Udacity Machine Learning Nanodegree

Capstone Project

Spam Email Detection

Gaurav Jain
March 2020

- ## Project Overview

  Nowadays, along with desired emails we get a lot of unsolicited emails like promotional, social, and other secondary types. Hence it becomes hard to distinguish between spam and genuine emails and create a lot of noise in our inbox. There are cases where an attacker can send you spam mails and can trick you to expose your private data using phishing or malicious attachments. In this project we'll be finding a model that most accurately detects if an email is Spam or not.

  In this project we'll be making use of the dataset provided by Apache SpamAssassin. We'll explore different algorithms and compare their performances in order to find the best suitable model for the given dataset.

- ## Problem Statement

  Most of the time spam mails do not greet you by name, they simply start with Hello, Hi, Dear, etc. Sometimes the subject of spam emails also doesn't make any sense and usually contains words such as free, money, discount, help etc. In some cases we can even see that email address is really weird e.g webscrapper57@hackerdev.com.

My curiosity behind this problem is that, a mail can be spam for one person and for another It can be a legitimate email so it should learn from the user's emails and behave as per that user.

The problem is to use the aforementioned dataset and find the best model which can predict whether an email is spam or not with high accuracy .

- ## Metrics

  Evaluation metrics would be an accuracy score along with Precision & Recall calculated using True/False Positive/Negative.

  $$\text{Accuracy} = \frac{true\ positives + true\ negatives}{total\_item\ in\ the\ dataset}$$

  $$\text{Precision} = \frac{true\ positives}{true\ positives + false\ positives}$$

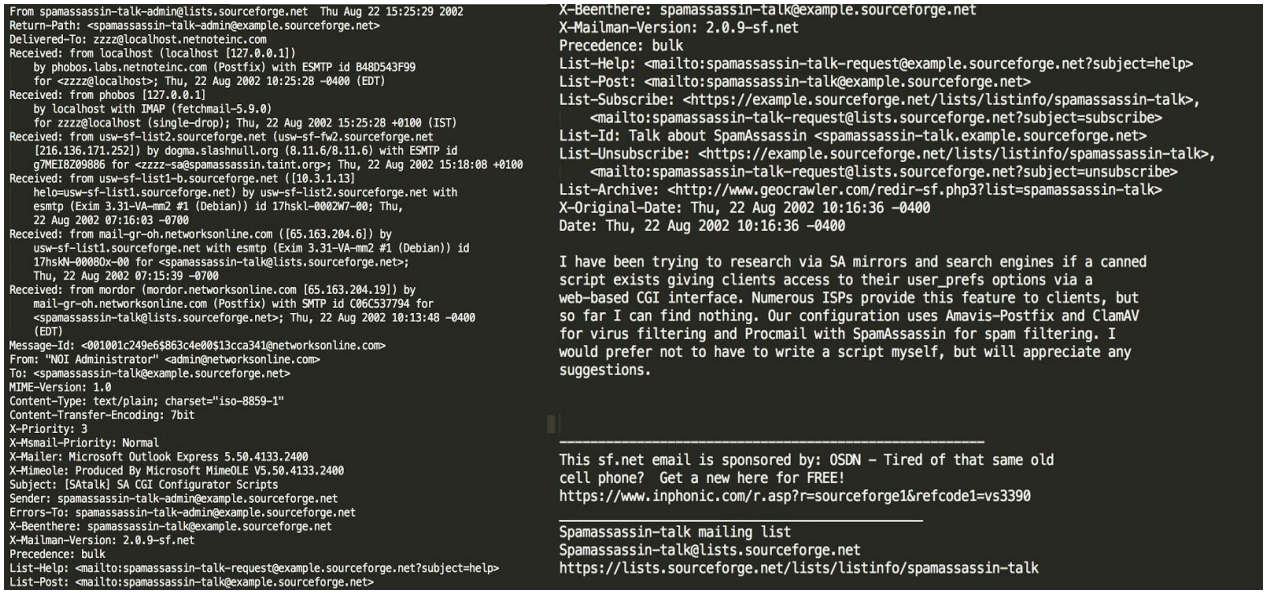
  $$\text{Recall} = \frac{true\ positives}{true\ positives + false\ negatives}$$

  We can also display the confusion matrix to visualize these numbers.

- ## Data Exploration

  We'll take *30228_easy_ham.tar.bz2* and *30228_spam.tar.bz2* from Apache SpamAssassin's public datasets https://spamassassin.apache.org/old/publiccorpus/. There are approximately 2500 ham and 500 spam emails in the dataset.

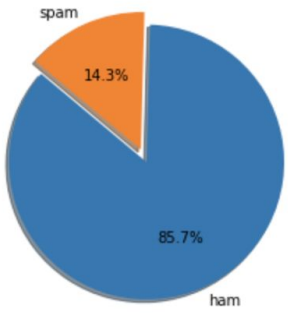
  Each email is stored in a raw format in a separate file example shown below-



As we can see, the first few lines contain information such as email addresses, time stamp, ip/dns address and some other headers & metadata that is not something that we are interested in. On the right side of the above image we can see that the actual message starts from "*I have been…"* . So we'll extract that message only.

We can also notice that the content type of the above email is text/plain (on the left side of the above image after MIME-version). However there are many emails which have different content types e.g text/html. For the sake of keeping the dataset simple, we'll only consider these content types and ignore others as they would be very few. In the below Pie chart we can visualize how many spam and ham mails we have after above filtering.

```
labels = ['ham', 'spam']
sizes = [len(ham_data), len(spam_data)]
plt.pie(sizes, explode=(0.1, 0), labels=labels, autopct='%1.1f%%', shadow=True, startangle=140)
plt.axis('equal')
plt.show()
```



- ## Algorithms and Techniques

  We'll start with a simple GaussianNB algorithm and then the LogisticRegression algorithm to see how it performs against the GaussianNB and in the end we'll also explore XGBoost model to come to a conclusion of which model works best for this. XGBoost algorithm is an extended version of the gradient boosting algorithm. Xgboost is a decision tree based algorithm with enhanced performance.It is basically designed to enhance the performance and speed of a Machine Learning model by performing parallel computations.

- ## Benchmark

  For benchmarking we'll use the simple Gaussian Naive Bayesian model and will try to get the improved result against it. The Reason behind using Gaussian NB is that it is fast for both training and prediction.

- ## Data Preprocessing & Refinement

  We will simplify the data set by removing headers/metadata, urls and other unnecessary information. We'll have two columns in the final

dataset. Column1 would be the actual email message and column2 would be a label that represents the email is spam or not. We'll split the data in training and testing set 70%, 30% respectively. After doing this we'll have a clear dataset defined for both email and label, for both training and testing set.

```
X = np.array(ham_data + spam_data)
```

```
y = np.array([0] * len(ham_data) + [1] * len(spam_data))
```

```
train_X, test_X, train_y, test_y = train_test_split(X, y, test_size=0.3)
```

Once we have the sanitized data, we'll tokenize both training and testing messages(splitting into words).

```
def email_to_words(email):
    nltk.download("stopwords", quiet=True)
    email = email.decode('utf-8')
    email = re.sub(r"http\S+", "", email)  # replace all urls with empty string
    email = re.sub("\d+", "", email)  # replace numbers with empty string
    email = re.sub("[^a-zA-Z0-9]", " ", email.lower())  # convert all characters into lowercase letters and replace all
    email = re.sub(r"\b[a-z]\b", "", email)  # replace single characters with empty string

    words = email.split()  # split into words
    words = [w for w in words if w not in stopwords.words("english")]  # remove all 'bad' words
    words = [PorterStemmer().stem(w) for w in words]

    return words
```

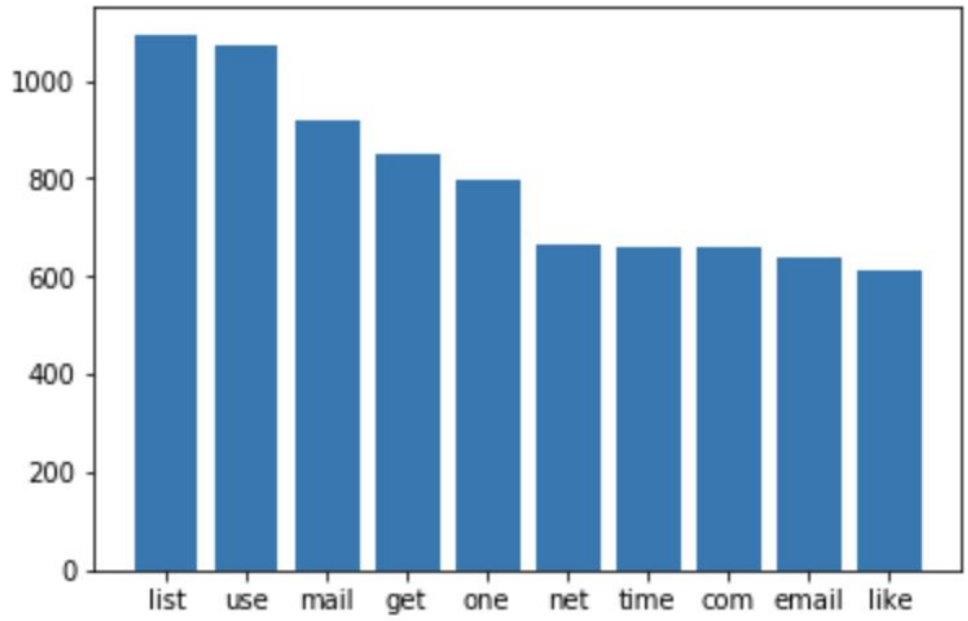
After doing that we'll have all
the messages converted into a list of words, we count the frequency of words across all messages and get the top 10000 words to proceed with. Here are the top 10 most frequent words.

```
# Top 10 most frequent words
sorted_words[:10]
```

```
[('list', 1094),
 ('use', 1074),
 ('mail', 917),
 ('get', 848),
 ('one', 796),
 ('net', 664),
 ('time', 658),
 ('com', 657),
 ('email', 640),
 ('like', 614)]
```

```
plt.bar(*list(zip(*sorted_words[:10])))
```

```
<BarContainer object of 10 artists>
```



After tokenizing messages/mails, we will assign each word a unique integer number (starting from 1), 0 is reserved for “no word”. Once we have this

word-integer mapping in place, we'll convert each mail into a series of integers from the mapping we just created. Also, we'll limit all the messages to have only 1000 words so we'll truncate them. One can have a better understanding by looking at the code below.

```
def convert_and_pad(word_dict, email, pad):
    NOWORD = 0  # We will use 0 to represent the 'no word' category

    working_sentence = [NOWORD] * pad
    for word_index, word in enumerate(email[:pad]):
        if word in word_dict:
            working_sentence[word_index] = word_dict[word]

    return working_sentence, min(len(email), pad)

def convert_and_pad_data(word_dict, data, pad):
    result = []
    lengths = []

    for email in data:
        converted, length = convert_and_pad(word_dict, email, pad)
        result.append(converted)
        lengths.append(length)

    return np.array(result), np.array(lengths)
```

```
train_X, train_X_len = convert_and_pad_data(word_dict, train_X, 1000)
```

```
test_X, test_X_len = convert_and_pad_data(word_dict, test_X, 1000)
```

```
train_X[4]
```

```
array([ 163,  257,    0,  159,  894,    1,    0,    2,  683,  894,  950,
       2397,   43, 1000,   98,  591,   18,   98,  171,   71,  353,  323,
         61, 1631, 1644,  406,    0,  703,  798,    0,    0,   27,    0,
          4, 5126,    0,   80,    4,  877,  154,    0,   94, 2174,    0,
         70,   94, 8262,   31,   47,   48,  465,   83,  439,  323,    0,
          8,    0,   22,  552,    5,   22,  114,   53,   25,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
```

Now, our data is ready to fit into algorithms. We'll also create csv files with this data for test and training set so that we can reuse those files later.

- ## Implementation

  After preprocessing steps, we'll use the LogisticRegression algorithm to see how it performs against the GaussianNB that we tried first, and in the end we'll also explore XGBoost model to come to a conclusion of which model works best for this.

  We'll calculate an Accuracy Score along with Precision & Recall for all three models to get the idea of False positive and False Negative.

- ## Model Evaluation and Validation

  We see that XGBoost works better than other two models without any hyperparameter tuning. We used scikit's **accuracy_score** utility to calculate performance. Our goal was to find a model that perform better than simple GaussianNB. XGBoost met our expectation and surpassed even LogisticRegression.

Model	**Accuracy Score**
Gaussian Naive Bayesian	0.8199
Logistic Regression	0.7914
XGBoost	0.8530
XGBoost (With hyperparameter Tuning)	0.8909

- ## Justification

  Final hyperparameter tuned model got an accuracy of almost 90% which is a significant(10%) improvement from our benchmark model.

We can still improve this by having more dataset and further hyperparameter tuning.

- ## References

https://en.wikipedia.org/wiki/Linear_regression

https://en.wikipedia.org/wiki/Binary_classification

https://en.wikipedia.org/wiki/Supervised_learning

https://xgboost.readthedocs.io/en/latest/