Data Challenge 1 - ArXiv Article Classification

Fundamentals of Machine Learning

Matteo Esposito (20173298)

University of Montreal

0 Introduction

This report will cover the efforts related to the classification of articles on the popular scientific research paper distribution website, `arxiv.org`.

More specifically, given the abstract of several thousand papers, our goal was to classify them into one of the following categories: astro-ph.GA, math.AP, astro-ph.CO, math.CO, stat.ML, cs.LG, gr-qc, astro-ph, astro-ph.SR, hep-th, physics.optics, hep-ph, cond-mat.mtrl-sci, cond-mat.mes-hall, quant-ph.

Four types of classifiers were used for the submissions. A random classifier which would randomly assign a class label to each observation with average accuracy of around 6.6%, followed by a bernoulli and multinomial naive bayes classifier which had average prediction accuracies of 77% and 79% respectively and a support vector machine classifier with an average accuracy of 74%.

1 Feature Design

Since at the lowest level we were interested in the count of unique words in our training set abstracts, our feature design could be seen as a dataset condensing/cleaning exercise. All efforts to tidy the dataset were done to reduce the number of unique words/strings (grouping of string characters between spaces) down from over 65,000 to 21,253 which sped up computation times significantly.

Note: All cleaning functions were implemented using the following syntax (adapted from the TA's OH notebook):

```
df[t] = df[t].apply(lambda x : re.sub("<regex pattern>", " ", x))
```

More specifically, the following changes were done:

1. Remove newline characters
2. Remove punctuation
3. Remove dollar signs and any mathematical symbols (very important given that many abstracts had mathematical terms and functions)
4. Remove standalone integers
5. Make all strings lowercase
6. Strip all strings of spaces (using x.strip())
7. Remove all common stopwords

2 Algorithms

The algorithms considered and implemented in this kaggle were the following:

1. **Random Classifier**. Randomly assign class labels from the training set to the test set observations.

2. **Bernoulli Naive Bayes Classifier** using base python and numpy. Binarize data (1 if the word is in the given abstract, otherwise 0) for all unique words in vocabulary. Calculate prior and likelihoods/conditional probabilities to get class probabilities (posteriors). Assign class associated with max posterior probability as predicted class.

3. **Multinomial Naive Bayes Classifier** from the sklearn library including the use of CountVectorizer, TfidfTransformer and Pipeline. Same as above, however, we are interested in the actual number of times a given word appears in an abstract.

4. **Support Vector Machine Classifier** from the sklearn library including the use of CountVectorizer, TfidfTransformer and Pipeline.

3 Methodology

For all experiments, a train/validation split of 70/30 was used. When it came time to submit a final submission, I trained the models on the entire training set (i.e. a split of 100/0) in an effort to maximize the generalizability of the model being used.

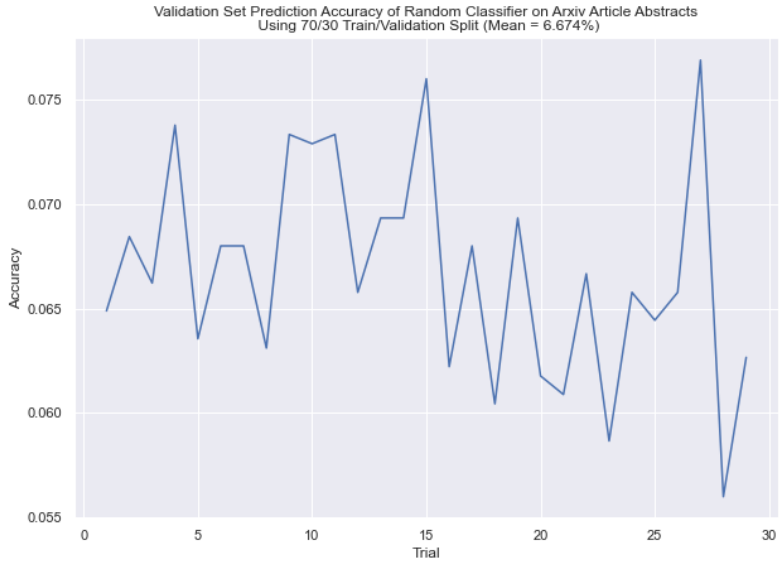
The distribution choice for the NB classifier was predetermined to be Bernoulli.

There was no regularization strategy (aside from a naive grid search in the SVM implementation), however, to optimize the hyperparameters (namely alpha in the case of the NB classifiers), a grid search was implemented. The results of these grid searches can be seen in the results section of this report.

4 Results

As detailed in Figure 1, the results for the random classifier were very poor. The average prediction accuracy of roughly 6.6% is reasonable in our application since we are randomly assigning 1 of 15 labels with equal weight. $\frac{1}{15} = 0.066 = 6.6\%$. The kaggle result for this submission was 0.06755, not beating the random baseline.

Figure 1: Random Classifier Accuracy Comparison



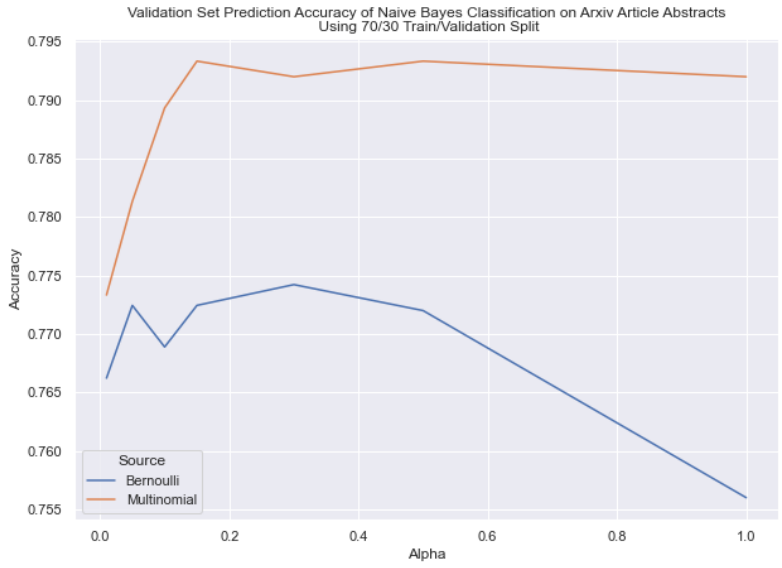
Performing a gridsearch on alpha values among [0.01, 0.05, 0.1, 0.15, 0.3, 0.5, 1], we observed that the optimal value for Bernoulli NB was 0.3 and 0.15 for Multinomial NB. They yielded the prediction accuracies below:

Table 1: Table of Naive Bayes Prediction Accuracies

Model	α	**In Development**	**Public Leaderboard**	**Private Leaderboard**
Bernoulli NB	0.3	0.774	0.778	0.781
Multinomial NB	0.15	0.793	0.801	0.794

The Bernoulli and Multinomial scores beat the Numpy only and best baselines respectively.

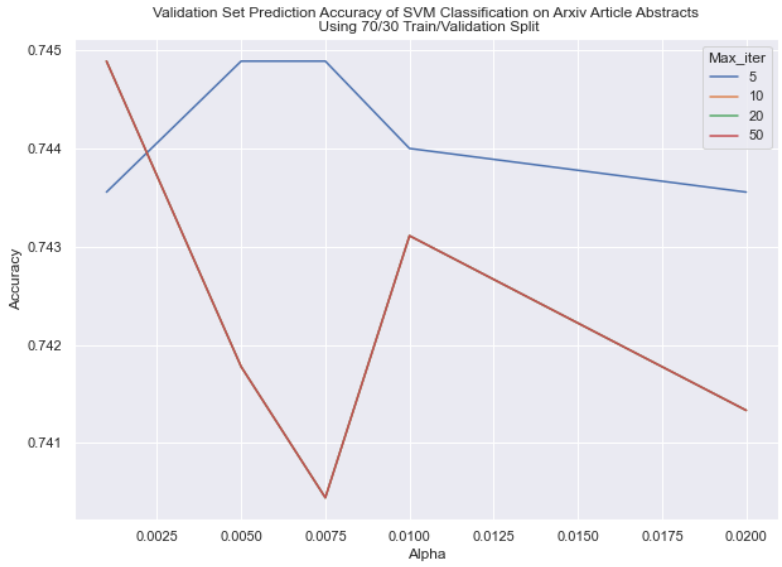
Figure 2: Bernoulli NB and Multinomial NB Accuracy Comparison



The final and second of the experimental algorithms implemented was a linear support vertor machine using stochastic gradient descent by means of the SGDClassifier module of the sklearn.linear model package. Here we used hinge loss, l2 regularization and preformed a grid search of alpha (regularization multiplier) terms among $[0.001, 0.005, 0.0075, 0.01, 0.02]$ and a grid search among max iteration/epoch values of $[5, 10, 20, 50]$. The optimal parameters yielded an average prediction accuracy of around 74%.

This model pipeline had the exact same preprocessing steps as in the multinomial nb implementation, namely the use of the cleaned data from the bernoulli nb implementation and the use of CountVectorizer and TfidfTransformer.

Figure 3: SVM Grid Search Results



5 Discussion

As a general observation, we could have used ensembling techniques to improve the accuracy of the model as none were actually used in all the work that was done in this data challenge.

Known pros of the **Naive Bayes classification** include the fact that it is good with independent data and is particularly well suited to tasks such as text classification. The downsides of NB come at runtime where applying smoothing is an additional step in the conditional probability/likelihood calculation step which adds to the runtime. Sparsity and very small probabilities are also an issue in NB, to which we present a potential solution below.

An area that could have significantly improved model performance here would be word stemming. This is the process by which we reduce inflected or derived words to their base or root form (source: `https://en.wikipedia.org/wiki/Stemming`). This would have a 2 fold effect. It would reduce the size of our dictionary which even after all of the data cleaning steps performed, was still over 21,000 words long. In consequence, it would increase the training and predicting speed to allow for more testing and experiements.

Concerning **SVMs**, a pro would be that it is particularly well suited for high dimensional data, which is very much the situation that we are in. 15000 observations and over 21000 features in our

testing set can be considered high dimensional. SVM is good for high dimensional data since, unlike other algorithms, it doesn't need to evaluate every single point in the data set to draw its decision boundary. It observes a subset that will build its support vectors so that it can set the best decision boundary.

On the other hand, a con would be that there were a large choice of hyperparameters to modulate and optimize. A very small amount of time was invested into optimizing these since at this point, I had already beaten the best baseline with my multinomial NB implementation. In particular, modulating the choice of kernel would likely yield a better decision boundary and therefore, a better ability to generalize.

For the SVM implementation, since the maximum number of iterations hyperparameter did not have any postive effect on the prediction accuracy when at a value of 10 or greater, perhaps an improvement could be seen with maximum number of iterations values less than 10. The only value lower than 10 tested was 5 which yielded up to a 0.5% improvement in predictions. The reasonable alpha parameters seem to be between 0 and 0.01 and therefore the range of alpha values selected for the grid search can remain and are appropriate.

6 References

1. Multinomial NB documentation: `https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html` and `https://medium.com/@anuuz.soni/pros-and-cons-of-naive-bayes-classifier-40b67249ae8`

2. SGD Classifier documentation: `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html`

3. Multinomial NB Implementation: `https://towardsdatascience.com/multinomial-naive-bayes-classifier-for-text-analysis-python-8dd6825ece67`

4. NLP Implementation: `https://towardsdatascience.com/machine-learning-nlp-text-classific using-scikit-learn-python-and-nltk-c52b92a7c73a`