STA 629 - Homework 4

Anthony Bernardi

November 12th, 2024

1 Problem 1

1.1 Question and Initial Code Work

For this problem, we are given the authorship dataset, which contains 69 columns of the most commonly used words for a given work, how often each of these words are used, and then finally the author of the work.

We will compare four different model classes in an attempt to predict the author based on this distribution of word usage. We will then compare and contrast the models, and provide some insights on the results.

The four model classes we will be comparing are:

- Classification Trees
- Bagging
- Boosting
- Random Forests

We will then try to answer a series of sub-questions, which we will get to after including the model code, evaluating, contrasting, and comparing.

We will start with the Classification Tree, which can be thought of as a baseline.

1.2 Classification Trees

To start, we use the following code to load and organize the data. This includes everything from importing the data, splitting via training and testing split, and fitting our Classification Tree Model.

```
import pandas as pd
import numpy as np


# we'll start by importing the data
df_train = pd.read_csv(
'/home/adbucks/Documents/sta_629/Homework_4/authorship_training.csv')

df_test = pd.read_csv(
'/home/adbucks/Documents/sta_629/Homework_4/authorship_testing.csv')

# now we can take a peek

print(df_train.head(10))
print(df_test.head(10))
# looks right

# here, we have 69 columns of words and their counts
# and the final count is the author name
# we are going to build models to predict author name based on
# the word count for the 69 most common words

####################################################
# in general, we'll use the following methods
# 1. Classification Tree
# 2. Bagging
# 3. Boosting
# 4. Random Forests
####################################################

# we'll start by building a CART, and this can be a baseline for the other methods
# given this is a multi-class classification problem with the authors, we
# will import the tree method accordingly
```

```
from sklearn import tree

# further preparing the data
X_train = df_train.iloc[:, :69]
y_train = df_train.iloc[:, 69]

# testing
#print(X_train.head(10))
#print(y_train.head(10))

# looks good so we can keep going
# creating the CART instance
cart = tree.DecisionTreeClassifier()

# fitting
cart.fit(X_train, y_train)

# now predicting and evaluating
# partitioning first
X_test = df_test.iloc[:, :69]
y_test = df_test.iloc[:, 69]

# predicting
y_pred = cart.predict(X_test)
```

This allows us to get our initial Classification Tree trained. Now, we can go through how we plot the initial tree and introduce metrics via SciKit Learn to evaluate our initial model.

```
import matplotlib.pyplot as plt
# we can even plot
tree.plot_tree(cart)
plt.show()

# a few different scoring metrics
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix

# now we can compare
```

```
print('CART Accuracy: ', accuracy_score(y_test, y_pred))
print('CART F1 Score: ', f1_score(y_test, y_pred, average = 'weighted'))
print('CART Confusion Matrix: ', confusion_matrix(y_test, y_pred))
```

Further investigating the data tells us that we are interested in the authors Austen, Shakespeare, London, and Milton.

For our evaluation, we will use multiple metrics ranging in interpretability, but I would favor the use of the F1 Score in this case, as this is a harmonic mean between Precision and Recall.

We will want to note that we have an 87% accuracy rate, which is quite encouraging for a baseline. Additionally, the interpretability of Classification Trees always make them an attractive choice for certain audiences.

With that in mind, we will move on to Bagging.

1.3 Bagging

Bagging, or bootstrap aggregating, sacrifices some interpretability for hopefully improved accuracy. We will see if that is the case now, with the code below responsible for fitting and evaluating the model.

We also spend some time evaluating and thinking about OOB, or Out of Bag Error, which relates to the bootstrapping phenomenon where around 33% of observations, over time, are not included in a bootstrap.

We'll look at the code now.

```
from sklearn.ensemble import BaggingClassifier

# creating the instance now
bag = BaggingClassifier() # using mostly defaults for starters

bag.fit(X_train, y_train)

# we'll do predicting now , and then scoring, and we can even try to do
# parameter extraction, to see if we can drive some inference
y_pred_bag = bag.predict(X_test)

# now can do our scoring as before
print('Accuracy: ', accuracy_score(y_test, y_pred_bag))
print('F1 Score: ', f1_score(y_test, y_pred_bag, average = 'weighted'))
print('Confusion Matrix: ', confusion_matrix(y_test, y_pred_bag))

# we see the slight jump in accuracy, giving a 92% f1 score
# we can see the benefit!
# let's try and extract the parameters now

# we can try using out of bag score and see if that makes much difference
bag_oob = BaggingClassifier(oob_score = True)
bag_oob.fit(X_train, y_train)

# now we can extract the oob score
print('OOB Score: ', bag_oob.oob_score_)
```

This code gives the noticeable increase in performance, with a 92% F1 Score. It seems as if what we have given up in interpreability might be a worthwhile trade.

We ought to also mention that our Out of Bag Score is 95%, which is quite high. This is a good sign that our model is generalizing well.

We'll now move on to Boosting.

1.4 Boosting

In this section we'll compare the performance of the AdaBoost adn Gradient Boosted models, we'll talk about our eventual decision, and compre the two models.

Also, we will use the concept of a validation set to compare different candidate models, so that we can keep our testing data "in a vault" until we have decided on a model choice. We'll include the code for generating the validation set and initializing the AdaBoost model below.

```
from sklearn.ensemble import AdaBoostClassifier

# can try and create a validation set just to
# keep the test set in a vault
# here, we'll just use the normal train test split again
from sklearn.model_selection import train_test_split

# and can apply this to our existing training data
```

```
X_train2, X_val, y_train2, y_val = train_test_split(X_train, y_train, test_size =

# now we can train and test on the validation set
# testing
print(X_val.head(10))
print(X_val.shape)
print(X_train2.shape)

# can create the instance now
ada_default = AdaBoostClassifier()

# fitting on the new training set
ada_default.fit(X_train2, y_train2)

# evaluating on the validation set
y_pred_val = ada_default.predict(X_val)

# now we can score for a baseline before cross-validation
print('Adaboost Accuracy: ', accuracy_score(y_val, y_pred_val))
print('Adaboost F1 Score: ', f1_score(y_val, y_pred_val, average = 'weighted'))
print('Adaboost Confusion Matrix: ', confusion_matrix(y_val, y_pred_val))

y_pred_adatest = ada_default.predict(X_test)

# scoring
print('Adaboost Test Accuracy: ', accuracy_score(y_test, y_pred_adatest))
print('Adaboost Test F1 Score: ', f1_score(y_test, y_pred_adatest, average = 'weig
print('Adaboost Test Confusion Matrix: ', confusion_matrix(y_test, y_pred_adatest)
```

We see that we have a validation test F1 score of 0.83, and a test F1 score of 0.81. It is interesting that the validation error proves optimistic still, despite the reduction in data.

It is also noteworthy that in this case, the default or baseline Adaboost performs the worst out of any model on this data.

Now that we have this as a point of measurement, we will move on to Gradient Boosting.

Below is the code for implementing the Gradient Boosting model, along with training and testing on the newly created validation set.

```
from sklearn.ensemble import GradientBoostingClassifier

# can just stick with the default values for comparison's sake
# can do this with the validation set as the test set as well
grad_def_val = GradientBoostingClassifier()

# fitting
grad_def_val.fit(X_train2, y_train2)

# predicting
y_pred_grad_val = grad_def_val.predict(X_val)

# now accuracy and scoring
print('Gradient Boosting Accuracy: ', accuracy_score(y_val, y_pred_grad_val))
print('Gradient Boosting F1 Score: ', f1_score(y_val, y_pred_grad_val, average = '
print('Gradient Boosting Confusion Matrix: ', confusion_matrix(y_val, y_pred_grad_

# an insane 98% accuracy with the gradient boosting...

# now we can use the test data with gradient boost given this is the method
# we are going with

y_pred_grad = grad_def_val.predict(X_test)

# now accuracy and scoring
print('Gradient Boosting Test Accuracy: ', accuracy_score(y_test, y_pred_grad))
print('Gradient Boosting Test F1 Score: ', f1_score(y_test, y_pred_grad, average =
print('Gradient Boosting Test Confusion Matrix: ', confusion_matrix(y_test, y_pred
```

By some distance our best performing model to this point, with a 98% accuracy rate on the validation set.

We continue with testing the validation set, where we observe a similarly high 97% accuracy.

For our preferred metric, F1 score, we observe a 98% rate on the validation set and a 97% score on the test set.

I'm now seeing the popularity of Gradient Boosted trees!

Finally, we'll move on to Random Forests before our discussion for Question 1.

1.5 Random Forests

We will now implement Random Forests and see if they can compare to our Gradient Boosted models in terms of performance.

The implementation code is below.

```
from sklearn.ensemble import RandomForestClassifier

# we might want to start with max depth, but we can also try the default
# we will probably have to prune after getting the default tree

rf_base = RandomForestClassifier(random_state = 42)

# fitting
rf_base.fit(X_train2, y_train2) # still using the validation set

# RF allows us to see a bit more, so we can print out a few things of
# interest here
# now predicting on the validation set
y_pred_rf_val = rf_base.predict(X_val)

# now accuracy and scoring, feature importance, and decision path
print('Base RF Decision Path: ', rf_base.decision_path(X_val))
print('Base RF Feature Importance: ', rf_base.feature_importances_)
print('Base RF Accuracy: ', accuracy_score(y_val, y_pred_rf_val))
print('Base RF F1 Score: ', f1_score(y_val, y_pred_rf_val, average = 'weighted'))
print('Base RF Confusion Matrix: ', confusion_matrix(y_val, y_pred_rf_val))

# let's now try on our testing data, as we might have to purne a second
# tree to avoid overfitting
y_pred_rf = rf_base.predict(X_test)

# now accuracy and scoring
print('Base RF Test Accuracy: ', accuracy_score(y_test, y_pred_rf))
print('Base RF Test F1 Score: ', f1_score(y_test, y_pred_rf, average = 'weighted')
print('Base RF Test Confusion Matrix: ', confusion_matrix(y_test, y_pred_rf))

# 98% test accuracy, which is encouraging
# finally, we can try pruning a bit and see how that goes
# we can try a naive max depth of 5, although this is something that
# mght be best for grid search
```

Before evaluating the model, we want to talk about model pruning. For the intial step, we set the *max depth* argument to the default value of None.

For comparison's sake, we will set the *max depth* argument to 5, and compare this to the default case of None.

In evaluating our default Random Forest model, we note a 99% test F1 score, comparable to the Gradient Boosted Trees, but slightly better.

For comparison, we will now use the *max depth* argument equal to 5, and see if we can attain a similar result while saving on computational complexity, or compute cost.

The code for implementing our second Random Forest model is below.

```
rf_p5 = RandomForestClassifier(max_depth = 5, random_state = 42) # naive

# fitting on the pre-validation training set
rf_p5.fit(X_train2, y_train2)

# predicting on the validation set
y_pred_rf_p5_val = rf_p5.predict(X_val)

# accuracy and scoring
print('Pruned RF Accuracy: ', accuracy_score(y_val, y_pred_rf_p5_val))
print('Pruned RF F1 Score: ', f1_score(y_val, y_pred_rf_p5_val, average = 'weighte
print('Pruned RF Confusion Matrix: ', confusion_matrix(y_val, y_pred_rf_p5_val))

# purned accuracy increases to 99%, now we can try to the test data
y_pred_test_rfp5 = rf_p5.predict(X_test)

# accuracy and scoring
print('Pruned RF Test Accuracy: ', accuracy_score(y_test, y_pred_test_rfp5))
print('Pruned RF Test F1 Score: ', f1_score(y_test, y_pred_test_rfp5, average = 'w
print('Pruned RF Test Confusion Matrix: ', confusion_matrix(y_test, y_pred_test_rf
```

We note that the pruned Random Forest still provides an F1 score of

98%, which suggests we can cut back on compute, and reduce the risk of overfitting, while achieving comparable results.

That is, even the pruned Random Forest model is tied for the best model performance with Gradient Boosted trees.

1.6 Discussion

Reflect upon your results. Which method yields the best error rate? Which method yields the most inrepretable results? Which words are most important for authorship attribution?

As far as error rate, when we use the F1 score, we see that the Random Forest model is tied with the Gradient Boosted trees method, with a test F1 score of 98%.

It is entirely possible that this data set might not provide the best opportunity for a model that generalizes, but a 98% test F1 score is exciting regardless.

In a follow-up study, it would be worth finding a similar multi-class data set to see if our Random Forest and Gradient Boosted models can generalize.

In terms of interpretability, there are measures we can take with Random Forests and other more "Algorithmic" approaches to increase their interpretability and explainability, but in a pure sense, a Classification Tree is preferable.

If explaining and contextualizing this problem to a non-technical audience, a Classification Tree with an impressive baseline F1 Score would be a productive option, despite not performing as well as other methods.

We'll now move on to our final question, that is, which words are most important for predicting authorship.

We will include the code that helps us generate a DataFrame with the most importance word features, for the base Random Forest model, and the pruned Random Forest model afterwards.

```
feature_names = df_train.columns[:69]

print('Feature Names: ', feature_names)

# now we have this, we can get feature importance for our two
# random forest models
importance_base = rf_base.feature_importances_
```

```
base_feat_df = pd.DataFrame({'Feature': feature_names,
                             'Importance': importance_base}).sort_values('Importar

print(base_feat_df)

# we can do this same thing with the pruned random forest and see if we get
# the same thing
importance_p5 = rf_p5.feature_importances_

prune_feat_df = pd.DataFrame({'Feature': feature_names,
                              'Importance': importance_p5}).sort_values('Importanc

print(prune_feat_df)
```

We'll include a table of the top 5 features below, with their importance score before continuing.

We'll start with the base Random Forest model.

Feature	Importance
was	0.10
be	0.08
the	0.08
to	0.07
her	0.06

And now for our pruned Random Forest model.

Feature	Importance
was	0.10
be	0.08
the	0.08
her	0.07
to	0.06

The generality of the words is somewhat surprising, I would imagine authors would have their own preferences or "turns of phrase", but I think the fact that no single feature scores higher than 0.10 bears mentioning.

This is a good sign that the model is generalizing well, and that the authors are not using words that are too unique to them.

2 Problem 2

Derive the expression for the updating parameter in the Adaboost algorithm.

We will start by outlining expression 10.12, the updating parameter.

$$\beta_m = \frac{1}{2} \log \left(\frac{1-\epsilon_m}{\epsilon_m} \right)$$

Given the context of AdaBoost, we will start with the exponential loss function, which we can write in the following way.

$$(\beta_m, G_m) = \text{argmin}_{\beta,G} \sum_{i=1}^{N} w_i^{(m)} \cdot \exp(-\beta y_i G(x_i)) \tag{1}$$

We can take this and plug in to the following equation to start thr process. As given in the text, we can re-write the above equation in the following way.

Which then gives us the following.

$$\sum_{y_i=G(x_i)} w_i^{(m)} \exp(-\beta) - \sum_{y_i \neq G(x_i)} w_i^{(m)} \exp(\beta) = 0 \tag{2}$$

Algebra and some manipulation gives us the following.

$$exp(2\beta) = \frac{\sum_{y_i=G(x_i)} w_i^{(m)}}{\sum_{y_i \neq G(x_i)} w_i^{(m)}} \tag{3}$$

We can write this in another way, giving us a more familiar form.

$$exp(2\beta)\frac{1-\epsilon_m}{\epsilon_m} \tag{4}$$

We can think about this term as the minimized weighted error rate, given below.

$$\epsilon_m = \frac{\sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} \qquad w_i^{(m)}} \tag{5}$$

Therefore, we get our original equation from the beginning of the question, given again here.

$$\beta = \frac{1}{2} log \frac{1-\epsilon_m}{\epsilon_m} \tag{6}$$