Jacob Dineen

Text Mining

8/24/2018

Homework 6

\*I’ve attached my ipynb file in the portal. Any code note referenced in the appendix can be seen there.

The goal of this assignment is to experiment with Sklearn’s functional API for statistical computation and mathematical modeling, along with exploring different vectorization techniques and hyperparameter tuning. We will focus on methods classified under Bayesian inference, particularly Bernoulli and Multinomial naïve bayes, for text classification. Aside from learning to approximate a feature map based on posteriors, we will also be exploring feature ranking (odds ratios), and theorizing performance boosts based on specified vectorization methods and tuning.

P(class|X) = P(X|Class) \* P(class) / P(X)

Or

Posterior = Conditional \* Prior

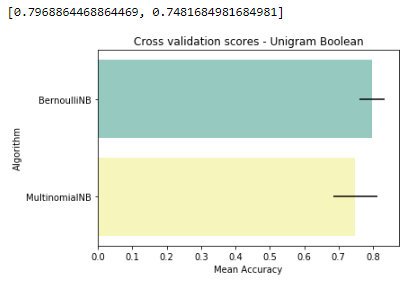
Having worked in Scikit quite a bit, I’d like to conduct this experiment using several concepts that we’ve explored throughout this course, as well as touching on some methods/functions that we haven’t. I will explore the use of pipelines, parameter grid searching and kfold cross validation on each sub-classification problem; Detecting Polarity of a review, as well as detecting authenticity.

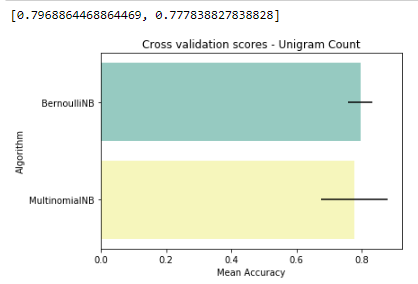
Excerpt from homework 4: The data itself contains two dependent variables; Lie and Sentiment. Each of these binarized classes will partitioned and used as isolates for two separate classification tasks.

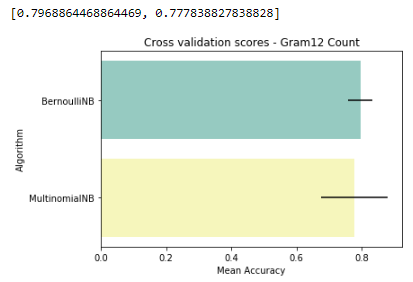
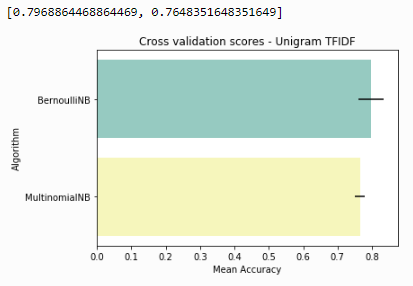
1. Predict the sentiment of a review (Positive/Negative).
2. Predict whether a review is truthful, or a lie (Fake/True).

The data is represented, initially, by a 3x92 matrix, meaning there are three classes and 92 instances of each class. In other words, we have 92 reviews, which would categorize this dataset as small and prone to overfitting [Appendix 1.1]. All dependent variables follow a uniform distribution, so we don’t need to worry about stratified sampling to preserve some idea of class balance. We can also use a baseline of 50% accuracy, which coincides with both random guess and majority vote.

I wanted to start with the sentiment analysis portion of the dataset – To do so, I built a mini pipeline to graph out Bernoulli vs Multinomial Naïve Bayes against a variety of different vectorizers using Kfold cross validation, where k = 5:



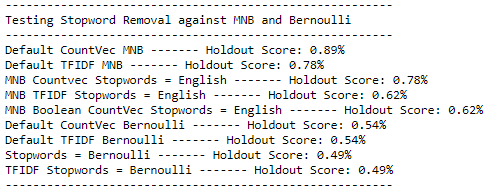




The results generally back the theory, albeit at a small scale, that Boolean vectorizers work better for Bernoulli models, as evidenced by the Multinomial Naïve Bayes model performing better (max performance) with both the Unigram and Bigram *count* vectorizers. With term frequency vectorization, however, the Bernoulli model outperformed the MNB in terms of max score, and average score. Interestingly, the average score for all Bernoulli models was higher than the average score for all MNB models. My guess on why this is occurring here is because the length of the total documents, 92, is not sufficient enough to create a dense representation regardless. Even converting from binary to counts is likely going to form the same sparse representation of the total vocabulary seen within the documents. The best performing model of the above was the MNB model with either unigram or bigram vectorization.

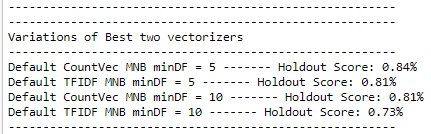
As I ventured into Pipelines with Sklearn, I realized that I have very little experience building out complex architectures – I initially wanted to build out a pipeline that could iterate through different vectorizations options and then, ultimately, gridsearch parameters. The latter was a futile task, however, as both the Bernoulli and Multinomial naïve bayes algorithms have very few parameters to tweak, side from alpha, which is the smoothing parameter that we have become accustomed to seeing in some of the reading materials. I’ve instead decided to build out individual pipelines for each variation of a vectorizer. This is a bit more hardcoding than I had hoped to do, but please do let me know, whenever reading this, if there’s a better way of completing this task (work in app. 1.2).

The first test I wanted to conduct was stopword removal:



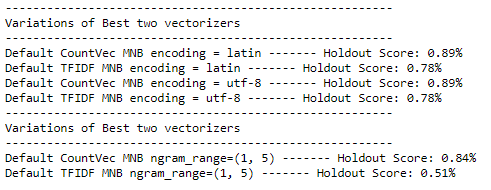
The above table shows test scores with descriptions of various vectorization techniques and induction algorithms. Multinomial Naïve Bayes proves to be the more generalizable algorithm for this dataset, in any regard, but we do see that stopword removal hinders performance against default vectorizers.

When fitting a model on a holdout set, you’d like a representation that is robust enough to generalize to unseen data. I wanted to toy around with min\_df, a parameter that ignores words in a vocabulary that do not meet a certain threshold. Theory lends me the intuition that a higher threshold may lead to overfitting on the training set, and underfitting on the test set, as there will be a smaller vocabulary and a greater likelihood that an instance in the test set isn’t represented in the model:

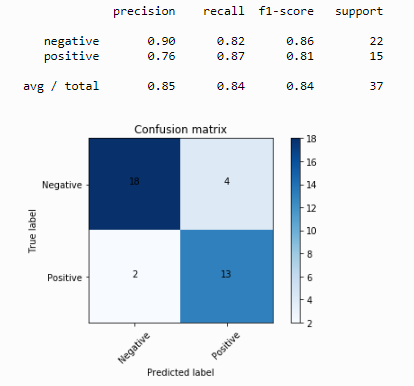


In this case, I was right. Increasing the threshold did have a negative impact on the test performance.

I also tested the encoding schema, measuring the impact of encoding in utf-8 or latin-1, but there was no difference in performance. Lastly, I tested the impact of ngrams on performance by setting a range of 1-5. With such a small dataset, I didn’t have to worry about the impact on compute, but I was worried about intentionally decreasing the denseness of my vectorization schema. Both of my best models dropped from their best run, however the Tfidf architecture really suffered with the introduction of ngrams.



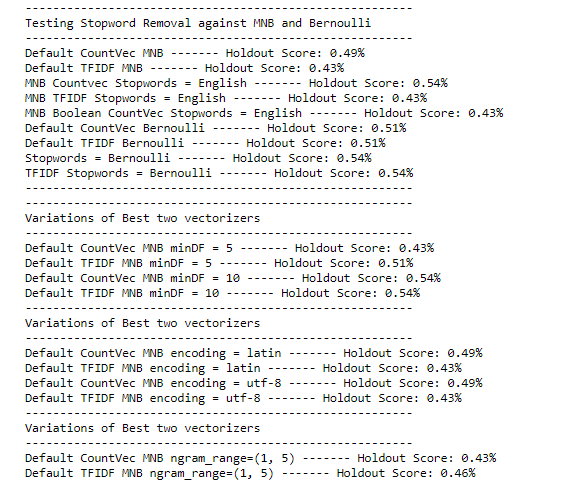
All in all, my best performing model was found using a default countvectorizer, along with multinomial naïve bayes – My train test split was 70/30, so I could have potentially tried to shift to a more uniform distribution (Edit- I went back and updated to a 60/40 split and saw better results). The below classification report and confusion matrix show us that we had exceedingly high precision when it came to the negative class, and very high recall with the positive class.



4 out of the 6 errors came when predicting a negative label as positive [App 1.3]. One of these errors appears to be happening due to a bad job labelling, but the other three are excessively long pieces of text that have both positive and negative adjectives, so it’s likely that the algorithm was misguided in these class mappings. Looking at our top feature [App. 1.4], based on the odds ratios for feature ranking a Bayesian model, it isn’t incredibly intuitive. Most of the highly ranked words are nouns rather than adjectives or action words, and the theory would lead me to believe that with a dataset of this size Boolean vectors would work better, but that isn’t the case. This is where sparsity and the lack of scale come back to hurt us.

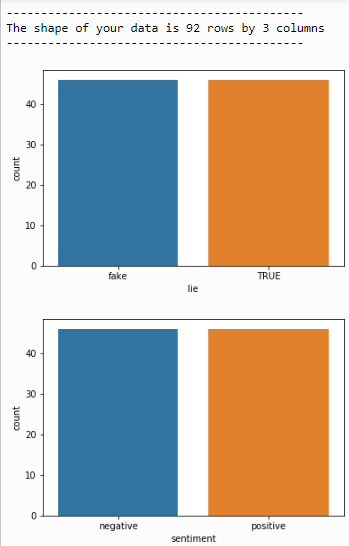
I’ve spent most of this essay writing about the sentiment classification task, mainly because I believe the efforts to classify authenticity would be ineffective. I’d still like to push these through the same pipeline that I used for the above analysis and see if we notice any shift near or above the majority vote baseline.

Unsurprisingly, we’re met with comparable results to what we saw during training w/ crossfold validation, although it does seem like Bernoulli naïve bayes outperformed MNB in most instances – This is probably meaningless because we have a bad model anyway:



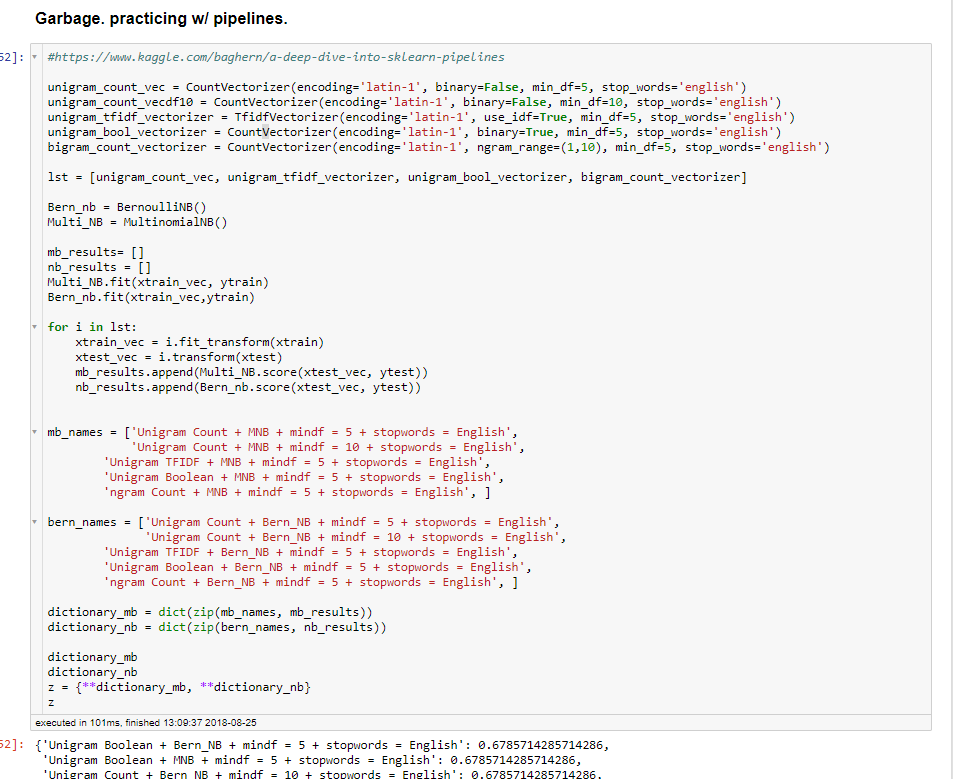
**Appendix**

* 1. **– Showing data distribution**

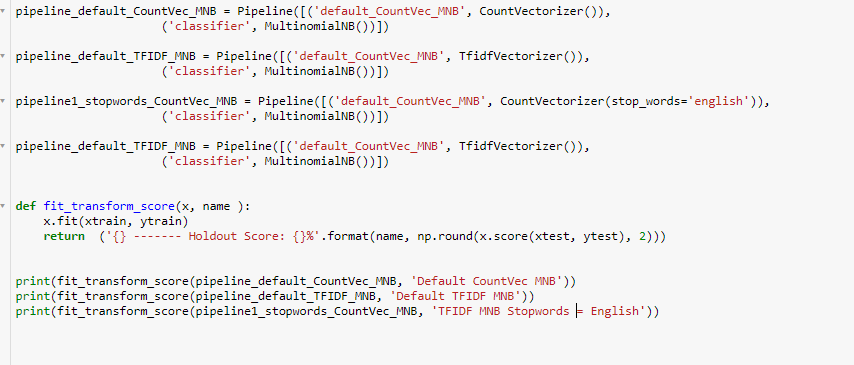


* 1. **– Bulk pipelines**

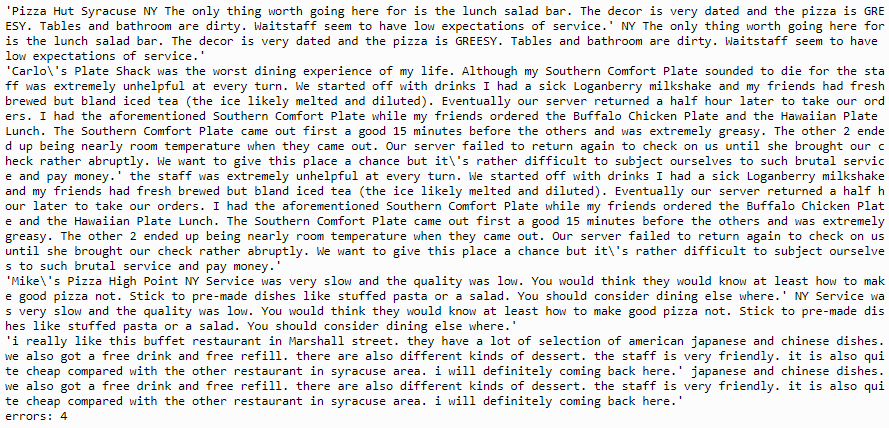
Initially tried iterating through a list of vectorizers, but this was too clunky:



So I moved back to the Sklearn API for pipelines and wrote a little function that handles that fitting/transforming/scoring:



* 1. **– False Positives:**



* 1. **Feauture Ranking:**

