William Lifferth
wileliff@vols.utk.edu

Naïve Bayes Classification of News Articles

Collaborators

This assignment was approached and completed individually, and I didn't have any particularly meaningful collaborations with any other students. Two students that I spoke briefly with about the project were Nick Skuda and Matt Gallagher. This "collaboration" was limited to the occasional conversation regarding their progress on the project, and a single conversation about with Matt Gallagher about one-hot encoding after a UTK Machine Learning club meeting.

Approach

The motivation of the project was to gain experience with relevant technologies such as PySpark, as well as explore how using a technology such as PySpark compares to other methods such as the SciKit-Learn in both performance and usability. The project specifically looked at a corpus of labeled news article and applied a Naïve Bayes approach to classify them based on their topic or focus. For the purpose of this assignment the topics were limited to the following topics: *money, fx, crude, grain, trade, interest, wheat, ship, corn, oil, dlr, gas, oilseed, supply, sugar, gnp, coffee, veg, gold, soybean, bop, livestock, cpi*.

Preprocessing

The dataset was fairly messy to begin with, so it required a decent amount of preprocessing work. This work was completed using Jupyter Notebooks, and the BeautifulSoup library. Because I opted to use the most recent stable version of Python (3.6) the SGML library named in the instructions was not available to me; BeautifulSoup is well known and well supported so it was a good option.

Using the python Operating Systems library I grabbed the filenames of all the files in the relevant directory, and only performed the following actions on those that matched the first string of `reut2` as all the correct .sgm files did. I couched all the following operations in `try` blocks, such that errors in one file would not mess up the entire program. One issue I ran into was the fact that the files are not technically correct XML; correct XML files must contain only a single XML root tag (these files contain each reuters XML object as it's own top-level tag). I got around this by first surrounding every file in a tag labeled `<ABSROOT>`.

After performing this first cleaning step the file's text was read into a BeautifulSoup object. I appended the `<BODY>` and `<D>` tags for each article to a body and topic python list respectively.

This was performed with the following code snippet (pulled from a Jupyter Notebook).

```python
body = []
topic = []
dir_name = 'reuters21578'
for fn in os.listdir(dir_name):
    if fn[:5] == 'reut2':
        try:
            file = open(dir_name + '/' + fn)
            content = file.read()
            content = content[:36] + "<ABSROOT>" + content[36:] + "</ABSROOT>"
            soup = BeautifulSoup(content, "xml")
            article_objects = soup.find_all('REUTERS')
            for a in article_objects:
                try:
                    b = a.find_all('BODY')
                    t = a.find_all('D')
                    if len(b) >= 1 and len(t) >= 1:
                        body.append(b[0].get_text().replace('\n', ' '))
                        topic.append(t[0].get_text())
                except:
                    pass
        except:
            pass
```

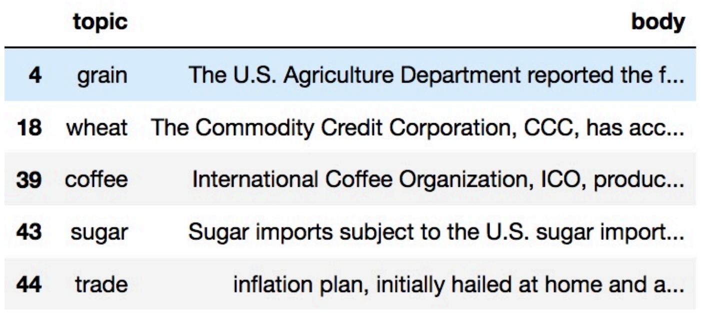
After this initial preprocessing was performed the body and topic lists were combined into a Pandas DataFrame object for easy manipulation. I then created a `topics_of_interest` list that held all the defined topic names this project listed. This was accomplished easily in the DataFrame format the data was in, as shown below.

```python
df = df[df['topic'].isin(topics_of_interest)]
```

Prior to this step the DataFrame included articles with all kinds of topics, as shown below.

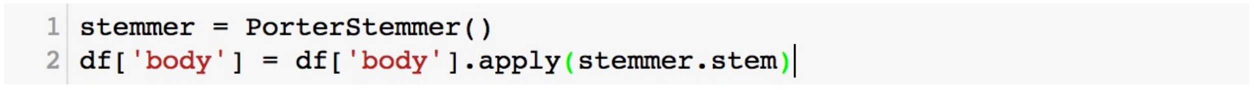
	topic	body
0	cocoa	Showers continued throughout the week in the B...
1	usa	Standard Oil Co and BP North America Inc said ...
2	usa	Texas Commerce Bancshares Inc's Texas Commerce...
3	usa	BankAmerica Corp is not under pressure to act ...
4	grain	The U.S. Agriculture Department reported the f...

After this step the DataFrame filtered out all the articles with topics not in our list `topics_of_interest`, and only the topics we still had substantial interest in were left, as illustrated in the next figure. After this corpus was saved into a csv file called `data.csv` using the Pandas `to_csv` function.



	topic	body
4	grain	The U.S. Agriculture Department reported the f...
18	wheat	The Commodity Credit Corporation, CCC, has acc...
39	coffee	International Coffee Organization, ICO, produc...
43	sugar	Sugar imports subject to the U.S. sugar import...
44	trade	inflation plan, initially hailed at home and a...

The next step as outlined in the project specification involved "stemming" our textual data such that similar words are not counted as distinct due to different endings that ultimately do not change the meaning of the word. It was recommended we use the NLTK library from python, which is what I did. After importing the Porter stemmer from the NLTK library the actual invocation of the stemmer was as simple as the following:



```
stemmer = PorterStemmer()
df['body'] = df['body'].apply(stemmer.stem)
```

At this point the corpus was saved again into csv file, this time called `training_test_data.txt` (per the instructions of the project specification). Again, per the instructions of the project specification, the first ten lines of this file were printed into another text file. I chose to call this file `first-ten.txt`.

The next step was to perform a TFIDF transformation on the data such that it could be easily used for machine learning (specifically the Naïve Bayes model). I was tasked with both performing the TFIDF transform with a non-spark library (I selected scikit learn) and PySpark. The process for both was fairly similar, but they both had quite different runtimes.

After TFIDF I completed in the final preprocessing steps before testing and training could be carried out. This involved using the StringIndexer object from the `pyspark.ml.feature` library to turn the string labels for topics into integers. It was necessary to release several of the models from memory after preprocessing had been completed to avoid using up the entirety of allocated memory.

Using the `NaiveBayes` model from `pyspark.ml.classification` and the `MultiClassificationEvaluator` from `pyspark.ml.evaluation` I then trained and tested a model. For the purpose of integrity I first did a 90%/10% split to pull out a 10% validation set, then defined the following splits for the rest of the data (to keep with them being 50%, 60%, and 80% of the remaining data) was 0.55555, 0.66666, and 0.77777. For organization the results of this can be found in the Results section of the paper. Out of personal interest I wanted to know how this trend would generalize over more extreme test/train splits, and did so for splits from 5% to 95% (with the remaining comprising the test set).

Results

For scikit learn I imported the Term Frequency Inverse Document Frequency Vectorizer class from ` from sklearn.feature_extraction.text import TfidfVectorizer`. Scikit learn is built to interact natively with Pandas dataframes, so the actual fitting and transforming of the `body` column could be done in one line. I used the python `time` library to time the running of the actual TFIDF command. It clocked in at **~0.5838239s**.

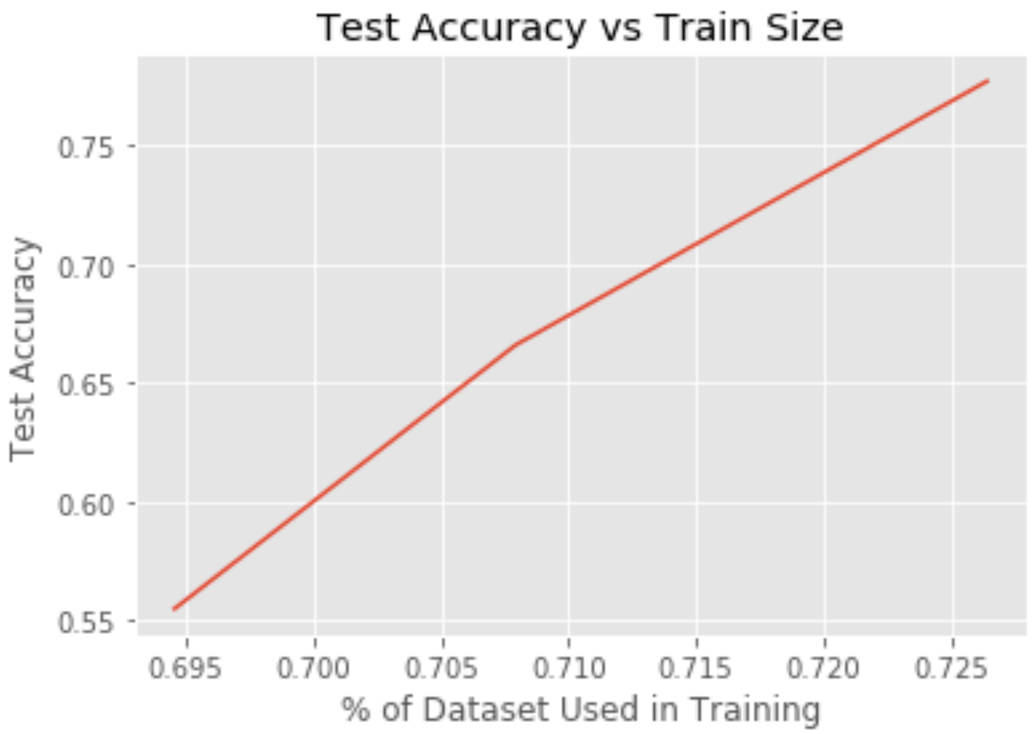
I then performed a fairly similar process in PySpark, importing the following libraries ` from pyspark.ml.feature import HashingTF, IDF, Tokenizer`, to perform the same task. In PySpark some of the operations seemed syntactically less direct, but overall the procedure was the same. Again I created a TFIDF vector with a Spark dataframe, and time it as well. For the PySpark implementation the total runtime was **~0.010882s**.

The difference in runtime between the scikit learn implementation and the PySpark implementation was fairly big. The longer of the two (SciKit Learn) ran for more than half a second, 0.5838239, while the faster (PySpark) only took 0.010882. The scikit learn implementation was slower than PySpark by a factor of 52.65—having the backend running in Spark had a non-negligible impact on the runtime of the operation.

The results of training can be seen below.

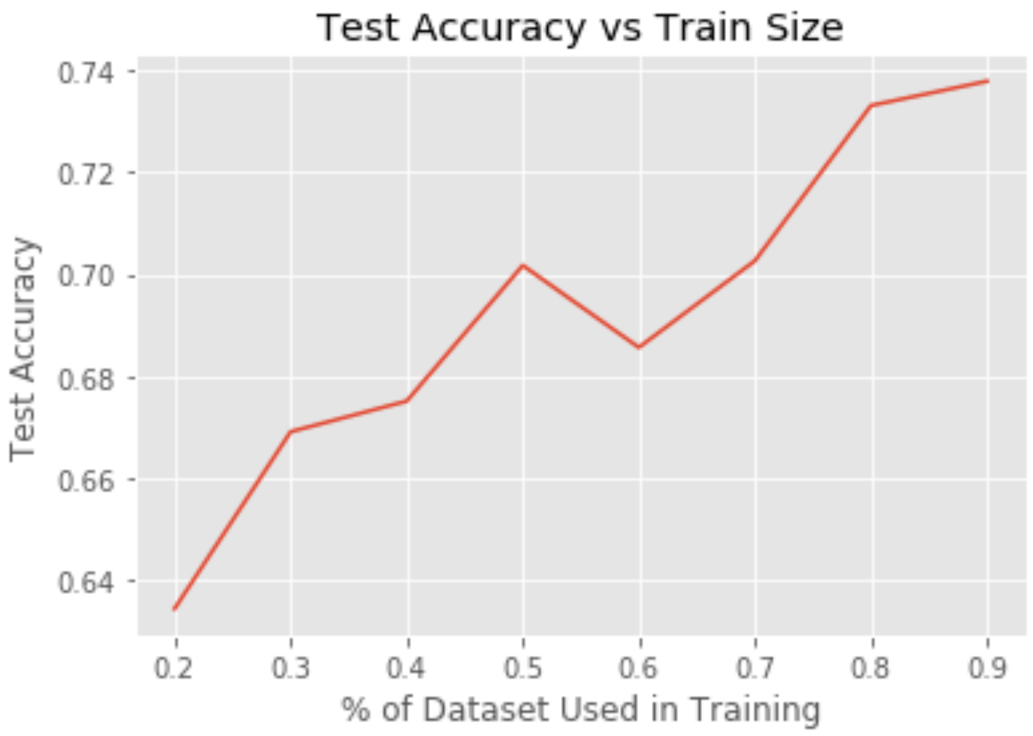
	50% training set	60% training set	70% training set
Run 1	0.6953441295546559	0.6823056300268097	0.7225433526011561
Run 2	0.6929375639713409	0.7064220183486238	0.7406015037593985
Run 3	0.697841726618705	0.7202216066481995	0.731610337972167
Run 4	0.6859421734795613	0.7191780821917808	0.7351778656126482
Run 5	0.6985365853658536	0.6934210526315789	0.7131630648330058
Run 6	0.7099697885196374	0.68717277486911	0.7196078431372549
Run 7	0.6751893939393939	0.7124183006535948	0.7641509433962265
Run 8	0.6705539358600583	0.7203728362183754	0.7125506072874493
Run 9	0.7173252279635258	0.7188703465982028	0.7170172084130019
Run 10	0.7018572825024438	0.7188755020080321	0.707635009310987
Average	0.6945497807775177	0.7079258150194307	0.7264057736323296

The tabular data above is plotted below for easy visualization. The graph was made with the matplotlib python library.



There are a couple notable things to call out. First of all, the best accuracy of any Model was ~0.7642. Given that we had 23 potential labels (one for each of our 23 topics), random guessing would've given us an accuracy of ~0.0435 (1/23). So it is an understatement to say that our model is doing better than random guessing. Even with default configurations without much tuning, our model is able to perform admirably well. It is also clear that as we increase the amount and proportion of data the model is allowed to train from, our overall accuracy goes up.

I was interested in this trend, so I additionally explored the effect of training size on accuracy at more extreme ranges, as shown in the graph below. I only performed testing and training on 2 rounds per split, so it is clearly a more noisy result set, but the trend is almost as clear. More data increases accuracy, especially on the lower end. The first 10-20% increase in dataset size grants a huge leap in accuracy, whereas the last 10% increase in dataset size only gives an additional 1% of accuracy, if that. So it seems that more data leads to better results but gives diminishing marginal returns.


Test Accuracy vs Train Size
Test Accuracy
0.74
0.72
0.70
0.68
0.66
0.64
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
% of Dataset Used in Training