Assignment 4 - Classification Models

Matthew Dunne

Create classification model, predicting the outcome of food safety inspection based on the inspectors' comments.

The Data

Leverage the results of your homework from Week-1 and Week-2 to extract free-form text comments from inspectors. Discard the text from "Health Code" – only keep inspectors' comments.

I also subsetted on rows where the 'Results' column contained the values: Pass, Pass w/ Conditions, or Fail. Even after dropping NAs there are restaurants with values in that column of: Not Ready, No Entry, or Out of Business. These must be taken out.

In [1]:

```
import pandas as pd
import nltk as nltk
import nltk.corpus
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
#use only the columns you need: whether the inspection failed and the specific violations
df = pd.read_csv('Food_Inspections.csv', usecols=['Results', 'Violations'])
#drop any rows with NA/NaN in either column
df=df.dropna()
df=df[(df['Results']=='Pass w/ Conditions') | (df['Results']=='Pass') | (df['Results']=='Fail')]
print(df.describe())
print(len(df))
```

```
       Results                                          Violations
count   128059                                              128059
unique       3                                              127363
top       Pass  30. FOOD IN ORIGINAL CONTAINER, PROPERLY LABEL...
freq     79156                                                  10
128059
```

Here I cycled through each cell in the Violations column, extracted all of the inspector comments, and then re-joined them into one long string. This resulting string is made the new value in the 'Violations' column.

In [2]:

```
import re
```

In [3]:

```
for i in range(len(df)):
    #number of separate violations in each restaurant, +1 because you're counting violations, pipe just separates.
    separate_violations=df.iloc[i]['Violations'].count('|')+1
    #for each restaurant go into the 'Violations' column and split on the pipe
    offenses=df.iloc[i]['Violations'].split('|', maxsplit=separate_violations)
    #for each restaurant's list of violations, discard the the text from "Health Code" - only keep inspectors' comments.
    comments=[]
    for j in range(len(offenses)):
        #some cells have comments and some don't. Skip the ones that don't
        if re.search('Comments: ', offenses[j]):
            comment=offenses[j].split(' - Comments: ')[1]
            comments.append(comment)
        else:
            continue
        df.iloc[i]['Violations']=' '.join(comments)
```

Here is an example of a typical new value of the extracted comments.

In [4]:

```
example=df['Violations'][1000]
print(example)
```

```
4-301.12 (B):  MUST PROVIDE A STEP BY STEP, WRITTEN, DETAILED, CLEANING PROCEDURE FOR EACH
SEPARATE PIECE OF FOOD-CONTACT EQUIPMENT HAS TO BE CLEANED IN PLACE AND/OR IS TOO LARGE TO BE
SUBMERGED IN THE THREE COMPARTMENT SINK; INCLUDING PROCEDURES FOR THE SEMI-COOKING TUB,
SOAKING/CLEANING OF CORN TUB, LARGE GRINDER, ROLLING TRANSPORT CART, AND ALL ATTACHED COMPONENTS.
MUST SUBMIT THE PROCEDURES TO CDPH AT FOOD@CITYOFCHICAGO.ORG FOR APPROVAL PRIOR TO RE-INSPECTION.
PRIORITY FOUNDATION VIOLATION#: 7-38-025. NO CITATION ISSUED.  4-301.14: MUST PROVIDE AN ADEQUATE
FUNCTIONING VENTILATION HOOD FOR THE FOOD EQUIPMENT UNIT THAT IS USED TO PARTIAL COOK THE
MAIZ/CORN.
```

We will need to remove stop words, numbers, single character words (punctuation), and put everything in lowercase. We then rejoin everything into a single string for later use by the CountVectorizer, which treats each string as a row. For the above example it looks like this.

In [5]:

```
stopwords = set(nltk.corpus.stopwords.words('english'))
words = nltk.tokenize.word_tokenize(example)
# Remove single-character tokens (mostly punctuation)
words = [word for word in words if len(word) > 1]
# Remove numbers
words = [word for word in words if word.isalpha()]
# Lowercase all words (default_stopwords are lowercase too)
words = [word.lower() for word in words]
# Remove stopwords
words = [word for word in words if word not in stopwords]
# Rejoin everything into string for the CountVectorizer
words=' '.join(words)
words
```

Out[5]:

```
'must provide step step written detailed cleaning procedure separate piece equipment cleaned place
large submerged three compartment sink including procedures tub corn tub large grinder rolling tra
nsport cart attached components must submit procedures cdph food approval prior priority
foundation violation citation issued must provide adequate functioning ventilation hood food equip
ment unit used partial cook'
```

Now for the whole dataframe:
NOTE: THIS TAKES AWHILE TO RUN

In [6]:

```
stopwords = set(nltk.corpus.stopwords.words('english'))
for i in range(len(df)):
    words = nltk.tokenize.word_tokenize(df.iloc[i]['Violations'])
    words = [word for word in words if len(word) > 1]
    words = [word for word in words if word.isalpha()]
    words = [word.lower() for word in words]
    words = [word for word in words if word not in stopwords]
    words=' '.join(words)
    df.iloc[i]['Violations']=words
```

In [7]:

```
df.iloc[0:5]
```

Out[7]:

	Results	Violations
1	Pass w/ Conditions	person charge unable provide employee health p...
2	Pass w/ Conditions	observed clean policy vomiting diarrheal event...
3	Fail	observed employee health policy premises manag...

	Results	Violations
4	Pass w/ Conditions	employee health policy premises required must ...
6	Fail	person charge city chicago food service sanita...

The Model

Build a classification model, predicting the outcome of inspection – your target variable is "Results". Explain why you selected a particular text pre-processing technique. You can choose to build a binary classifier (limiting your data to Pass / Fail) or multinomial classifier with all available values in Results.

The first thing to do is create a flag variable that maps each value in 'Results' to a numeric value.

In [8]:

```
# convert label to a binary numerical variable
df['result_flag'] = df.Results.map({'Fail':0, 'Pass w/ Conditions':1, 'Pass':2})
df['result_flag'] = df['result_flag'].astype(int)
```

Then split into predictors and labels (for use with CountVectorizer).

In [9]:

```
X=df.Violations
y=df.result_flag
```

Then split into train and test.

In [10]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=7043)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(96044,)
(32015,)
(96044,)
(32015,)
```

Now we vectorize our data, both the train and test sets.

In [11]:

```
from sklearn.feature_extraction.text import CountVectorizer
# instantiate the vectorizer
vect = CountVectorizer(min_df=.03,  ngram_range=(1,3))
# combine fit and transform into a single step
X_train_dtm = vect.fit_transform(X_train)
# transform testing data (using fitted vocabulary) into a document-term matrix
X_test_dtm = vect.transform(X_test)
```

In [12]:

```
# examine the document-term matrix
X_train_dtm
```

Out[12]:

```
<96044x705 sparse matrix of type '<class 'numpy.int64'>'
 with 5751450 stored elements in Compressed Sparse Row format>
```

In [14]:

```
X_test_dtm
```

Out[14]:

```
<32015x705 sparse matrix of type '<class 'numpy.int64'>'
 with 1945652 stored elements in Compressed Sparse Row format>
```

Summation and Explanation of Pre-Processing

For pre-processing in the CountVectorizer, I used a min_df parameter of 0.03 and a n-gram range of 1-3. The min_df made the number of variables in the models more manageable, particularly when I added the n-gram range of 1-3. Without a min_df, the matrix generated has 25,359 columns. With a min_df of as low as 0.03, this falls to 705, even with use of 1-3 n-grams. However, when I increased the min_df the number of variables dropped precipitously which affected accuracy across all models. It appears they are relatively few terms/n-grams that are in a significant percentage of documents.

Naive Bayes Model

In [15]:

```
from sklearn.naive_bayes import MultinomialNB
nb = MultinomialNB()
#train the model
nb.fit(X_train_dtm, y_train)
#make predictions
nb_pred_class=nb.predict(X_test_dtm)
#calculate accuracy
print(accuracy_score(y_test, nb_pred_class))
```

```
0.789130095268
```

Logistic Regression Model

In [16]:

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
#train the model
logreg.fit(X_train_dtm, y_train)
#make predictions
lr_pred_class=logreg.predict(X_test_dtm)
# calculate accuracy of class predictions
print(accuracy_score(y_test, lr_pred_class))
```

```
0.923754490083
```

Support Vector Machine

In [17]:

```
from sklearn.linear_model import SGDClassifier
# instantiate a SVM model
svm = SGDClassifier(max_iter=200, tol=None)
#train the model
svm.fit(X_train_dtm, y_train)
# make class predictions for X_test_dtm
svm_pred_class = svm.predict(X_test_dtm)
print(accuracy_score(y_test, svm_pred_class))
```

```
0.917132594097
```

Commentary

The logistic regression and SVM models performed the best and by some margin. Interestingly, the use of n-grams 1-3 did not improve the accuracy of these models. In fact, the Naive Bayes model performed somewhat worse with its accuracy dropping from 0.84 to 0.79. However, I thought it best to use n-grams and so chose a CountVectorizer with min_df=0.03 and n-gram range of 1-3.

Visualize results of at least two text classifiers and select the most robust one.

In [18]:

```
import matplotlib.pyplot as plt
import seaborn as sns
```

First we will visualize the results of the Logistic Regression Model:

In [19]:

```
confusion_lr = confusion_matrix(y_test, lr_pred_class)
confusion_svm = confusion_matrix(y_test, svm_pred_class)
confusion_lr
```

Out[19]:

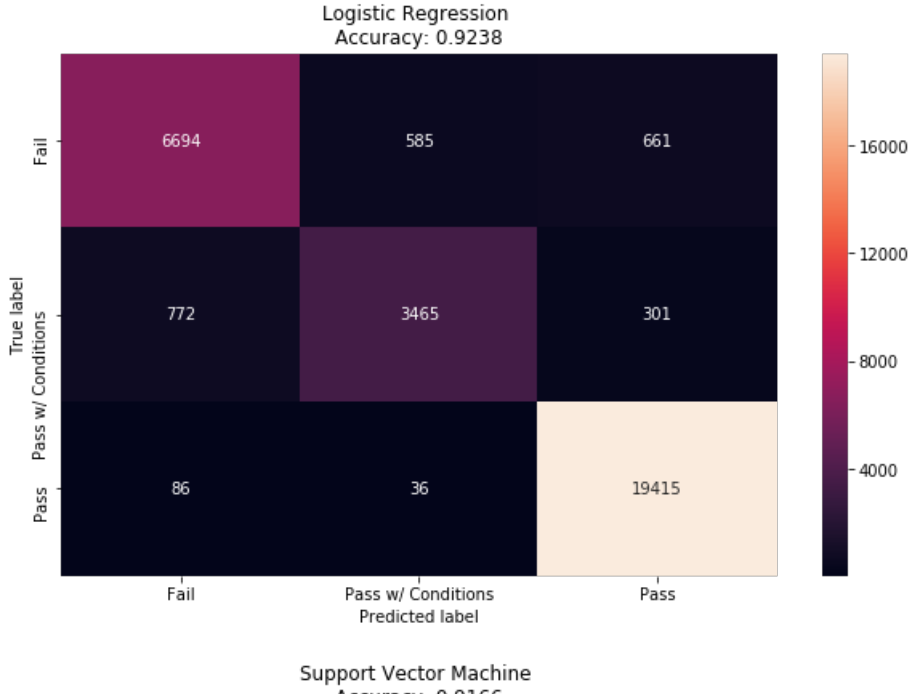
```
array([[ 6694,   585,   661],
       [  772,  3465,   301],
       [   86,    36, 19415]], dtype=int64)
```

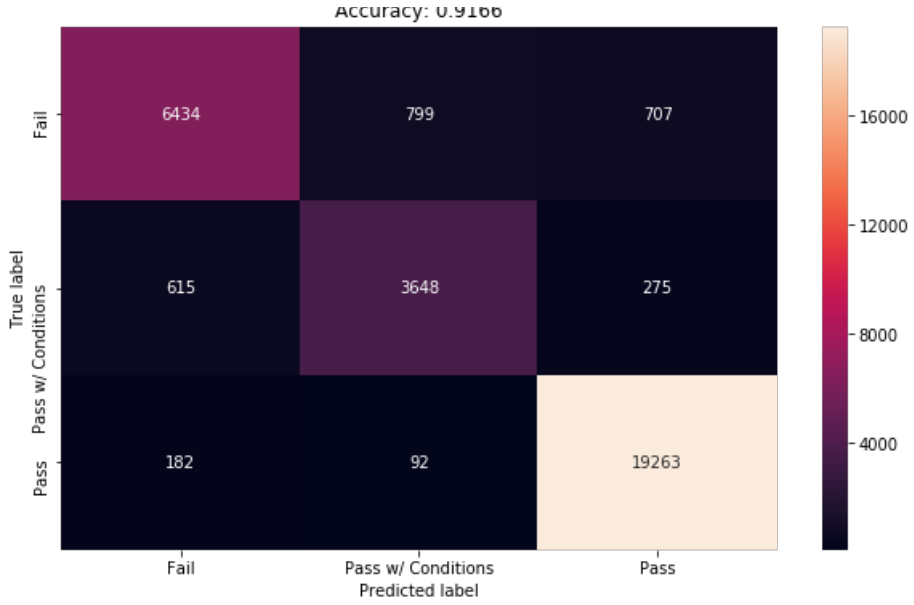
In [134]:

```
df_lr = pd.DataFrame(confusion_lr, index = [['Fail', 'Pass w/ Conditions', 'Pass']], columns = [['F
ail', 'Pass w/ Conditions', 'Pass']])
plt.figure(figsize = (10,6))
sns.heatmap(df_lr, annot=True, fmt='g')
plt.title('Logistic Regression \nAccuracy: {0:1.4f}'.format(accuracy_score(y_test, lr_pred_class)))
plt.ylabel('True label')
plt.xlabel('Predicted label')

df_svm = pd.DataFrame(confusion_svm, index = [['Fail', 'Pass w/ Conditions', 'Pass']], columns = [[
'Fail', 'Pass w/ Conditions', 'Pass']])
plt.figure(figsize = (10,6))
sns.heatmap(df_svm, annot=True, fmt='g')
plt.title('Support Vector Machine \nAccuracy: {0:1.4f}'.format(accuracy_score(y_test, svm_pred_clas
s)))
plt.ylabel('True label')
plt.xlabel('Predicted label')

plt.show()
```





The Logistic Regression model performs slightly better, and so I would pick it unless business use case dictates otherwise. That does not appear to be the case here. The SVM misses more True Fails than the Logistic Regression, which is likely what one would want to catch. For this reason, I choose the Logistic Regression model.