Report on Determination of Authorship

Soumyakanti Das

April 24, 2019

1 Introduction

In this lab, I had to develop Decision Tree(DT) and Logistic Regression(LR) classifiers and train them to determine author of a test text of 250 lines. For this, I downloaded freely available novels of Arthur Conan Doyle, Herman Melville, and Jane Austen from Project Gutenberg. The novels are saved in the folder `./data`, and further segregated by author names.

Besides this report, I will attach two python files, `lab2.py` and `classifiers.py`. `lab2.py` is the main python file for the lab2, which also deals with data processing and feature extraction, while `classifiers.py` has code for the classifiers.

2 Preprocessing and Feature Selection

The `/data` folder contains three folders `acd`, `hm`, and `ja`, which are short for the respective author's name. Each of these folders contain novels of the respective authors. For each author, a list of words is created which contains every word in all novels combined. Then this list is added as a value to a dictionary where authors are keys. This is done in the functions `create_dataset`, `read_file`. The returned dictionary is then passed to the function `stack_dataset` which reshapes the list of words for each author into rows of 250 words. This dictionary of stacked dataset is then passed to the function `processed_dataset` where various features are extracted from each row a 2D numpy array of features is created, where the last column is the target.

Some of the features I have tried are,

1. punctuation count
2. count of words with freq = f (1, 2, etc)
3. frequency of a particular word
4. number of unique words, or vacoabulary
5. building a word count vector of some popular words and computing cosine distance with another vector
6. building a character count vector of len 26 and computing cosine distance with another vector of similar size of 1s
7. average length of words
8. count of words with length above n (8, 9, etc)

Initially I tried to only consider one feature at a time, to see how powerful that particular feature is. Given below are some of the resutls I observed.

In item 1, I have tried using various combination of punctuations. Using all common punctuations reduced the accuracy to less that 45%, while using only **,** (comma) increased the accuracy

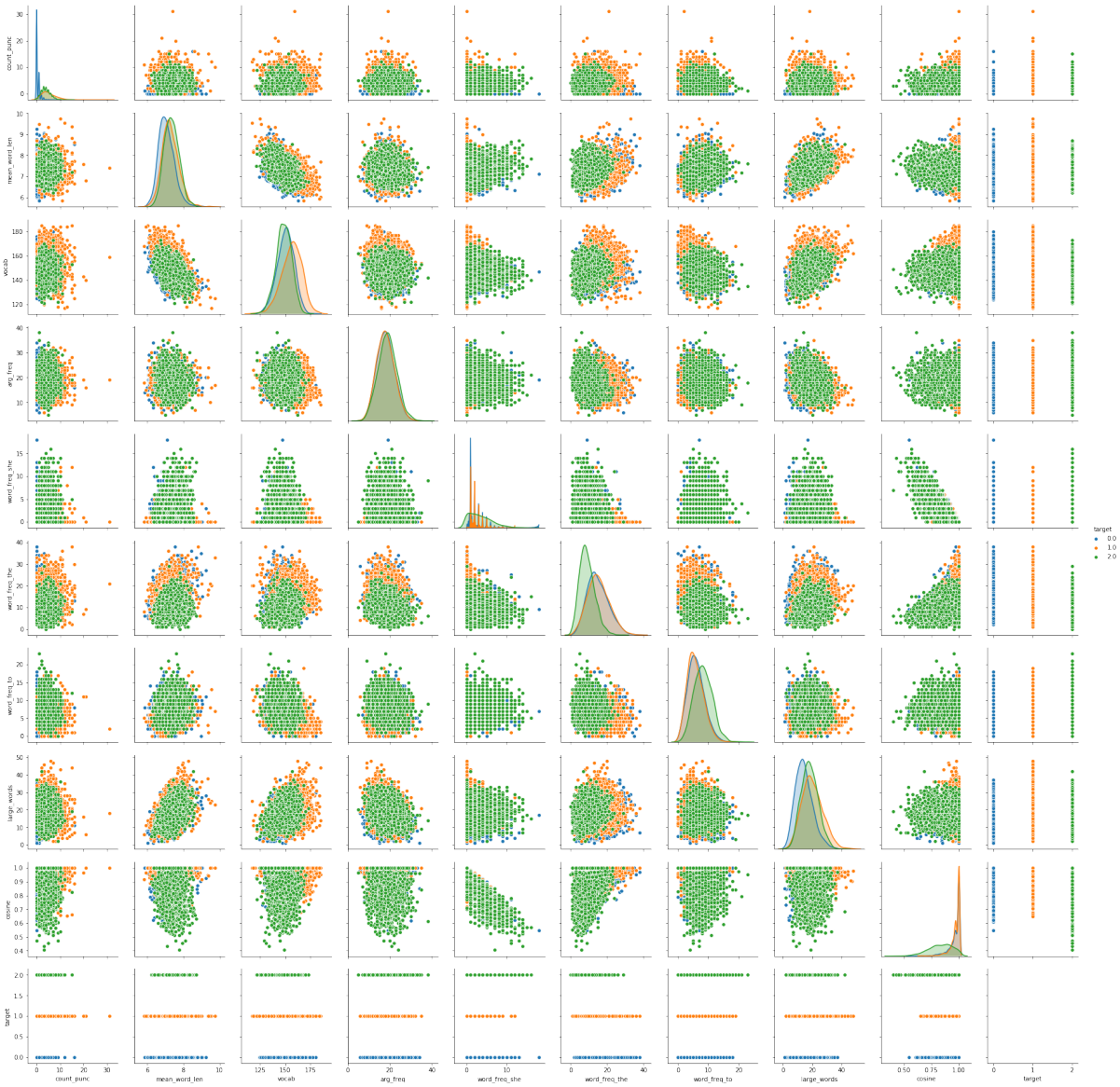

Figure 1: pairplot of features

to 50%. After trying different combinations, I found that **;:** (colon, semi-colon) increased the accuracy of classifying all three authors to around 65%. As can be seen in the diagonal of Figure 1, the 1st subfigure show the distribution of this particular feature, `count_punc`, and it can be seen that the distributions are not overlapping, and thus easier for a classifier to classify the dataset.

For item 2, I calculate the number of words with a certain frequency. When f = 1, I calculate the number of words that appear only once in the sequence of 250 words. For f = 1, I get around 44% accuracy, and with f = 2, the number is down to 33% or random chance. I could not get good accuracy with this feature, so I didn't use it for my final set of features. In the diagonal of Figure 1, it can be seen that for `arg_freq`, the distributions are overlapping almost completely, making them indistinguishable.

For item 3, word frequency, I found that using word frequency of **she** and **the** produces good results. Both of these features, when taken alone, gives accuracy of about 52 - 55% (which is much greater than random chance because I'm using all three authors). The word **to** also can be used, however, it gives an accuracy of around 47%, and thus, I decided not to use it, but I'm using the other two for my final feature set. It can be seen in Figure 1, that the distributions agree with the observed accuracies, with `word_freq_the` being most distinguishable. I would like to mention that coming up with these words took some time. I had to calculate word frequencies for each author and look for words which are commonly used by 1 author but not so commonly used by others. For example, I found that Arthur Conan Doyle and Herman Melville used "the" almost twice as often as Jane Austen, at least for the novels I selected.

For item 4, vocabulary, number of unique words in a row is calculated. Using this feature gives an accuracy of around 45%. Also, it can be seen from Figure 1 that the distributions for `vocab` are not exactly overlapping, but are not separated well. I have used this feature for my final feature set, as using it does boosts the final accuracy by a couple of points.

For item 5, cosine distance is calculated between two vectors. First, I choose a list of 10 words, such that some of them are commonly used by all three authors, while others are commonly used by only two or one of the authors. The idea is to choose a few words which are common to all authors and some words which can differentiate the authors. Then, I calculate frequency of these words, store them in a numpy array and normalize them by dividing them with the sum of the array. For the second vector, I choose three words from the list of 10 words. Then, I build a the second vector such that for the three words, the value at their index is 1, and the other values are 0s. Then I normalize the seocnd vector and calculate cosine distance between these 10 dimensional vectors. The resulting feature performs quite well, with an accuracy of over 55%. Also, from Figure 1, it can be seen that the distributions are different from each other for **cosine**. This feature took a long time to get right because of the number of choices for the words. First, coming up with a list of 10 words was a challenge and then choosing the 3 words for the second vector was also a task. What I observed was, choosing most popular words among all authors for the second vector decreases accuracy drastically. This could be because the two vectors become more similar, and angle between them is always less. Also, choosing least popular words decreased accuracy too, so finding a set of words was like a balancing act. Although, I may not have chosen the optimum set of words, this feature boosts accuracy by a couple of points when used with other features, and it is always used in second or third level of the DT classifier in multiple branches, confirming its importance.

For item 6, I tried to do something similar to item 5, but with character counts instead of word count, i.e, building two vectors of length 26. But I didn't achieve good results with it, so I didn't use it.

Item 7 is a straightforward feature where I calculate average length of words for each row, and it gives an accuracy of around 43%, so I decided to use it for my final set of features.

For item 8, I have calculated count of words whose length is more than some number n. Through extensive search, I found that the classifiers works best with n = 8, which gives an

accuracy of around 46%. Decreasing or increasing n decreases accuracy. I have used this feature, `large_words` for my final feature set.

For my final feature set, I have tried various combinations of the discussed features. I tried to use all features first and then tried removing a feature. If removing the feature didn't reduce accuracy, then I didn't include that feature. Finally, after extracting all features, I normalize all columns with min-max normalization. For my final feature set, I get an accuracy of around 85% for all three authors. I have 2000 rows for each author and train-test split of the dataset is done using `sklearn.model_selection.train_test_split` - this is the only instance where I use sklearn library, it's not used anywhere else.

One thing to note is that the mentioned accuracies are for decision trees only. Logistic Regression classifier perfomed slightly better than Decision Tree classifier (once normalized).

3 Decision tree classifier

The code for decision tree classifier is divided into four classes, `Question`, `Leaf`, `Node`, and `Decision_Tree`. `Question` is used to store a split value for a particular column, and is also used for pretty-printing the decision tree. `Leaf` represents the leaves of the classifier and contains class counts and prediction of the leaf. `Node` represents every other decision node of the classifier; it stores a `Question` object and left and right split of the feature array. `Decision_Tree` class ties everything together and contains the user APIs of `fit`, `predict` and `score` - I have tried to keep the APIs similar to sklearn, for easier understanding.

When features and target are passed to the fit method of the classifier, first, it finds all possible split points for all columns. This is done by getting sorted unique values for all columns and finding mean for consecutive pairs of values. Then, information gain is calculated for all these split points, and for the maximum gain, left and right splits are used to recursively build the tree. The whole process takes around 10 seconds to train the decision tree, so I didn't convert the feature set into boolean features. I also contemplated taking random sample of around 10% of the dataset to calculate gain in order to reduce training time, but since the time taken is not too long, I didn't include it in my final submission.

The decision tree has a depth cutoff limit, which defaults to 5, but can be changed from the command line with a `--max_depth`. With depth cutoff of 5, the accuracy is around 85%, which reduces to around 80% if a cutoff of 100 is passed. There is also a gradual decline in accuracy for depth of 10 and 20, to 82%. Reducing the depth also decreases accuracy, with 82% for depth of 3 and 2, and 60% for a stump (depth = 1).

I will include a separate README for how to run everything, but, these results can be obtained by running,

```
$ python3 lab2.py --train_test
```

which extracts features, uses `sklearn.model_selection.train_test_split` to split the training and testing set, fits both classifiers and prints overall and class accuracies (in the order of Arthur Conan Doyle, Herman Melville, Jane Austen).

4 Logistic Regression Classifier

Implementation of Logistic Regression classifier was much simpler than the Decision Tree. `LogisticMulti` class implements the whole classifier using one vs all strategy. I have used log likelihood as my error function for gradient descent, with learning rate = 0.001 and 20000 iterations. I have used this link for the formulae.

Since there are three authors, I have used one vs all strategy to train the classifier. First, the target column is copied and then transformed with only binary values (0s and 1s). This is done for each author, with 1s for positive samples and 0s for negative samples for the author. Then,

the classifier is trained with gradient descent. The weights learned are stored in a dictionary where keys are authors. For testing a row, I simply use the dictionary to get weights and calculate the sigmoid activation of the row for each author. The prediction is the author whose weights produce the maximum activation value.

I have tried training the classifier using different combinations of features. The accuracies mentioned in the section 2 are for Decision Tree but those values are true for Logistic Regression classifier too, although, this classifier almost always performs slightly better, by 2-3 points. Using all selected features gives an accuracy of around 87%, whereas, it is around 85% for decision tree. Interestingly, removing the most important feature (`count_punc` - decided by the root node of DT classifier) reduces accuracy of LR classifier to around 70% and DT classifier to 67%. Other combinations, which includes `count_punc`, give accuracies between 70% and 87%.