Authorship of the Federalist Papers

February 27, 2019

1 Authorship of the Federalist Papers

The *Federalist Papers* were a set of 85 essays published between 1787 and 1788 to promote the ratification of the United States Constitution. They were originally published under the pseudonym "Publius". Although the identity of the authors was a closely guarded secret at the time, most of the papers have since been conclusively attributed to one of Hamilton, Jay, or Madison. The known authorships can be found in `/data301/data/federalist/authorship.csv`.

For 15 of the papers, however, the authorships remain disputed. (These papers can be identified from the `authorship.csv` file because the "Author" field is blank.) In this analysis, you will train a classifier on the papers with known authorships and use your classifier to predict the authorships of the disputed papers. The text of each paper can be found in the `/data301/data/federalist/` directory. The name of the file indicates the number of the paper.

In [1]:
```
%matplotlib inline
import numpy as np
import pandas as pd
from collections import Counter

authorship = pd.read_csv("/data301/data/federalist/authorship.csv")
authorship.head()
```

Out[1]:

	Paper	Author
0	1	Hamilton
1	2	Jay
2	3	Jay
3	4	Jay
4	5	Jay

1.1 Question 1

When analyzing an author's style, common words like "the" and "on" are actually more useful than rare words like "hostilities". That is because rare words typically signify context. Context is useful if you are trying to find documents about similar topics, but not so useful if you are trying to identify an author's style because different authors can write about the same topic. For example, both Dr. Seuss and Charles Dickens used rare words like "chimney" and "stockings" in *How the Grinch Stole Christmas* and *A Christmas Carol*, respectively. But they used common words very differently: Dickens used the word "upon" over 100 times, while Dr. Seuss did not use "upon" at all.

Read in the Federalist Papers. Convert each one into a vector of term frequencies. In order to restrict to common words, include only the top 50 words. Then, train a k-nearest neighbors model on the documents with known authorship. Determine an optimal value of k (it's up to you to decide what's "optimal").

Report an estimate of the test accuracy, precision, and recall of your model.

```
In [2]: complete_df = pd.DataFrame()

        for i in authorship.Paper:
            text_name = "/data301/data/federalist/{0}.txt".format(i)
            paper = pd.read_csv(text_name, sep="\t")

            bag_of_words = (
            paper['To the People of the State of New York:'].
            str.lower().
            str.replace("[^A-Za-z\s]", "").
            str.split()
            ).apply(Counter)

            tf = pd.DataFrame(list(bag_of_words))
            tf = tf.fillna(0)

            word_counts = tf.sum(axis=0).sort_values(ascending=False)
            top50 = word_counts[0:50]
            top50_df = pd.DataFrame(top50)
            top50_df = top50_df.T
            top50_df["my_index"] = i - 1
            complete_df = complete_df.append(top50_df)

        complete_df = complete_df.set_index("my_index")
        complete_df = complete_df.fillna(0)
        complete_df["the_author"] = authorship.Author
```

```
/opt/conda/lib/python3.6/site-packages/pandas/core/frame.py:6211: FutureWarning: Sorting becaus
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=True'.

  sort=sort)
```

```
In [3]: top50_sum = complete_df.sum(axis=0).sort_values(ascending=False)[0:50]

In [4]: df = pd.DataFrame(top50_sum).T

In [5]: complete_df = complete_df[df.columns]
```

In [6]:
```
complete_df["the_author"] = authorship.Author
```

In [7]:
```
complete_df.head()
```

Out[7]:

my_index	the	of	to	and	in	a	be	that	it	is	\
0	130.0	104.0	71.0	40.0	27.0	25.0	34.0	28.0	20.0	13.0	
1	105.0	81.0	52.0	83.0	34.0	29.0	15.0	44.0	38.0	16.0	
2	91.0	60.0	55.0	60.0	25.0	13.0	31.0	20.0	21.0	7.0	
3	84.0	70.0	50.0	90.0	24.0	16.0	26.0	17.0	28.0	10.0	
4	64.0	51.0	44.0	72.0	28.0	9.0	31.0	23.0	21.0	7.0	

my_index	...	people	them	one	any	if	there	can	constitution	\
0	...	0.0	0.0	0.0	6.0	0.0	0.0	0.0	8.0	
1	...	21.0	0.0	10.0	0.0	0.0	0.0	0.0	0.0	
2	...	7.0	8.0	8.0	5.0	7.0	0.0	0.0	0.0	
3	...	7.0	12.0	13.0	0.0	13.0	0.0	8.0	0.0	
4	...	0.0	11.0	10.0	0.0	0.0	0.0	0.0	0.0	

my_index	upon	the_author
0	6.0	Hamilton
1	0.0	Jay
2	0.0	Jay
3	0.0	Jay
4	0.0	Jay

[5 rows x 51 columns]

In [8]:
```
known_authors_df = complete_df[(complete_df.the_author == "Hamilton") |
                               (complete_df.the_author == "Jay") |
                               (complete_df.the_author == "Madison")]

unknown_authors_df = complete_df[(complete_df.the_author != "Hamilton") &
                                 (complete_df.the_author != "Jay") &
                                 (complete_df.the_author != "Madison")]
```

In [9]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

X_train = known_authors_df.iloc[:, 0:50]
y_train = known_authors_df.the_author

scaler = StandardScaler()
scaler.fit(X_train)
X_train_sc = scaler.transform(X_train)
```

```
is_hamilton_train = (y_train == "Hamilton")

best_k = []
best_cv = []
best_accuracy = []
best_recall = []
best_precision = []
best_score = 0


for k in [1,2,3,4,5,6,7,8,9,10]:
    model = KNeighborsClassifier(n_neighbors=k)
    model.fit(X_train_sc, y_train)

    pipeline = Pipeline([
        ("scaler", scaler),
        ("model", model)
    ])

    for cross in [2,3,4,5]:
        f1 = cross_val_score(pipeline, X_train, is_hamilton_train,
                             cv=cross, scoring="f1").mean()

        accuracy = cross_val_score(pipeline, X_train, y_train,
                                   cv=cross, scoring="accuracy").mean()

        recall = cross_val_score(pipeline, X_train, is_hamilton_train,
                                 cv=cross, scoring="recall").mean()

        precision = cross_val_score(pipeline, X_train, is_hamilton_train,
                                    cv=cross, scoring="precision").mean()

        if f1 > best_score:
            best_score = f1
            best_k.append(k)
            best_cv.append(cross)
            best_accuracy.append(accuracy)
            best_recall.append(recall)
            best_precision.append(precision)

print("Neighbors: ", best_k.pop())
print("Cross: ", best_cv.pop())
print("F1: ", best_score)
print("Accuracy: ", best_accuracy.pop())
print("Recall: ", best_recall.pop())
print("Precision: ", best_precision.pop())
```

```
Neighbors:  1
Cross:  3
F1:  0.89022556391
Accuracy:  0.839646464646
Recall:  0.941176470588
Precision:  0.846560846561
```

I found that the optimal value of k to be 1. Using 1 neighbor and testing my model with 3 folds yielded an F1 score of .8902, an accuracy score of .8396, a recall of .9411, and a precision score of .8465, when testing my predictions for Hamilton.

To obtain these answers, I first created an empty data frame to which I appended the term frequencies from the bag of words vector for each of the Federalist Papers. I then split this dataframe into two subsets - one for known authors and the other for the unknown authors. To find the ideal k, I set the training data to all columns of the known authors data frame except for the author column. I then iterated over different values of k and different values of folds. I used the f1 score to compare different the models because the f1 score is a combination of precision and recall, and accuracy alone is not always the best statistic to use. To compare each iteration, I set the "best F1 score" equal to 0 and if the new F1 score was greater than the previous, I appended the number of crosses, accuracy, recall, and precision scores to respective lists and then popped the last value from each of the lists knowing it would correlate with the highest F1 score.

I chose to use a range of 1-10 for k because it seemed more appropriate to use a smaller number of neighbors in comparison to 30 since the data set is only 70 observations. I also only chose to use up to 5 folds because the data set is small and there needs to be enough observations in each fold to create a better model.

1.2 Question 2

What if we used TF-IDF on the top 50 words instead of the term frequencies? Repeat Question 1, using TF-IDF instead of TF. Which approach is better: TF-IDF or TF?

```
In [10]: tf = complete_df.iloc[: , 0:50]
         df = (tf > 0).sum(axis=0)
         idf = np.log(len(tf) / df)
         tf_idf = tf * idf
         tf_idf["the_author"] = authorship.Author

In [11]: known_authors_tf_idf = tf_idf[(tf_idf.the_author == "Hamilton") |
                                       (tf_idf.the_author == "Jay") |
                                       (tf_idf.the_author == "Madison")]

         unknown_authors_tf_idf = tf_idf[(tf_idf.the_author != "Hamilton") &
                                         (tf_idf.the_author != "Jay") &
                                         (tf_idf.the_author != "Madison")]

In [12]: X_train_idf = known_authors_tf_idf.iloc[: , 0:50]
         y_train_idf = known_authors_tf_idf.the_author

         scaler_idf = StandardScaler()
```

```
scaler_idf.fit(X_train_idf)
X_train_sc_idf = scaler_idf.transform(X_train_idf)

is_hamilton_train_idf = (y_train_idf == "Hamilton")

model_idf = KNeighborsClassifier(n_neighbors=1)
model_idf.fit(X_train_sc_idf, y_train_idf)

pipeline_idf = Pipeline([
    ("scaler", scaler_idf),
    ("model", model_idf)
])

f1_idf = cross_val_score(pipeline_idf, X_train_idf,
                         is_hamilton_train_idf, cv=3,
                         scoring="f1").mean()

accuracy_idf = cross_val_score(pipeline_idf, X_train_idf,
                               y_train_idf, cv=3,
                               scoring="accuracy").mean()

recall_idf = cross_val_score(pipeline_idf, X_train_idf,
                             is_hamilton_train_idf, cv=3,
                             scoring="recall").mean()

precision_idf = cross_val_score(pipeline_idf, X_train_idf,
                                is_hamilton_train_idf, cv=3,
                                scoring="precision").mean()


print("Neighbors: 1")
print("Cross: 3")
print("F1: ", f1_idf)
print("Accuracy: ", accuracy_idf)
print("Recall: ", recall_idf)
print("Precision: ", precision_idf)
```

```
Neighbors: 1
Cross: 3
F1:  0.916172784903
Accuracy:  0.839646464646
Recall:  0.960784313725
Precision:  0.878773731715
```

Using TF-IDF produces different results than when using just TF on the same model. The F1 score, recall, and precision increase, while the accuracy score stays the same from before. This got me thinking if the TF model form before is the best model for TF-IDF, so in the next cell, I find the "most ideal" model for TF-IDF.

I obtained this answer by first creating the IDF dataframe. The dimensions of the dataframe are the same as before and the process to obtain the f1 score, accuracy, recall, and precision was all the same. I simply changed the training set data to the IDF's of known authors. Since, I was using a set number of neighbors and folds, I did not have iterate over variations of neighbors and crosses.

```
In [13]: X_train_idf = known_authors_tf_idf.iloc[: , 0:50]
         y_train_idf = known_authors_tf_idf.the_author

         scaler_idf = StandardScaler()
         scaler_idf.fit(X_train_idf)
         X_train_sc_idf = scaler_idf.transform(X_train_idf)

         is_hamilton_train_idf = (y_train_idf == "Hamilton")

         best_k_idf = []
         best_cv_idf = []
         best_accuracy_idf = []
         best_recall_idf = []
         best_precision_idf = []
         best_score_idf = 0

         for k in [1,2,3,4,5,6,7,8,9,10]:
             model_idf = KNeighborsClassifier(n_neighbors=k)
             model_idf.fit(X_train_sc_idf, y_train_idf)

             pipeline_idf = Pipeline([
                 ("scaler", scaler_idf),
                 ("model", model_idf)
             ])

             for cross in [2,3,4,5]:
                 f1_idf = cross_val_score(pipeline_idf, X_train_idf,
                                          is_hamilton_train_idf,
                                          cv=cross, scoring="f1").mean()

                 accuracy_idf = cross_val_score(pipeline_idf, X_train_idf,
                                                y_train_idf, cv=cross,
                                                scoring="accuracy").mean()

                 recall_idf = cross_val_score(pipeline_idf, X_train_idf,
                                              is_hamilton_train_idf, cv=cross,
                                              scoring="recall").mean()

                 precision_idf = cross_val_score(pipeline_idf, X_train_idf,
                                                 is_hamilton_train_idf, cv=cross,
                                                 scoring="precision").mean()
```

```
                if f1_idf > best_score_idf:
                    best_score_idf = f1_idf
                    best_k_idf.append(k)
                    best_cv_idf.append(cross)
                    best_accuracy_idf.append(accuracy_idf)
                    best_recall_idf.append(recall_idf)
                    best_precision_idf.append(precision_idf)

        print("Neighbors: ", best_k_idf.pop())
        print("Cross: ", best_cv_idf.pop())
        print("F1: ", best_score_idf)
        print("Accuracy: ", best_accuracy_idf.pop())
        print("Recall: ", best_recall_idf.pop())
        print("Precision: ", best_precision_idf.pop())
```

```
Neighbors:  2
Cross:  5
F1:  0.922198830409
Accuracy:  0.729084249084
Recall:  0.92
Precision:  0.934920634921
```

It turns out that for TF-IDF, the "best model" is a model using 2 neighbors and 5 folds. I received a F1 score of .9221 which is greater than the F1 score from the TF model. The precision of this TF-IDF model has also increased from before, however, the accuracy decreased over 10% and the recall decreased 4% as well. It seems that this "best model" is not as good as from before, but that is because I am defining the "best" to be the F1 score, so I will proceed to use the IDF model with 2 neighbors.

1.3 Question 3

Using the model that you determined to be best in Questions 1 and 2, fit a *k*-nearest neighbors model to all 70 documents with known authorship. Create a confusion matrix for your model that shows how often you predicted Hamilton, Jay, or Madison, and how often it actually was Hamilton, Jay, or Madison (on the training data, of course).

From your confusion matrix, you should be able to calculate the (training) precision and recall of your model for predicting Hamilton. What is it?

```
In [14]: model_idf = KNeighborsClassifier(n_neighbors=2)
         model_idf.fit(X_train_sc_idf, y_train_idf)

         y_train_pred = model_idf.predict(X_train_sc_idf)
         y_train_pred
```

```
Out[14]: array(['Hamilton', 'Hamilton', 'Jay', 'Jay', 'Hamilton', 'Hamilton',
                'Hamilton', 'Hamilton', 'Hamilton', 'Madison', 'Hamilton',
                'Hamilton', 'Hamilton', 'Madison', 'Hamilton', 'Hamilton',
                'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton',
```

```
                'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton',
                'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton',
                'Hamilton', 'Hamilton', 'Hamilton', 'Madison', 'Madison',
                'Hamilton', 'Madison', 'Madison', 'Madison', 'Hamilton', 'Jay',
                'Madison', 'Madison', 'Hamilton', 'Hamilton', 'Hamilton',
                'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton',
                'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton',
                'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton',
                'Hamilton', 'Hamilton', 'Hamilton'], dtype=object)
```

```
In [15]: from sklearn.metrics import confusion_matrix
```

```
In [16]: y_pred = list(y_train_pred)
         y_true = list(y_train_idf)
         conf = confusion_matrix(y_pred, y_true,
                                 labels=["Hamilton", "Jay", "Madison"])
         conf
```

```
Out[16]: array([[51,  3,  4],
                [ 0,  2,  1],
                [ 0,  0,  9]])
```

```
In [17]: authorship.Author.value_counts()
```

```
Out[17]: Hamilton    51
         Madison     14
         Jay          5
         Name: Author, dtype: int64
```

```
In [18]: accuracy = 59/70
         print("Accuracy: ", accuracy)

         precision = 51/(51+3+4)
         print("Precision: ", precision)

         recall = 51/51
         print("Recall: ", recall)
```

```
Accuracy:  0.8428571428571429
Precision:  0.8793103448275862
Recall:  1.0
```

Using the confusion matrix, I predicted for Hamilton. I found that the accuracy of my model to be .8428. The precision of my model is .8793 and the recall of my model is 1.0.

I calculated the accuracy by summing the True Positives and True Negatives and dividing the sum by the total observations.

I calculated the precision by dividing the number of True Positives by the sum of True Positives and False Negatives (how many times my model predicted the author to be Hamilton).

I calculated the recall by dividing the number of True Positives by the sum of True Positives and False Negatives.

1.4 Question 4

Finally, use the model you trained in Question 3 to predict the authorships of the 15 documents with unknown authors. Summarize what you find.

```
In [19]: X_val = unknown_authors_tf_idf.iloc[: , 0:50]
         scaler_idf.fit(X_val)
         X_val_sc = scaler_idf.transform(X_val)

         y_val_pred = model_idf.predict(X_val_sc)
         y_val_pred

Out[19]: array(['Hamilton', 'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton',
                'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton', 'Hamilton',
                'Hamilton', 'Hamilton', 'Hamilton', 'Madison', 'Hamilton'], dtype=object)
```

According to my TF-IDF model using 2 neighbors, I am predicting that the unknown author of the 14 Federalist Papers is Hamilton and that Madison wrote 1 of the Federalist Papers.

I obtained this answer by setting the X_validation data to the unknown authors TF-IDF dataframe, scaling the data, and then transforming it. I then used the IDF model with 2 neighbors to predict the author.

2 Submission Instructions

Once you are finished, follow these steps:

1. Restart the kernel and re-run this notebook from beginning to end by going to `Kernel > Restart Kernel and Run All Cells`.
2. If this process stops halfway through, that means there was an error. Correct the error and repeat Step 1 until the notebook runs from beginning to end.
3. Double check that there is a number next to each code cell and that these numbers are in order.

Then, submit your lab as follows:

1. Go to `File > Export Notebook As > PDF`.
2. Double check that the entire notebook, from beginning to end, is in this PDF file. (If the notebook is cut off, try first exporting the notebook to HTML and printing to PDF.)
3. Upload the PDF to PolyLearn.