Introduction

The domain i have chosen is the research of email communications, identifying spam emails amongst real emails.

According to Statista [2], 45% of the worlds emails were unsolicited spam in the year 2021 [1] which are scary stats. There are different types of spam that gets sent, some not being harmful such as 'marketing mails' from legitimate companies, though some can be harmful such as spoofing spam containing malware. Spam emails are costing companies billions of dollars each year in security breaches, delayment of productivity amongst other issues. Though the more inhumane are spoofing emails targeting individuals for the purpose of gaining access to their banking and investment accounts in order to rob them.

These issues are enough of a reason to work on a solution to this problem. Classifying emails into spam and non-spam would be a contribution to a solution to the problem, and is the contribution we will be trying to achieve in this project.

If we could classify emails into spam and non-spam with a 100% success rate, this would directly solve our predefined problem and would also allow users to communicate via email securely again without everything being a constant threat. Although 100% would be close to impossible as while the methods and ideas of filtering spam improve, so do the cybercriminals involved in these spam attacks[4]. One should always remember, when there is money in an industry, that industry will evolve. Within 2022 it was reported spam sender sites could earn up around 7000 USD per day [5].

Since emails are text, and are classified as spam or non-spam, it will be suited to handle this as a binary classification problem which we will be proceeding with.

Dataset

I had come across a reading on classification modeling that directed me to a very nice dataset on Spam emails, which i will be using here. The dataset is available online at Kaggle[3]. This dataset seems to be well suited for this projects criteria.

Dataset Description

This is a structured dataset that contains data that was sourced via 'spam assassin' [6] which is an open source spam project used to classify emails and block spam. This particular dataset was published in 2020.

- The dataset was made available as a CSV file.
- The file size is 5.5mb, and contains 5172 records with 2 attributes.

- The structure of the dataset is in NF1, where each record represents one email. Although there is some structure within the 'text' attribute, being a 'subject' and 'body' field, this dataset doesn't allow an easy way of re-structuring this. So we will not re-structuring the data, we will just remove the structure.

Data-types

The dataset's data-types are a mix of textual as well as numerical. Currently the data set consists of 2 attributes being:

- 'text' : Being a complex data structure of text, containing a 'Subject' field as well as a 'Body' field
- 'spam' : Being a boolean, identifying if it's a spam email or non-spam

Although there is a mixture of data-types and the 'text' attribute is of a complex data structure, computationally they all are atomic within the date-set, so we won't need to normalize the structure further.

Objectives

Our objective in this project is:

- Develop a method in which we can classify emails into 'spam' and 'non-spam' based on the title and body of the email.

Achieving this objective will be beneficial to potential victims of unsolicited mails, by identifying and removing spam emails before the user becomes a victim would be the goal. How well we are able to classify emails will determine how useful our project would be at contributing to the domains problems.

We will be using a statistical approach using supervised learning, focusing on the binary classification of emails, where our objective is to accurately classify an email based on the textual context. Future projects could be the implementation of this classification model into email servers to filter out spam emails.

Evaluation Metrics

As we are going to be creating a classification model, we are going to need to evaluate how well our model does. We will be doing this by creating a simpler baseline model to compare our model iterations to.

We will create a Naive Bayes classification model as our baseline, where we will then score this model and gain values for the following:

- Accuracy : The proportion of correct predictions over all predictions
- Recall : The ratio of True Positives to Actual Positive counts, So the measurement of how well the model identifies a spam email.
- Precision : The ratio of True Positives to total count of True Positives, So measures if only spam was identified as spam.

- F1 : A combination of recall and precision

The precision score would be the most important one for us, as we would prefer non-spam not being classified as spam, as this would create the issue of people not receiving possibly important emails. It would be more acceptable for the classifier to having spam emails being classified as non-spam. While we proceed with this project we will see, we end up with very close scores with our baseline and our main model, so having a few different scoring metrics to compare, will help us evaluate our model.

We will then create our main model with the goal of beating the baseline models scores.

Implementation

In this section, we will:

- Process the dataset to get it to a suitable state and normalized enough for training a classification model.
- Create a simple baseline classification model that will be able to classify textual data, possibly a Naive Bayes classification model.
- Create a classification model that will be able to classify textual data better than our base model.

```
# Setting up our notebook
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

```
# Importing libraries
import pandas as pd
import numpy as np
from statistics import mean

from sklearn.metrics import classification_report
from sklearn.metrics import r2_score
from sklearn.pipeline import make_pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split

from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
```

```
# Importing the dataset
emails = pd.read_csv(r'./data/emails.csv')
```

```
# Lets see a fraction of our data
emails.head()
```

	text	spam
0	Subject: naturally irresistible your corporate...	1
1	Subject: the stock trading gunslinger fanny i...	1
2	Subject: unbelievable new homes made easy im ...	1
3	Subject: 4 color printing special request add...	1
4	Subject: do not have money , get software cds ...	1

Pre-processing

The dataset is almost clean enough for us to train models on, though there still is some issues we will need to address for both our baseline model and any other model we choose to train with this dataset:

- Remove any records containing NULL fields
- Remove the structure in the 'text' attribute
- Balancing the dataset
- Split our dataset into a training and testing set

We also aren't able to create any sort of classification while our datasets 'text' attribute is in a textual state as it currently is. We will need to convert this attribute to be represented numerically. We will decide on this method within each model.

We aren't able to make a accurate analysis with the present of **NULL** values, and there is no way to populate these, so we will be deleting these records.

```
# Identify if any attributes contain NULL
emails.isna().sum()
```

```
text    0
spam    0
dtype: int64
```

Great, there are no NULL values, we can proceed with removing the structure within the 'text' feild. There is no easy way of doing this but just removing the string 'Subject: ' from the email records.

```
# Removing structure in the dataset, where 'subject' is indicating structure and
        not part of the data
emails['text'] = emails['text'].str.replace('Subject: ','')
```

Next we will make sure our dataset is balanced. As with Naive Bayes which will be our baseline, our scoring can get corrupted if our dataset is skewed to one of the binary classifications. So in the case the dataset isn't balanced, we should make it balanced.

```
# Counting spam and non spam records
spam = emails.loc[emails['spam'] == 1]
non_spam = emails.loc[emails['spam'] == 0]

print("Orginal spam count: ", len(spam))
print("Orginal non_spam count: ", len(non_spam))

# Shuffling the datasets
shuffled_spam = spam.sample(frac=1)
shuffled_nonspam = non_spam.sample(frac=1)

# Identifying which group has the most values
if len(spam) > len(non_spam):
    max_length = len(non_spam)
else:
    max_length = len(spam)

spam_concat = shuffled_spam[:max_length]
non_spam_concat = shuffled_nonspam[:max_length]

# Creating our balanced dataset
ballanced_emails = pd.concat([spam_concat, non_spam_concat], axis=0)
```

```
Orginal spam count:  1368
Orginal non_spam count:  4360
```

Now our dataset is balanced, we can create our data sets, which will consist of training and testing groups of our data for our classification model. We will create 5 different versions of the data for analysing the average scores form each model.

We have opted for a ratio of 8:2, for our training:testing groupings.

```
# Lets create our datasets
data_sets = []
for count in range(0,5):
    X_train,X_test,y_train,y_test = train_test_split(ballanced_emails.text,
        ballanced_emails.spam, test_size=0.20, random_state=count)
    data_sets.append([X_train,X_test,y_train,y_test])
```

Let's create a scoring function for our models

```
'''
GetAverageScores

Takes 2 arguments
```

```
- data_sets : which is an array of objects of the form
        {X_train,X_test,y_train,y_test}
- model : which is the ML model to test on

Returns an object containing average scores on the model
'''
def GetAverageScores(data_sets, model):
    # Lets iterate over our dataset and obtain an average score for model
    scores = []
    for data_set in data_sets:
        # Lets fit the model with our training data
        model.fit(data_set[0],data_set[2])

        # Lets do a prediction with our test data
        prediction = model.predict(data_set[1])

        # Record the bayes scores
        report = classification_report(data_set[3], prediction, output_dict = True)
        score = {
            'R2': r2_score(data_set[3],prediction),
            'Accuracy' : report['accuracy'],
            'Average_precision' : report['macro avg']['precision'],
            'Average_recall' : report['macro avg']['recall'],
            'Average_f1' : report['macro avg']['f1-score']
        }
        scores.append(score)

    # Lets record this models average score
    return {
        'Average_R2': mean([x['R2'] for x in scores]),
        'Average_accuracy' : mean([x['Accuracy'] for x in scores]),
        'Average_precision' : mean([x['Average_precision'] for x in scores]),
        'Average_recall' : mean([x['Average_recall'] for x in scores]),
        'Average_f1' : mean([x['Average_f1'] for x in scores])
    }
```

```
# Create a record to keep all models scores for evaluating at the end
scores = pd.DataFrame()
```

Baseline : Naive Bayes

Naive Bayes will be our baseline classification model as this is a simple and intuitive method that requires just a few lines of code, and few tunable parameters.

We will be representing the textual data as a 'TF-IDF' (Term frequency and Inverse document frequency), the method which we will use to represent 'text' numerically will be with a term-document matrix, by counting the amount of times a term is represented in an email / record as well as calculating its weight amongst all emails we can do an analysis with these values. We will be using the tool TfidfVectorizer to do this.

```
# Lets make our bayes model
bayes_model = make_pipeline(TfidfVectorizer(), MultinomialNB())
```

```
# Lets train and test our model
bayes_average_score = GetAverageScores(data_sets, bayes_model)
print(bayes_average_score)
bayes_average_score['name'] = 'Bayes'
scores = scores.append(bayes_average_score, ignore_index=True)
```

```
{'Average_R2': 0.9561674318332903, 'Average_accuracy': 0.989051094890511,
'Average_precision': 0.989221462170163, 'Average_recall': 0.9889271565809679,
'Average_f1': 0.9890385809223926}
```

From the above results of our baseline model, these aren't bad scores at all, and could possibly be used for our final model. Though the problem with this model is that, words out of the vocabulary aren't taken into consideration, so doesn't handle new words very well. It also doesn't consider ordering of words, so estimating the semantics of the email is difficult.

Because of this we will develop another model to see if we can achieve better and more consistent results.

Support Vector Machine

Here we will create a SVM model, and compare its scores to our baseline.

As a SVM model is more complex than the bayes, and involves many more hyper-parameters, this will allow us to fine tune our model to get the best possible results. To make finding the most optimal hyper-parameters easier, we will be utilizing a tool called 'grid search' which will run all permutations of the parameters and tell us which would be the best to use.

Pre-processing

As we are attempting to beat our baseline performance, we will be trying some further processing of our original dataset, we will attempt the following:

- Lemmatize the dataset.
- Removing stop words

Depending on the performance of the model, we will decide if we keep the data processed in this way.

We will be representing the textual data as a 'TF-IDF' again in this model, although 'TF-IDF' is based on bag-of-words model which doesn't record text position and semantics, as our baseline had such high scoring, so we will be using this part of it.

Lets determine what the best parameters would be for our SVM, using grid search.

```
# Defining parameter range for grid search
param_grid = {'C': [0.1, 1, 10, 100, 1000],
              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['rbf', 'linear']}

# Lets make a svm model with grid search
svm_model_grid = GridSearchCV(SVC(), param_grid, refit = True, verbose = 3)
```

```
# Let's create our text processor
tfidf_vectorizer = TfidfVectorizer()
X_training = tfidf_vectorizer.fit_transform(data_sets[0][0])

# Lets fit the svm model with one of the sets to determine what would be the best
        parameters
svm_model_grid.fit(X_training,data_sets[0][2])
```

```
print(svm_model_grid.best_params_)
```

```
{'C': 10, 'gamma': 1, 'kernel': 'linear'}
```

From the above we can know the best parameters to use for our SVM.

```
# Lets make our svm model with those parameters
svm_model = make_pipeline(TfidfVectorizer(), SVC(C=10, gamma=1, kernel='linear'))
```

```
# Lets train and test our model
svm_average_score = GetAverageScores(data_sets, svm_model)
svm_average_score['name'] = 'SVM_non_processed'
scores = scores.append(svm_average_score, ignore_index=True)
```

```
# Lets lemmatize the data
data_sets_lemmatized = data_sets.copy()
lemmatizer = WordNetLemmatizer()
for index in range(0,len(data_sets_lemmatized)):
    data_sets_lemmatized[index][0] = data_sets_lemmatized[index][0].apply(lambda x:
        ' '.join([lemmatizer.lemmatize(w) for w in x.split()]))
```

```
# Lets train and test our model with the lemmatized data
svm_average_score = GetAverageScores(data_sets_lemmatized, svm_model)
svm_average_score['name'] = 'SVM_lemmatized'
scores = scores.append(svm_average_score, ignore_index=True)
```

```
# Lets remove stopwords
stop_words = stopwords.words('english')
data_sets_no_stopwords = data_sets.copy()
for index in range(0,len(data_sets_no_stopwords)):
    data_sets_no_stopwords[index][0] = data_sets_no_stopwords[index]
        [0].apply(lambda x: ' '.join([word for word in x.split() if word not in
        (stop_words)]))
```

```
# Lets train and test our model with the no stopword data
svm_average_score = GetAverageScores(data_sets_no_stopwords, svm_model)
svm_average_score['name'] = 'SVM_no_stopwords'
scores = scores.append(svm_average_score, ignore_index=True)
```

Visualizing results

We have run an instance of our baseline classification model, as well as 3 different instances of a SVM classification model. The models we have scored are:

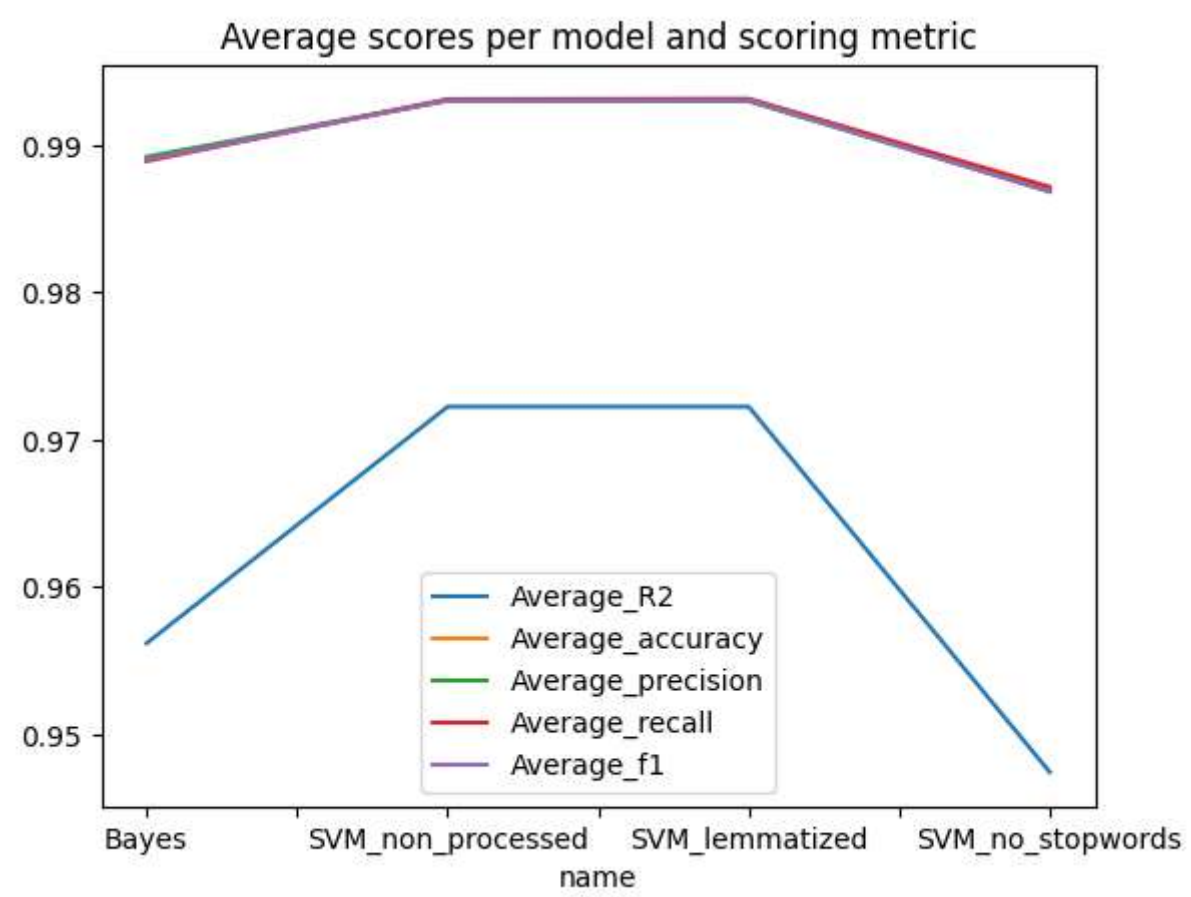
- Bayes : Being our baseline
- SVM_non_processed : Being our SVM model with not further processing
- SVM_lemmatized : Being our SVM model with lemmatized training data
- SVM_no_stopwords : Being our SVM with stopwords removed from the training data

```
# Let view the score records of the models
scores = scores.set_index('name')
scores
```

	Average_R2	Average_accuracy	Average_precision	Average_recall	Average_f1
name					
Bayes	0.956167	0.989051	0.989221	0.988927	0.989039
SVM_non_processed	0.972242	0.993066	0.993036	0.993134	0.993062
SVM_lemmatized	0.972243	0.993066	0.993000	0.993162	0.993062
SVM_no_stopwords	0.947408	0.986861	0.986890	0.987173	0.986859

```
# Lets plot the score records
scores.plot(title = 'Average scores per model and scoring metric')
```

```
<AxesSubplot: title={'center': 'Average scores per model and scoring metric'},
xlabel='name'>
```



Conclusion

Evaluation

According to our scoring records, referencing our baseline being Naive Bayes classification model as well as different permutations of an SVM classification model. We found that:

- Our SVM non processed model did slightly better than our baseline.
- Removing stop words from our training data, had no positive effect on our SVM model, and actually effected the scoring negatively.
- Lemmatizing the training data, didnt have much of an effect on the scoring.

There was a surprising factor in our results:

- Removing stopwords having a negative effect. This was strange as one would think this would fine tune our model to have a more accurate classification, but this was not the case.
  This could be because stop words actually play a role in identifying spam emails, it would be interesting to analyse the difference in stop words within spam and non-spam emails, though we will leave that for another time.

Summary

We have developed a email spam classifier with very good scores. Though because our classifier isn't 100% accurate, this means there is still going to be some spam slipping through as well as legitimate mails getting classified as spam. Because of this, people won't be secure from unsolicited mails and there will still always be a threat, though implementing this model within companies as well as general email server providers would be a start in solving our predefined problems in the domain.

What could be better?

The accuracy of our model, this possibly would not be the thing to improve on, as it is pretty high. I think using our highest SVM classifier again as a baseline, and try training different types of models and using different methods of representing text numerically, that focuses on semantics and grammar, would be the better way to go as ordering of words and semantics is obviously a huge part of a language, it would only make sense to make sure these are addressed in a classifier.

If I had more time with this project, I would analyse how other encoders such as 'word2Vec' do on our SVM classifier. I would also have concatenated more datasets of spam and non-spam mails, i felt my dataset got to small after balancing it.

References:

1. The Eleventh Sense, "HOW SPAM FILTERS WORK (AND HOW TO STOP EMAILS GOING TO SPAM)", Ivan LaBianca, 2022/03/4, https://www.theseventhsense.com/blog/how-spam-filters-work-and-how-to-stop-emails-going-to-spam
2. Statista, 2022/03/01, https://www.statista.com/statistics/420391/spam-email-traffic-share/
3. Kaggle, Venkatesh Garnepudi, 2018, https://www.kaggle.com/datasets/venky73/spam-mails-dataset
4. Velomethod, "Why Is Spam Filtering Not 100% Accurate?", Ruthie Toce, 2019/05/17, https://www.velomethod.com/post/why-is-spam-filtering-not-100-accurate
5. Data Pro, "What's On the Other Side of Your Inbox - 20 SPAM Statistics for 2022", 2022/07/20, Nikolina Cveticanin, https://dataprot.net/statistics/spam-statistics/
6. Apache SpamAssassin, website, https://spamassassin.apache.org/index.html