Sentiment Analysis on IMDb Database

Daniele Gilio

May 6, 2020

0.1 Introduction

For this assignment we are going to perform sentiment analysis on film reviews from the popular IMDb site. In order to do that we were asked to build a Naïve Bayesian Classifier able to assign a binary class (good or bad) given the word content of the review itself. The dataset is made of 25000 training samples, 12500 validation samples and 12500 test samples.

0.2 Data Preprocessing

The first thing needed to make the Naïve Bayesian Classifier work is some data processing. A crucial passage is the creation of a dictionary which we will use to build the reviews' Bag of Words representation. To do that we scan all the training documents, clean the inputs by removing unwanted characters and record the frequency of appearance of each word, then we select the n-most frequent ones. The choice of n is obviously going to affect the classifier performance. After the dictionary is built, we proceed to create the BoW representation of the documents which is basically a matrix in which columns represent the word in the dictionary and rows documents. The entries in the matrix are then each word's frequency of occurrence in each document.

0.3 Classification

We used the pvml library to build our Naïve Bayesian Classifier, which by means of Laplacian smoothing, ensures that we never encounter a zero probability. The Naïve Bayesian Classifier has a very low computation time expense since it does not require SGD optimization. On the other hand this kind of classifier might not be the most versatile and makes a strong assumption on the dataset (the data items MUST be conditionally independent). Since in reality this is not always the case, the Naïve Bayesian Classifier might suffer performance-wise if compared with other classifiers. Its simplicity though makes it a very good starting point in tasks like sentiment analysis. Given these premises we also built an SVM classifier and a Logistic Regression one. For our baseline training we chose $n = 1000$, the results we obtained can be seen in Table 1. As we can see our concerns

	Training Accuracy [%]	Validation Accuracy [%]	Test Accuracy [%]
Bayes	82.004	82.064	81.632
SVM	87.432	86.528	86.192
LogReg	86.492	85.995	85.712

Table 1: Baseline Training Results

about the performance of the Naïve Bayesian Classifier revealed to be true, even if the performance difference is of a couple of percentage points.

0.4 Classification Variants

In the assignment we were hinted to use more preprocessing techniques that may lead to performance improvements, which were Common Word Exclusion and Stemming[1]. The first one, as the name implies, consists in discarding common words which do not carry much useful information. The second one can be regarded as a normalization technique, the words are basically cut to their roots, e.g. "running" becomes "run". The results of the application of these techniques are shown in Tables 2, 3 and 4.

	Training Accuracy [%]	Validation Accuracy [%]	Test Accuracy [%]
Bayes	82.92	82.992	82.576
SVM	86.98	85.351	85.536
LogReg	85.988	84.8639	84.894

Table 2: Ignore Common Words Results

All the results above were produced while keeping every other parameter constant. We can observe that the application of the described techniques improves the performance of all the models most of the times. From now on we will keep using both since it seems that this is the most balanced choice.

[1]For this technique we used the Porter Stemming Algorithm provided with the dataset

	Training Accuracy [%]	Validation Accuracy [%]	Test Accuracy [%]
Bayes	82.32	81.94	81.624
SVM	87.44	86.236	86.048
LogReg	86.632	85.688	85.784

Table 3: Stemming Results

	Training Accuracy [%]	Validation Accuracy [%]	Test Accuracy [%]
Bayes	82.54	82.296	81.888
SVM	86.848	85.576	85.56
LogReg	86.144	85.264	85.168

Table 4: Stemming + Ignore Common Words Results

0.5 Hyperparameters

0.5.1 Dictionary Size

As we previously stated the dictionary size is a very important parameter because it carries the most information content given to the model. In order to verify that and to find the best size for our purposes we trained all the classifiers varying only the dictionary size. Figure 1 shows the plot of Test Accuracy against dictionary size. We can see a very clear peak at $n = 2500$ for the Naïve Bayesian Classifier, whereas the SVM and the LogReg

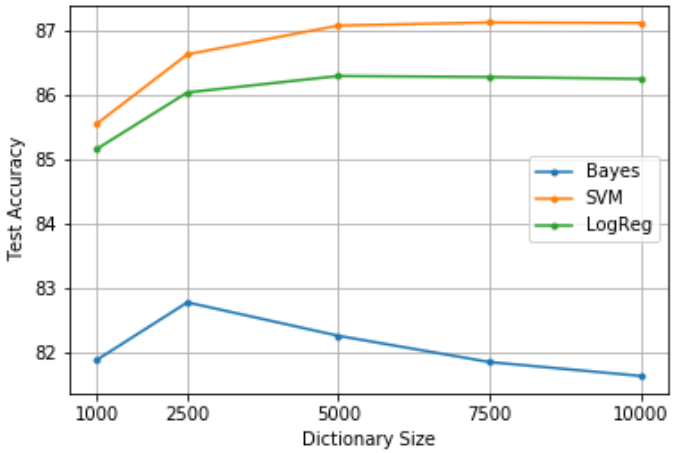

Figure 1: Test Accuracies vs Dictionary Size

increase until $n = 5000$ and then remain constant. We can conclude that the optimal values are those ones both in terms of accuracy and computation time.

0.5.2 Learning Rate, Steps and Regularization (SVM & LogReg)

After a couple of quick tests we determined that the optimal number of steps is 2000 and the regularization values are 0.0001 for the SVM and 0 for the LogReg. We did not perform a proper grid search due to the long computation time required. The Learning Rate was a little bit more challenging: we started trying with a fixed one but the results were not satisfactory, they were on par or under the Naïve Bayesian Classifier. After a little bit of online research we settled on an inverse scaling learning rate for both SVM and LogReg, basically the learning rate is initialized as $\gamma_0 = 1$ and updated according to:

$$\gamma(t) = \frac{\gamma_0}{\sqrt{t}} \tag{1}$$

where t is the number of steps. This formula makes so that the model learns very fast in the first steps and as the steps progress it reduces its learning rate. Applying this method resulted in much better results.

0.6 Results Analysis

The code we wrote is able to output the most influential words, based on the weights computed during training. The results can be then found in the "most_influential.txt" file. Taking a look at the one produced with $n = 1000$ we can see that the words meet our expectations. We find "pointless", "worst", "laughable" amongst the "bad" words and "superb", "delightful" and "excellent" amongst the "good" ones. It is to notice though that our method of spotting these words might not be the best. We basically compute the difference between the positive score and the negative score, the words with the most positive difference are the most polarizing "good" words and viceversa. This method seems to introduce first names amongst the most influential words, with increasing frequency as the dictionary size increases. This is true even for $n = 1000$ as we find "Stewart" amongst the "good" words. Regarding the wrong predictions made by the model, our code produces a file called "wrong_classifications.txt" containing the path of the reviews which the model classifies with the highest confidence (again computed via the score difference). Reading them we find that even we would struggle classifying them, we would define them borderline because they contain both "good" and "bad" words. Some of them are sarcastic so it is understandable that the classifier struggles to correctly classify them.

0.7 Conclusions

Based on the results we presented we can confirm that the Naïve Bayesian Classifier is a feasible choice for a fast first approach to a problem. The SVM proved to be the best classifier amongst the ones we tested, with LogReg taking the second place. It is worth noticing that both SVM and LogReg take a long time to train, mostly because of the huge dimensionality of the BoW matrix. In conclusion we would like to stress the fact that in this case the Naïve Bayesian Classifier is a great choice if one seeks a fast but not so precise solution, but if one wants more precision and can bear with the computation time it is better to go with SVM.

0.8 Appendix: Code Modifications

In order to speed things up we made a modification to the code written during the lab session. We used sets instead of lists to load the dictionary. This was done to speed up the creation of the BoWs since the "in" function in Python takes a time that goes with $O(n)$ if used with lists and dictionaries whereas it goes with $O(1)$ with sets. The other modification we did was to the pvml library modules for the SVM and the LogReg. We substituted the Numpy library with Cupy, which is basically a numpy implementation that exploits CUDA GPUs capabilities. The first modification produced a $\sim$ 1.6x increase whereas the second one increased speed by $\sim$ 4x across the board.

I affirm that this report is the result of my own work and that I did not share any part of it with anyone else except the teacher.