Portfolio 7: Text Classification

Bushra Rahman

Text Classification on Spam Data

"NLP Email Classification" is a dataset of email bodies and labels for text classification, uploaded to Kaggle (URL https://www.kaggle.com/datasets/datatattle/email-classification-nlp). This dataset consists of a folder named **archive** which contains 2 CSV files, **SMS_train** and **SMS_test**. SMS_train contains **957** observations, and SMS_test contains **125** observations. Both CSV files have 3 columns: the row number of the observation, a string containing the **message body**, and the binary label for **spam/non-spam**. Altogether, the two CSV files contain **1,082** observations. The percentage of the train/test split between CSV_train and CSV_test is about 88/12. However, for the purposes of this program, we will ignore the preset train/test split and create our own.

Opening & Exploring the CSV Files

The following code uses Pandas to open each CSV file and stores it in its own data frame. Each data frame's shape and first few rows are output.

In [1]:
```
import pathlib
import pandas as pd

path_string = pathlib.Path.cwd().joinpath('archive/SMS_train.csv')

train_df = pd.read_csv(path_string, header=0, usecols=[1,2], encoding='latin-1')
print('Rows and columns of SMS_train:', train_df.shape)
print(train_df.head())
```

```
Rows and columns of SMS_train: (957, 2)
                                        Message_body     Label
0                           Rofl. Its true to its name  Non-Spam
1  The guy did some bitching but I acted like i'd...  Non-Spam
2  Pity, * was in mood for that. So...any other s...  Non-Spam
3                Will ü b going to esplanade fr home?  Non-Spam
4  This is the 2nd time we have tried 2 contact u...      Spam
```

In [2]:
```
path_string = pathlib.Path.cwd().joinpath('archive/SMS_test.csv')

test_df = pd.read_csv(path_string, header=0, usecols=[1,2], encoding='latin-1')
print('Rows and columns of SMS_test:', test_df.shape)
print(test_df.head())
```

```
Rows and columns of SMS_test: (125, 2)
                                         Message_body Label
0  UpgrdCentre Orange customer, you may now claim...  Spam
1  Loan for any purpose £500 - £75,000. Homeowner...  Spam
2  Congrats! Nokia 3650 video camera phone is you...  Spam
3  URGENT! Your Mobile number has been awarded wi...  Spam
4  Someone has contacted our dating service and e...  Spam
```

The two data frames are then concantenated into one data frame. In order to make later evaluation of metrics easier, the categorical data under 'Label' is converted into numeric form (ie. **0 for non-spam and 1 for spam**).

In [3]:
```
# Concatenate into one df
df = pd.concat([train_df, test_df])
print('Rows and columns of concatenated df:', df.shape)

# Replace Non-Spam/Spam with 0/1
df['Label'].replace(['Non-Spam', 'Spam'], [0, 1], inplace=True)
print(df.head())
```

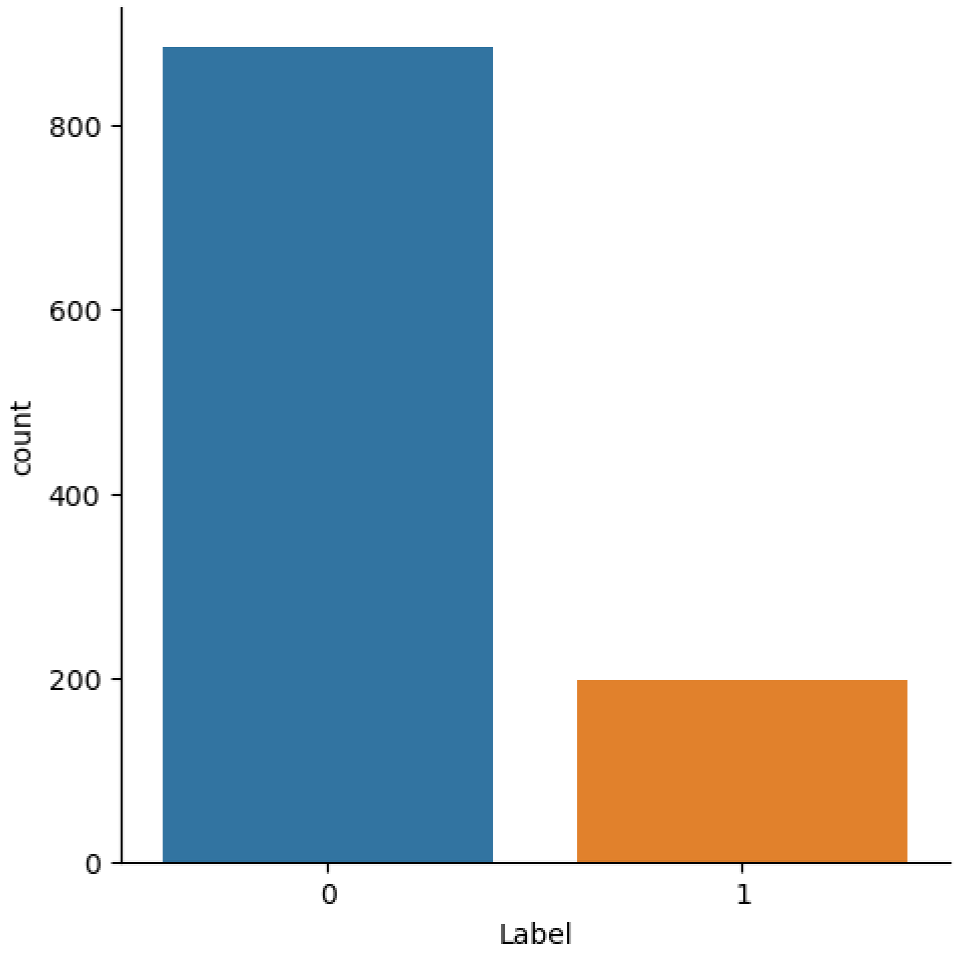
```
Rows and columns of concatenated df: (1082, 2)
                                         Message_body  Label
0                          Rofl. Its true to its name      0
1  The guy did some bitching but I acted like i'd...      0
2  Pity, * was in mood for that. So...any other s...      0
3                Will ü b going to esplanade fr home?      0
4  This is the 2nd time we have tried 2 contact u...      1
```

Target Distribution Graph

The following code uses Seaborn to show the distribution of the target class **Label**.

In [4]:
```
import seaborn as sb
import matplotlib.pyplot as plt  # Plots created using seaborn need to be displayed li

sb.catplot(x='Label', kind='count', data=df)
plt.show()
```



Naive Bayes

Naive Bayes is a classification algorithm that calculates the probability of the positive class for each data point, which is 1 for 'Spam' in this case. This algorithm makes the "naive" assumption that each predictor is independent, so that their joint probabilities don't have to be calculated. However, this assumption usually works well and allows this algorithm to act as a good baseline for comparison against other classifiers.

First, before applying the classifier, the data must be properly set up. Stopwords are removed and a **tf-idf vectorizer** is applied to the train and test sets.

In [5]:
```
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

# Set up tf-idf vectorizer with stop words
stopwords = set(stopwords.words('english'))
vectorizer = TfidfVectorizer(stop_words=stopwords)

# Set up X and y
X = df.Message_body
y = df.Label

# Divide into train/test sets on an 80/20 split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.
# Apply tf-idf vectorizer
X_train = vectorizer.fit_transform(X_train)  # Fit and transform the train data
X_test = vectorizer.transform(X_test)        # Only transform the test data

# See that their combined rows add up to 1082
print('Size of X_train:', X_train.shape)
print('Size of X_test:', X_test.shape)
```

```
Size of X_train: (865, 2931)
Size of X_test: (217, 2931)
```

Next, train the Naive Bayes classifier and evaluate on the test data. **MultinomialNB** is used because it can handle features that are discrete, like word counts, or tf-idf representations. An alternative is to use BernoulliNB.

In [6]:
```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, c

# Train classifier
naive_bayes = MultinomialNB()
naive_bayes.fit(X_train, y_train)

# Predict on the test data
pred = naive_bayes.predict(X_test)

# Print confusion matrix
print(confusion_matrix(y_test, pred))
# Form of confusion matrix:
# [[TP   FP
#   FN   TN]]
```

```
[[177   0]
 [ 12  28]]
```

The **confusion matrix** shows 177 true positives, 0 false positives, 12 false negatives, and 28 true negatives. The positive cases were all successfully classified, which is a good sign of the Naive Bayes algorithm's performance. However, the classification of the negative cases could use some improvement.

Finally, evaluate the accuracy of the predictions. An **evaluation report** can be generated by sklearn. The subsequent code verifies the precision and recall scores given by the report.

In [7]:
```
from sklearn.metrics import classification_report

# Evaluation metrics for NB
print(classification_report(y_test, pred))
print('______________________________________________________')
print('______________________________________________________')

print('\nPrecision and Recall for Non-Spam (0):')
print('Precision score: ', precision_score(y_test, pred, pos_label=0))
print('Recall score: ', recall_score(y_test, pred, pos_label=0))

print('\nPrecision and Recall for Spam (1):')
print('Precision score: ', precision_score(y_test, pred))
print('Recall score: ', recall_score(y_test, pred))
print('\nAccuracy score: ', accuracy_score(y_test, pred))
print('F1 score: ', f1_score(y_test, pred))
```

```
              precision    recall  f1-score   support

           0       0.94      1.00      0.97       177
           1       1.00      0.70      0.82        40

    accuracy                           0.94       217
   macro avg       0.97      0.85      0.90       217
weighted avg       0.95      0.94      0.94       217

_____________________________________________________
_____________________________________________________

Precision and Recall for Non-Spam (0):
Precision score:  0.9365079365079365
Recall score:  1.0

Precision and Recall for Spam (1):
Precision score:  1.0
Recall score:  0.7

Accuracy score:  0.9447004608294931
F1 score:  0.8235294117647058
```

The **precision** metric measures accuracy of the positive class using the formula TP/(TP+FP). The **recall** metric measures the true positive rate using the formula TP/(TP+FN). **Accuracy** is the measure of true positives and true negatives, given by the formula (TP+TN)/(TP+TN+FP+FN). All 3 of these metrics range from 0 to 1, with values closer to 1 being the desired values.

The report shows that recall for Non-Spam and precision for Spam are both 1, which is good. The other metrics are okay, but could be better.

Next, try using **regex** to improve the accuracy of the results. Regex is used here to recognize punctuation and capital case in the data.

In [8]:
```
import re

df['Message_body'].replace('[\d][\d]+', ' num ', regex=True, inplace=True)
df['Message_body'].replace('[!@#*][!@#*]+', ' punct ', regex=True, inplace=True)
df['Message_body'].replace('[A-Z][A-Z]+', ' caps ', regex=True, inplace=True)

# Set up X and y again
X = df.Message_body
y = df.Label

# Divide into train/test sets again
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.

# Apply tf-idf vectorizer again
X_train = vectorizer.fit_transform(X_train)
X_test = vectorizer.transform(X_test)

# Train classifier again
naive_bayes = MultinomialNB()
```

```python
naive_bayes.fit(X_train, y_train)

# Predict on the test data
pred = naive_bayes.predict(X_test)

# Print confusion matrix
print(confusion_matrix(y_test, pred))

# Evaluation metrics for NB w/ RegEx
print(classification_report(y_test, pred))
print('_____________________________________________________')
print('_____________________________________________________')

print('\nPrecision and Recall for Non-Spam (0):')
print('Precision score: ', precision_score(y_test, pred, pos_label=0))
print('Recall score: ', recall_score(y_test, pred, pos_label=0))

print('\nPrecision and Recall for Spam (1):')
print('Precision score: ', precision_score(y_test, pred))
print('Recall score: ', recall_score(y_test, pred))

print('\nAccuracy score: ', accuracy_score(y_test, pred))
print('F1 score: ', f1_score(y_test, pred))
```

```
[[177   0]
 [  6  34]]
              precision    recall  f1-score   support

           0       0.97      1.00      0.98       177
           1       1.00      0.85      0.92        40

    accuracy                           0.97       217
   macro avg       0.98      0.93      0.95       217
weighted avg       0.97      0.97      0.97       217

_____________________________________________________
_____________________________________________________

Precision and Recall for Non-Spam (0):
Precision score:  0.9672131147540983
Recall score:  1.0

Precision and Recall for Spam (1):
Precision score:  1.0
Recall score:  0.85

Accuracy score:  0.9723502304147466
F1 score:  0.9189189189189189
```

These metrics show that the results have improved after using regex. The confusion matrix shows that that the amount of false negatives went down and the amoung of true negatives went up, which is a good sign of improved accuracy. Additionally, the precision score for Non-Spam went up from 0.94 to 0.97, and the recall score for Spam went up from 0.70 to 0.85. Finally, the overall accuracy score went up from about 0.94 to 0.97, and the overall F1 score went up from 0.82 to 0.92. So, overall, using regex improved the performance of the Naive Bayes algorithm.

Logistic Regression

Logistic regression, contrary to its name, is a qualitative classification algorithm that creates linear decision boundaries between data points of different classes. Let's see how logistic regression compares against Naive Bayes.

In [9]:
```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss

# Set up tf-idf vectorizer again
vectorizer = TfidfVectorizer(binary=True)

# Set up X and y
X = df.Message_body
y = df.Label

# Divide into train/test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.

X_train = vectorizer.fit_transform(X_train)  # Fit and transform the train data
X_test = vectorizer.transform(X_test)        # Only transform the test data

# Train classifier
classifier = LogisticRegression(solver='lbfgs', class_weight='balanced')
classifier.fit(X_train, y_train)

# Predict on the test data
pred = classifier.predict(X_test)

# Print confusion matrix
print(confusion_matrix(y_test, pred))

# Evaluation metrics for LogReg
print(classification_report(y_test, pred))
print('_____________________________________________________')
print('_____________________________________________________')
print('\nPrecision and Recall for Non-Spam (0):')
print('Precision score: ', precision_score(y_test, pred, pos_label=0))
print('Recall score: ', recall_score(y_test, pred, pos_label=0))

print('\nPrecision and Recall for Spam (1):')
print('Precision score: ', precision_score(y_test, pred))
print('Recall score: ', recall_score(y_test, pred))

print('\nAccuracy score: ', accuracy_score(y_test, pred))
print('F1 score: ', f1_score(y_test, pred))

probs = classifier.predict_proba(X_test)
print('\nLog loss: ', log_loss(y_test, probs))
```

```
[[175   2]
 [  1  39]]
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       177
           1       0.95      0.97      0.96        40

    accuracy                           0.99       217
   macro avg       0.97      0.98      0.98       217
weighted avg       0.99      0.99      0.99       217

_____________________________________________________
_____________________________________________________

Precision and Recall for Non-Spam (0):
Precision score:  0.9943181818181818
Recall score:  0.9887005649717514

Precision and Recall for Spam (1):
Precision score:  0.9512195121951219
Recall score:  0.975

Accuracy score:  0.9861751152073732
F1 score:  0.9629629629629629

Log loss:  0.21652373788268733
```

These metrics show some improvement from Naive Bayes. A couple of the positive cases were misclassified, but a much greater amount of negative cases were correctly classified compared to Naive Bayes. The additional statistic of log loss penalizes misclassifications, thus meaning a low log loss score is better. Log loss was 0.21, which is fairly low.

The precision of Non-Spam and the recall of Spam, which were lacking in Naive Bayes, improved significantly. However, the recall for Non-Spam and precision for Spam, which were both 1 with Naive Bayes, have gone down somewhat, so there is a trade-off there. However, the accuracy and F1 scores are 0.99 and 0.96 respectively, which is much better compared to Naive Bayes' improved scores of 0.97 and 0.92. So, overall, logistic regression performed better than Naive Bayes.

Neural Networks

Neural networks are a recent development in the machine learning field, although the algorithmic concepts underlying them have been known for much longer. If a machine learning problem has sufficient data that is distributed more complexly than linearly, then neural networks tend to perform better than other classification algorithms do. This may give neural networks an advantage over logistic regression in this case.

First set up the data as with before. This time the setup is slightly different; X is set up with the vectorizer directly applied. This is necessary for the NN algorithm to work as intended.

In [10]:
```python
# Set up tf-idf vectorizer again
vectorizer = TfidfVectorizer(stop_words=stopwords, binary=True)
```

```python
# Set up X and y w/ applied vectorizer
X = vectorizer.fit_transform(df.Message_body)
y = df.Label

# Divide into train/test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, train_size=0.
```

Now apply the NN classifier on the training data and evaluate on the test data.

```python
In [11]: from sklearn.neural_network import MLPClassifier

# Train classifier
classifier = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(15, 2), rar
classifier.fit(X_train, y_train)

# Predict on the test data
pred = classifier.predict(X_test)

# Print confusion matrix
print(confusion_matrix(y_test, pred))

# Evaluation metrics for NN
print(classification_report(y_test, pred))
print('_____________________________________________________')
print('_____________________________________________________')
print('\nPrecision and Recall for Non-Spam (0):')
print('Precision score: ', precision_score(y_test, pred, pos_label=0))
print('Recall score: ', recall_score(y_test, pred, pos_label=0))

print('\nPrecision and Recall for Spam (1):')
print('Precision score: ', precision_score(y_test, pred))
print('Recall score: ', recall_score(y_test, pred))

print('\nAccuracy score: ', accuracy_score(y_test, pred))
print('F1 score: ', f1_score(y_test, pred))
```

```
[[174   3]
 [  6  34]]
              precision    recall  f1-score   support

           0       0.97      0.98      0.97       177
           1       0.92      0.85      0.88        40

    accuracy                           0.96       217
   macro avg       0.94      0.92      0.93       217
weighted avg       0.96      0.96      0.96       217

_____________________________________________________
_____________________________________________________

Precision and Recall for Non-Spam (0):
Precision score:  0.9666666666666667
Recall score:  0.9830508474576272

Precision and Recall for Spam (1):
Precision score:  0.918918918918919
Recall score:  0.85

Accuracy score:  0.9585253456221198
F1 score:  0.8831168831168831
```

Performance Analysis

Out of the four approaches tested — neural networks, logistic regression, and Naive Bayes with and without regex — overall logistic regression had the best accuracy score. The confusion matrix for neural networks showed more misclassified cases than logistic regression, and only comparable precision and recall statistics to those of the improved Naive Bayes approach. The reason neural networks may not have performed the best is because the dataset used was small. Additionally, logistic regression performing the best implies that the data follows a strongly linear distribution, whereas neural networks performs better on complex data.