49%. We decided to try logistic regression because it uses a probabilistic objective function, instead of a sign function. Thus, logistic regression enables us to not only classify our points, but also know how certain we are about our classification. These factors make logistic regression a stronger classifier, so we wanted to experiment with it and see if we could get good results on this data. We used "*class* sklearn.linear_model.**LogisticRegression".**

Parameter: **maximum number of iterations (max_iter)**. We chose this parameter because we were interested if we could get a better result by running more iterations. What we found out was more interesting:



Parameter Name	Parameter Value	Accuracy	Time to Run
max_iter	default (100)	86.76%	3 seconds
	1	86.51%	3 seconds
	2	86.87%	3 seconds
	3	86.89%	3 seconds
	4	86.78%	3 seconds
	5	86.76%	3 seconds
	10	86.76%	3 seconds
	1000	86.76%	3 seconds

This is a plot of the accuracies as they changed with the different max_iters values we tried. The default value was max_iters=100 and according to sklearn documentation, the max_iters parameter refers to the maximum number of iterations necessary for the solvers to converge. After we tested it by running max_iters=1000, we were slumped because the accuracy was not better than for the default of 100. In fact, it was exactly the same: 86.76%. Then, we decided to see if we could get better results by purposely setting max_iters to lower than the default. We started at the lowest possible: would the best accuracy happen if it only had one iteration? That wouldn't make sense. Thankfully, we did get a slightly lower accuracy on that run, and the first different one yet. We were on the right track. You can see our results summarized in the table and graph above. We found that the best value was actually max_iters=3. You can see this on the graph above where the accuracy increases only slightly (to 86.89%) for max_iters=3 and then drops down on either side. Another reason we chose this parameter was because there weren't a lot of parameters we could mess with without having the .19 version of the software. Regardless, it was interesting that we got such a cool accuracy curve, and repeated identical results despite shuffling every time.

Pros and cons of this model for this data set: The Logistic Regression model got us great results. The accuracy of the optimized model, 86.89% (max_iters=3), and even the default 86.76% is high. We likely got such good results because logistic regression is easy to implement and is well normalized. In fact, one of logistic regression's famous flaws, that it is dependent on good features, actually works to our advantage in this case

because we have very good data. Additionally, logistic regression is weak on feature selection, which also works for us because that's not why we are using it. Overall, logistic regression seems like a logical (>.>) choice for classifying this data set.

b. *Support Vector Machine:*

We chose this model because unlike other models which perform classification using a decision boundary, SVM performs classification with **the best** decision boundary. SVM is able to find the optimum decision boundary by maximizing the margin around the bound. The margin is the distance between the decision boundary and its closest point on either side. Maximizing the margin is the mathematical way of finding the best decision boundary, and ensures points that are very close to the line get classified correctly. We tried the perceptron model earlier, so we think that using SVM will give us better results because it will find the best decision boundary, not just the first one that works.

We used *class* sklearn.svm.**SVC**

Parameters:

Parameter: C. We chose to vary the parameter C, which corresponds to the penalty parameter of the error term. We chose to vary this parameter because the penalty on the error term determines how quickly the learner learns from mistakes. We wanted to see if we could obtain a higher accuracy by making the penalty higher or lower than the default. We don't have a lot of intuition in this space, so our results are in the table below. The default C was 1.

Parameter Name	Parameter Value	Accuracy	Time to Run
C	default (1)	87.19%	5 minutes
	2	86.79%	6 minutes 20 sec
	.1	87.04%	5 minutes
	.5	87.36%	5 minutes 20 sec
	.7	87.33%	5 minutes
	.9	87.22%	5 minutes

We first thought that increasing C would lead to a higher accuracy, so we set it to 2 and actually got worse results. Then we decided to try if lowering the penalty on error would actually make the accuracy better. Indeed, this worked and we actually determined the optimal C for this data set to be .5.

Pros and cons of this model for this data set: SVM worked very well for this data set, with an accuracy of 87.36% after we optimized the error penalty parameter, C. This was the best result out of any of the models. SVM worked so well for this data set because due to the nature of the optimization used in the algorithm, SVM is guaranteed to produce the BEST decision boundary, not just a decision boundary, so it should have the best accuracy compared at least to other algorithms that use decision boundaries. Finally, even after it finds the best decision boundary, you can further optimize SVM by simply manipulating the penalty parameter C, or any of the other useful parameters depending on your data set. It was also very easy to implement in python, and appears to work well with our data set because of the high accuracy. SVM is reported to be bad at Natural Language Processing(NLP), but is also reported to be good with Bag of Words clustering, which is what we are doing here, even though Bag of Words is less sophisticated than NLP.

c. *Fisher Linear Discriminant Analysis:*

We chose Fisher Linear Discriminant Analysis (LDA) because we thought it would be interesting to select a Bayesian model. SVM took 5 minutes to run and only increased the accuracy by less than a percent. This was too much time for such a small increase in accuracy. We were interested in seeing if a simpler, but still very powerful, model could do better.

We used *class* sklearn.discriminant_analysis.**LinearDiscriminantAnalysis**
Parameter: **n_components:** Because Fisher Linear Discriminant analysis can perform both dimensionality reduction and classification we decided to see if we could beat the default accuracy of 86.74%, but with varied numbers of components. We predicted that increasing the number of components would increase the accuracy.

Parameter Name	Parameter Value	Accuracy	Time to Run
n_compoments	default (?)	86.74%	2.5 seconds
	1	86.74%	2.5 seconds
	20	86.74%	2.5 seconds
	99	86.74%	2.5 seconds

LDA can project your data onto a maximum of d-1 dimensions, if d is the number of dimensions in each observation. In our case, the maximum number of components LDA can project onto is (100-1=99), but as you can see from the data table above, no matter how many components for dimensionality reduction we chose, we actually got the same accuracy each time!
Pros and cons of this model for this data set: LDA was a great fit for this dataset, and the accuracy 86.74% is almost as good as SVM, which took much more time to run. LDA was great to use because it was easy to implement, and because it performs feature selection and classification. It also converges to the same answer because it has an explicit objective function to maximize. This is why we kept getting the same answer. One potential problem with using LDA is that it is hard to scale when dealing with large dimensions. Additionally, its classification power is supposed to be limited because it looks in one direction only. Despite all of these potential limitations, it appears our data has a good dimensionality for LDA, and it is a great model for this data set.

6. Conclusion

In this report, we compared the classification accuracies of five supervised learning methods: KNN, perceptron, logistic regression, SVM, and Fisher LDA. See the table below for a summary of the best results for each classifier:

Name of Classifier	Best Accuracy	Runtime
KNN	81.85%	2 minutes 20 seconds
Perceptron	81.01%	3 seconds
Logistic Regression	86.89%	3 seconds
SVM	87.36%	5 minutes 20 seconds
Fisher LDA	86.74%	2 ½ seconds

As you can see by the results, we picked some great machine learning models for our dataset. We were trying to classify movie reviews as being positive or negative in sentiment. We trained our models and optimized their parameters to try to get the best accuracy we could get. KNN was not exceptional at classifying this data, but it was interesting to see how different distance metrics affected the accuracy. The perceptron model was also fun because each time the accuracy was slightly different. This made it pretty scary because of the 80% minimum assertion, because sometimes with default parameters it wasn't clear. The model that had the best accuracy

was SVM, but it also took significantly longer than any of the other models to run. Each model had its pros and cons when it came to modeling this data set, but all of these supervised learning methods were successful in being more than 80% accurate. It was interesting comparing the models and being able to use classification accuracy as a way to evaluate models against one another. If we were to do this experiment again, we would like to have more data. Additionally, we spent a lot of time optimizing parameters, but would have loved to spend more. It would also be interesting to see if classifying the reviews by sentiment using NLP would get us better results. All in all, if we were to declare one winner for this report, it would be SVM.

Work Cited

SVM- https://en.wikipedia.org/wiki/Support_vector_machine