Building a supervised machine learning model to classify emails from the given Enron dataset

Student Name: Shanmugapriya Murugavel

Student No: R00195696

For the module COMP9060 – Applied Machine Learning as part of the Assignment 1

Master of Science in Data Science and Analytics, Department of Mathematics

DECLARATION:

I hereby certify that this material which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent, that such work has been cited and acknowledged within the text of my work.

I understand that my project documentation may be stored in the library at CIT and may be referenced by others in the future.

OBJECTIVE:

The main objective of this assignment is to **build a supervised machine learning model to classify emails as spam or non-spam** that are present in the given dataset. The dataset provided is the **Enron email dataset** which is a collection of public domain emails from the Enron corporation. To start with, one must understand what a machine learning algorithm is, its types and how they are can be used to solve real life problems. Here in the **given dataset, emails have been manually classified as spam and non-spam** (ham – as referred in this work).

OVERVIEW:

Pre-processing is initially carried out where the emails from the dataset are **cleaned to get rid of** the **special characters**, numerical values, single letters, **empty and duplicate** mails. Then the mails are converted to a format that the ML model can interpret. Further, **an exploratory data analysis is carried** out to educe insights to better understand the input data. Followed by this, a supervised classification model is designed to categorise the emails are spam or non-spam. **Several models can be used to compare the accuracy** of the model. In the end, all these models can be tested and criticised.

LOADING THE DATASETS:

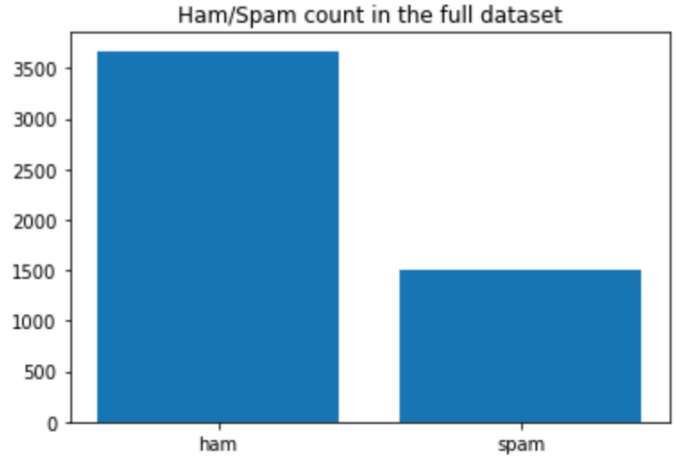
Download the dataset and load the ham and spam emails into a dataframe in Python. The below snaps from the Jupyter notebook.

```
#loading files
from sklearn.datasets import load_files

# creating lists to store the data and the target
X, Y = [], []
emails = load_files("/Users/shanmugapriyamurugavel/Documents/Sem 2/AML/Assignment/enron1",encoding='latin1')
X = np.append(X, emails.data)
Y = np.append(Y, emails.target)

classes = emails.target_names
print("The dataset enron1 has been loaded.")
print(f"The available classes are {classes[0]} and {classes[1]}.")

# printng size of docs and labels
print("The number of documents and labels:")
print(f"X.shape: {X.shape:}")
print(f"Y.shape: {Y.shape}")
```

```
The dataset enron1 has been loaded.
The available classes are ham and spam.
The number of documents and labels:
X.shape: (5172,)
Y.shape: (5172,)
```

Checking the composition of the dataset. The dataset is plotted against a bar chart to display its count of ham and spam mails.

```
plotHamSpam(y)
```



PRE-PROCESSING:

A function is defined to clean the strings passed, by removing the special characters, newline characters and numbers. It also removes single character words and the word ‘subject’ which is commonly occurring. The function also converts the whole string into lowercase and returns the final cleaned string.

```
def clean_string(the_string):
    '''
    The function cleans the string passed by removing special characters, newline characters and numbers.
    It also removes single character words and converts the whole string into lower case and removes some user requi
    Returns the final cleaned string
    '''
    #remove special characters
    special_removed_string = re.sub(r"[\;\?\<\>\,\"\|\:\.\`\~\{\}\\\/@$\%\[\]\(\)\\'\^\-\+\&#\*\!\_\=]*","", the_str

    #remove newline characters
    special_removed_string = re.sub(r"\r\n"," ",special_removed_string)

    #lowercasing the string
    lowered_string = special_removed_string.lower()

    #remove number and single character words
    number_removed_string = re.sub(r"[0-9]|\b[a-z]\b"," ",lowered_string)

    #substitute multiple spaces with single space
    space_removed_string = re.sub(r'\s+', ' ', number_removed_string)

    #removing the words subject and etc as they are most commonly occuring and subject is a part of every mail
    cleaned_string = re.sub(r'\bsubject\b|\bect\b'," ",space_removed_string)
    # the_clean_string = re.sub(r"[@\/\"$%#\[\]{}\^?&\.:!-_\d\{\}]", "", the_string)

    return cleaned_string
```

The cleaned string is stored as a list format and can be used in further processing. Displaying the cleansed string to check the difference before and after cleaning.

```
#clean and store the dataset in X_cleaned
X_cleaned = []
print("Sample email before cleaning:\n-------------------------------------------------------------------------------")
print(X[5])
for mail in X:
    X_cleaned.append(clean_string(mail))
X_cleaned = np.array(X_cleaned)
print("Sample email after cleaning:\n-------------------------------------------------------------------------------")
print(X_cleaned[5])
```

```
Sample email before cleaning:
-------------------------------------------------------------------------------
Subject: and the final numbers for may are . . .
iferc 1 , 240 , 000 ( last volume was 72084 on day 18 )
enron 930 , 000 ( last volume was 21667 on day 26 )
gas daily 1 , 033 , 416 ( last volume was 80000 on day 31 )
please advise ,
ami
Sample email after cleaning:
-------------------------------------------------------------------------------
  and the final numbers for may are iferc last volume was on day enron last volume was on day gas daily last volume
was on day please advise ami
```

The dataset has been cleaned to remove unwanted characters and words that will not be useful in building the model. **Cleaning the dataset helps to remove the feature size** by neglecting unwanted characters.

TRAINING AND TEST SPLITS

Splitting into training and test data:

Shuffle the dataset before splitting the test and train data. This can be done using the sample command to ensure the split is not bias. According to the given data (from question), a split is made on **70% of the data for training** the model and the rest **30% is saved for testing**.

The '**random_state**' is useful in ensuring there is a consistent set of samples chosen every time the split is made. And the **stratify ensures there is same ratio of ham and spam emails** in both the training and test dataset.

Splitting cleaned dataset into training and testing data

```
# Split the input dataset into 70% training set and the remaining into 30% test set
# To ensure such that test set is not biased in any way(sample bias) - shuffle dataset

X_train, X_test, Y_train, Y_test = train_test_split(X_cleaned, Y, test_size=0.3, random_state=42, stratify = Y)

# random_state -> is used to ensure that a consistent result of the splits is obtained each time the model runs
# stratify -> ensures to have equal samples of data in our test and train so the splits are not biased

print("Before Splitting:")
print("X Shape:",X.shape)
print("Y Shape:",Y.shape)

print("\nAfter Splitting:")
print("X_train Shape:",X_train.shape)
print("X_test  Shape:",X_test.shape)
print("Y_train Shape:",Y_train.shape)
print("Y_test  Shape:",Y_test.shape)
```

```
Before Splitting:
X Shape: (5172,)
Y Shape: (5172,)

After Splitting:
X_train Shape: (3620,)
X_test  Shape: (1552,)
Y_train Shape: (3620,)
Y_test  Shape: (1552,)
```

Storing the resulting datasets in files:

The resulting training and testing datasets are stored in a **.csv file** format for future usage.

Storing the resulting training and testing datasets in files

```
#store train and test in csv files with ham-0 and spam-1
train_df = pd.DataFrame({'email': X_train, 'class': Y_train})
test_df = pd.DataFrame({'email': X_test, 'class': Y_test})
print("Training set:\n--------------------------------------------------------------------------------\n",train_df.head(5))
print("\nTesting set:\n--------------------------------------------------------------------------------\n",test_df.head(5))

#storing into separate files
train_df.to_csv("email_train.csv",index=False)
test_df.to_csv("email_test.csv",index=False)
print("\nTraining and test data are stored in separate csv files...")
```

```
Training set:
--------------------------------------------------------------------------------
                                                email  class
0    gasper rice resources ltd vance sitara deal ...    0.0
1    meter scherlyn per our conversation here is ...    0.0
2    hi paliourg all available meds everything fo...    1.0
3    dec daren meter flowed dth on dth on and dth...    0.0
4    hot jobs global marketing specialties po box...    1.0

Testing set:
--------------------------------------------------------------------------------
                                                email  class
0    re new turn ons the deals are all in meter i...    0.0
1    prom dress shopping please respond to hi jus...    0.0
2    big cowboy gepl actuals for february thu we ...    0.0
3    no more doctors visits stop wasting money on...    1.0
4    kinder morgan cannon sale change fyi please ...    0.0

Training and test data are stored in separate csv files...
```

Recording statistics on the datasets:

Displaying total number of ham and spam emails in each set. The **'value.counts( )'** function helps in displaying the count.

```
# Display the number of ham and spam emails in the training dataset
train_count = train_df['class'].value_counts()
print ("Total number of Ham and Spam emails in the training dataset")
print(train_count)
```

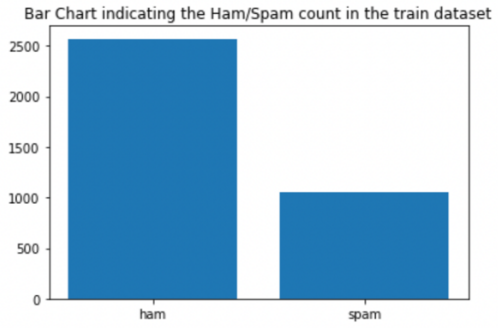
```
Total number of Ham and Spam emails in the training dataset
0.0    2582
1.0    1038
Name: class, dtype: int64
```

```
# Display the number of ham and spam emails in the test dataset
test_count = test_df['class'].value_counts()
print ("Total number of Ham and Spam emails in the test dataset")
print(test_count)
```

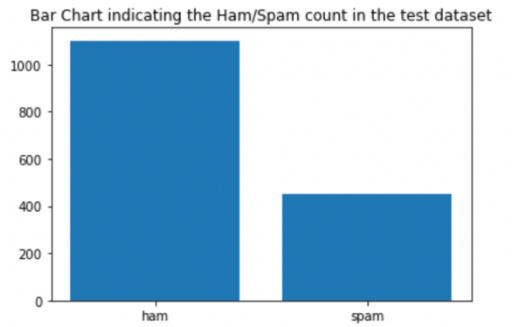
```
Total number of Ham and Spam emails in the test dataset
0.0    1090
1.0     462
Name: class, dtype: int64
```

A bar chart is used as shown below to visualise the count of ham and spam emails in both our test and training datasets.

```
plotHamSpam(Y_train,"train")
```




```
plotHamSpam(Y_test,"test")
```



FEATURE EXTRACTION

Extracting features from the dataset is crucial for training the machine learning model. Some of the techniques include **lemmatization, stemming, tokenization and sentence segmentation, etc.** These techniques help in training the model in an efficient manner [1].

Stop-words removal:

Certain words are normally filtered before processing the text, these **contains low level information**. In order to focus more on the most important information and **reduce the dataset size and training time** they are removed during text pre-processing. The **Natural Language Toolkit (NLTK)** package provides suitable libraries and programs to carry out this process.

```
#stop words from nltk(Natural Language Toolkit) and sklearn packages
nltk_stopwords = set(stopwords.words('english'))
skl_stopwords =ENGLISH_STOP_WORDS

#set to contain all the stop words
allStopWords = set()

#combining stop words from both libraries by union
allStopWords |= nltk_stopwords
allStopWords |= skl_stopwords
print("Total stop words in all 'StopWords': ",len(allStopWords))
print("\nSample stop words:",random.sample(allStopWords,10))
```

```
Total stop words in all 'StopWords':  378

Sample stop words: ['nothing', 'its', 'being', 'find', 'the', 'below', 'amongst', 'myself', 'of', 'him']
```

Bag of Words:

Bag of words creation is the most used method of feature extraction. Here the number of occurrences of the word or the order of the words doesn't really matter. It simply forms a collection of words that are present.

Bag of words creation

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
#ref: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html
#TF-IDF: term-frequency times inverse document-frequency

#count vector for whole datasaet
count_vector = CountVectorizer(stop_words=allStopWords)

#separate count vectors are generated to make visualizations and understand the common words
#count vector for ham mails
count_vector_ham = CountVectorizer(stop_words=allStopWords)
#count vector for spam mails
count_vector_spam = CountVectorizer(stop_words=allStopWords)
```

```
#extracting features
#tfidfTransformer converts a collection of raw documents(input email words) to a matrix of TF-IDF features

X_train_fit = count_vector.fit(train_df['email'])
X_train_bagofwords = X_train_fit.transform(train_df['email'])
tf_transformer = TfidfTransformer(use_idf=False).fit(X_train_bagofwords)
X_train_tf = tf_transformer.transform(X_train_bagofwords)

#bag of words for ham mails
X_trainham_fit = count_vector_ham.fit(train_df[train_df['class']==0]['email'])
X_trainham_bagofwords = X_trainham_fit.transform(train_df[train_df['class']==0]['email'])

#bag of words for spam mails
X_trainspam_fit = count_vector_spam.fit(train_df[train_df['class']==1]['email'])
X_trainspam_bagofwords = X_trainspam_fit.transform(train_df[train_df['class']==1]['email'])
```

The above lines of codes represent the bag of words creation and the TF-IDF process.

Count Vectorizer: Converts a collection of text documents to a matrix of token counts [3].
TF-IDF: Term Frequency Inverse Document Frequency: To solve the problem of higher frequency words being dominant, its frequency is rescaled by considering how frequent it occurs in all documents.

- Term Frequency (TF) gives the frequency of the word in current document.
- Inverse Document Frequency (IDF)

$$TFIDF\ score\ for\ term\ i\ in\ document\ j = TF(i,j) * IDF(i)$$

where

$IDF = Inverse\ Document\ Frequency$

$TF = Term\ Frequency$

$$TF(i,j) = \frac{\text{Term i frequency in document j}}{\text{Total words in document j}}$$

$$IDF(i) = \log_2 \left(\frac{\text{Total documents}}{\text{documents with term i}}\right)$$

and

$t = Term$

$j = Document$

The above equations helps to calculate the TF and IDF values. As seen in the code, the TF-IDF method is used to extract the features. The vector stores the non-zero elements in a sparse matrix.

Tokenization:

The process where the input email is broken into pieces of texts or tokens so that in can be used to build the model. The tokenizer then created them into vocabularies.

```
# Tokenization
import re

def word_tokenize(words):
    tokens = re.split('\W+',words)
    return tokens


train_df['Tokenized_email'] = train_df['email'].apply(lambda x: word_tokenize(x.lower()))

train_df.head()
```

	email	class	X_tokenized
0	gasper rice resources ltd vance sitara deal ...	0.0	[, gasper, rice, resources, ltd, vance, sitara...
1	meter scherlyn per our conversation here is ...	0.0	[, meter, scherlyn, per, our, conversation, he...
2	hi paliourg all available meds everything fo...	1.0	[, hi, paliourg, all, available, meds, everyth...
3	dec daren meter flowed dth on dth on and dth...	0.0	[, dec, daren, meter, flowed, dth, on, dth, on...
4	hot jobs global marketing specialties po box...	1.0	[, hot, jobs, global, marketing, specialties, ...

From the above code and output, it is seen that the tokenizer function has split our input email text into individual tokens. These are then stored in a list, separated by comma.

Stemming:

During the process of stemming, the tokens are reduced into their stem words (base form or root form). For example: plays, played, playing – play. All the word forms of play are considered the same word.

```
# Stemming

ps = nltk.PorterStemmer()

def word_stemming(stemmed_word):
    text = [ps.stem(word) for word in stemmed_word]
    return text

train_df['emails_stemmed'] = train_df['X_tokenized'].apply(lambda x: word_stemming(x))

train_df.head()
```

	email	class	X_tokenized	emails_stemmed
0	gasper rice resources ltd vance sitara deal ...	0.0	[, gasper, rice, resources, ltd, vance, sitara...	[, gasper, rice, resourc, ltd, vanc, sitara, d...
1	meter scherlyn per our conversation here is ...	0.0	[, meter, scherlyn, per, our, conversation, he...	[, meter, scherlyn, per, our, convers, here, i...
2	hi paliourg all available meds everything fo...	1.0	[, hi, paliourg, all, available, meds, everyth...	[, hi, paliourg, all, avail, med, everyth, for...
3	dec daren meter flowed dth on dth on and dth...	0.0	[, dec, daren, meter, flowed, dth, on, dth, on...	[, dec, daren, meter, flow, dth, on, dth, on, ...
4	hot jobs global marketing specialties po box...	1.0	[, hot, jobs, global, marketing, specialties, ...	[, hot, job, global, market, specialti, po, bo...

Here the inbuilt function **PorterStemmer()** from NLTK package is being used. It can be noticed that the **token 'flowed' is converted into its word stem 'flow'**.

EXPLORATORY DATA ANALYSIS

In this section, the training dataset of the model is taken to perform some data analysis for gaining useful insights from it. Visualisations are carried out to better understand the training data and its composition. Various forms of analysis were carried out as follows:

- Word cloud of Non-Spam and Spam emails by masking up 'positive' and 'negative' symbols respectively.
- Word Cloud of top 100 words from both the email sets.
- Bar plot representing the top 20 words from Non-Spam and Spam emails.
- Word Cloud to compare the top 20 words from bar plot.

Word Cloud of Non-Spam emails:

The below lines of code were generated to visualise a word cloud of the most frequent words from the non-spam emails. Since the **non-spam indicates positivity** the word cloud is masked into the '**Plus +' image**.

Exploring the dataset - few visuals to understand

```
# wordcloud styling using a mask image
# indicating words from non spam emails are put into a plus icon

from PIL import Image
green_plus = '/Users/shanmugapriyamurugavel/Documents/Sem 2/AML/Assignment/plus.png'
icon = Image.open(green_plus)
image_mask = Image.new(mode='RGB', size=icon.size, color=(255, 255, 255))
image_mask.paste(icon, box=icon)

# converting the image file into an array
rgb_array = np.array(image_mask)

# Generating the text as a string for using it into the word cloud
spam_words = ''.join(train_df[train_df['class']==0]['email'])

word_cloud = WordCloud(mask=rgb_array, background_color='white', max_font_size=400,
                       max_words=1500, colormap='summer')

# Displaying the words in upper cases for better visibility
word_cloud.generate(spam_words.upper())

plt.figure(figsize=[16, 12])
plt.imshow(word_cloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

Here the **training dataset used before performing the feature extraction is used**. In these non-spam emails as there is positivity, the colour of the tone is set to '**Summer**' palette. The word cloud is given as follows for the non-spam emails.

Word Cloud for Spam emails:

The below lines of code were generated to visualise a word cloud of the most frequent words from the spam emails. Since the **spam indicates negativity** the word cloud is masked into the '**Minus –' image**.

In the spam emails as there is **negativity**, the colour of the tone is set to '**Autumn**' palette.

The below lines of codes help us to plot the word cloud shown above.

```
# wordcloud styling using a mask image
# indicating the common words of the spam mails in a minus icon

red_minus = '/Users/shanmugapriyamurugavel/Documents/Sem 2/AML/Assignment/minus.png'
icon = Image.open(red_minus)
image_mask = Image.new(mode='RGB', size=icon.size, color=(255, 255, 255))
image_mask.paste(icon, box=icon)

# converting the image file into an array
rgb_array = np.array(image_mask)

# Generating the text as a string for the word cloud
spam_words = ''.join(train_df[train_df['class']==0]['email'])

word_cloud = WordCloud(mask=rgb_array, background_color='white', max_font_size=350,
                       max_words=2500, colormap='autumn')

# Display the words in upper cases for better visibility
word_cloud.generate(spam_words.upper())

plt.figure(figsize=[16, 12])
plt.imshow(word_cloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

Filtering top 100 words from both mails:

There is no clear picture of the top words which are more frequent in these mails. To find these, a function is created to filter the top ‘n’ number of words. This is then displayed in a word cloud. Let’s consider n as 100 and generate the word clouds.

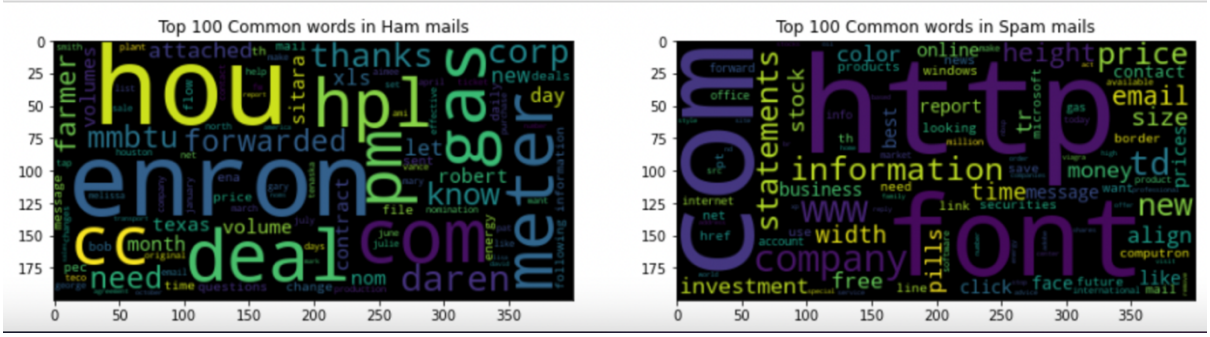
Here the **new training set is used** which is obtained **after performing the feature extraction**. This is mainly done to **understand the importance of various feature extraction processes** used in building our model. By eliminating those stop words and infrequent words the actual content is recognised.

The below lines of codes, plots the word cloud for the top 100 words in both ham and spam emails. These plots are placed side-by-side to draw a comparison between them.

Displaying top 100 words

```
#Get top n words
def topNWords(bag_of_words,vec, n=None):
    '''
    returns top n words with its frequency'''

    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx in     vec.vocabulary_.items()]
    words_freq =sorted(words_freq, key = lambda x: x[1], reverse=True)
    return words_freq[:n]
```

```
#top 100 words in ham and spam emails
topham = topNWords(X_trainham_bagofwords,X_trainham_fit,100)
topspam = topNWords(X_trainspam_bagofwords,X_trainspam_fit,100)
fig = plt.figure(figsize=(15,15))
ax1 = fig.add_subplot(1,2,1)
wordcloud = WordCloud().generate_from_frequencies(dict(topham))
ax1.imshow(wordcloud)
ax1.title.set_text('Top 100 Common words in Ham mails')

ax2 = fig.add_subplot(1,2,2)
wordcloud = WordCloud().generate_from_frequencies(dict(topspam))
ax2.imshow(wordcloud)
ax2.title.set_text('Top 100 Common words in Spam mails')
```



From the second set of word clouds (above image) the actual words are obtained rather than the stop words.

Bar plot to show the relative frequencies of top 20 words in ham emails:

The below lines of codes are used to plot the relative frequencies of the top 20 words which are being used in the ham emails. Here a list is first created to store these words and then the list is used to generate the bar plot. Finally, the image is saved using the savefig command for importing it into to the report.

To verify the results against the bar plot, the same frequency is set as 20 for the above word cloud.

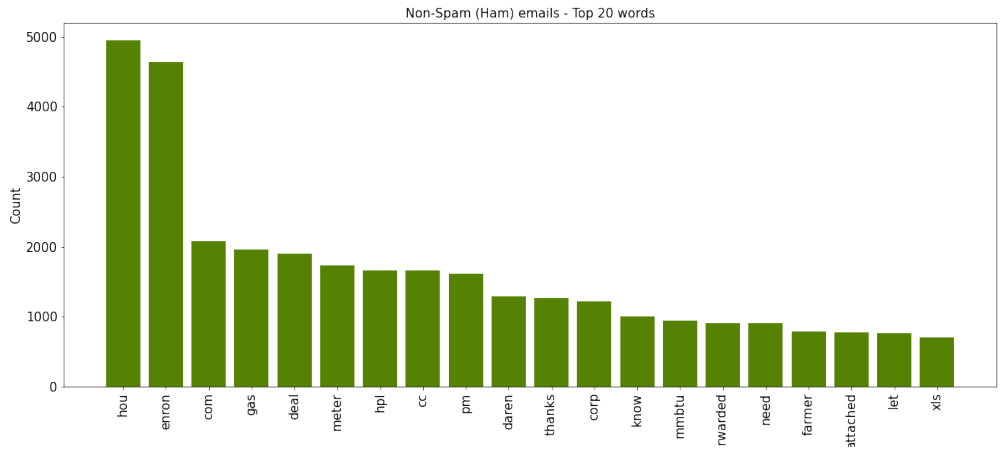
Barplot for the top 20 words in both ham and spam emails

```
# creating 2 new lists to store the top 20 frequently used words in each set

topham = topNWords(X_trainham_bagofwords,X_trainham_fit,20)
topspam = topNWords(X_trainspam_bagofwords,X_trainspam_fit,20)
#print(topham)
#print(type(topham))
```

Bar plot to show the relative frequencies of top 20 words in Non-spam emails

```
import numpy as np
import matplotlib.pyplot as plt

# barplot showing the relative frequencies of the most frequently used words in ham emails
ham_label, ham_y = zip(*topham)
ham_x = np.arange(len(ham_label))
width = 10
fig, (bar1) = plt.subplots(1,1,figsize = (20, 8))
plt.bar(xs, ham_y, width=0.8, align='center',color=['#568203'])
plt.xticks(ham_x, ham_label,rotation='vertical')
plt.yticks()
bar1.set_ylabel('Count', fontsize = 15)
bar1.set_xlabel('Top 20 frequently used words',fontsize = 15)
bar1.tick_params(labelsize=15)
bar1.set_title('Non-Spam (Ham) emails - Top 20 words', fontsize = 15)
plt.savefig('nonspam_bar.png')
```

The bar plot is given as below.



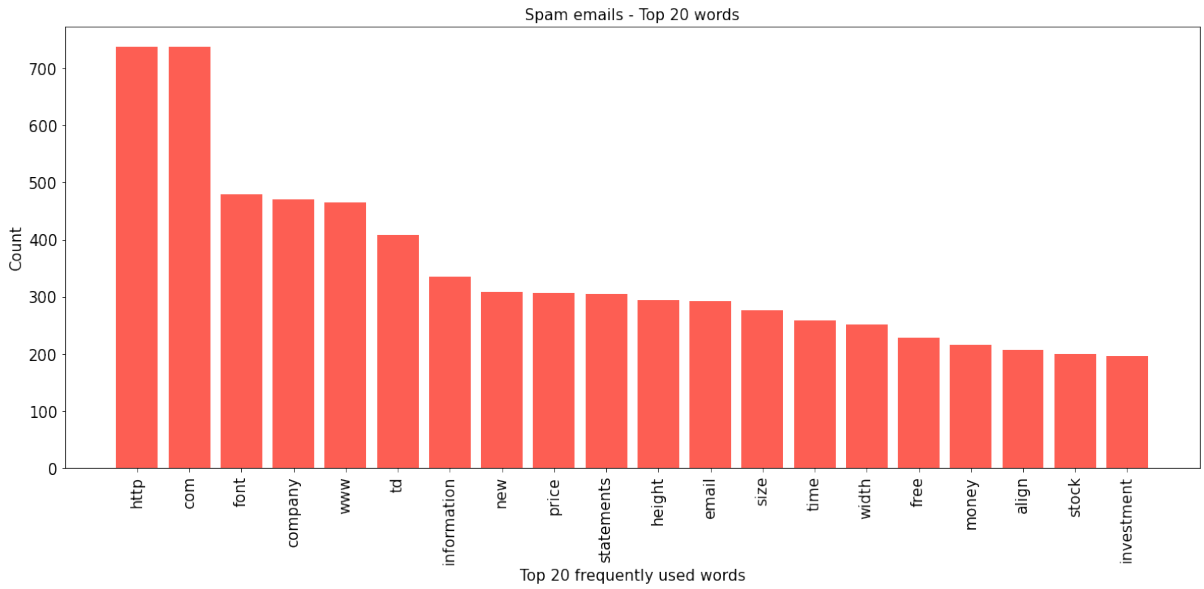
From the above bar plot, it is inferred that the words ‘hou’, ‘enron’, ‘com’, ‘gas’ are few of the most frequently used words from the non-spam set of emails given.

Bar plot to show the relative frequencies of top 20 words in spam emails:

The below lines of codes are used to plot the relative frequencies of the top 20 words which are being used in the spam emails. The below lines of codes are used for generating this plot.

Bar plot to show the relative frequencies of top 20 words in spam emails

```
# barplot showing the relative frequencies of the most frequently used words in spam emails
spam_label, spam_y = zip(*topspam)
spam_x = np.arange(len(spam_label))
width = 10
fig, (bar1) = plt.subplots(1,1,figsize = (20, 8))
plt.bar(spam_x, spam_y, width=0.8, align='center',color=['#fd5e53'])
plt.xticks(spam_x, spam_label,rotation='vertical')
plt.yticks()
bar1.set_ylabel('Count', fontsize = 15)
bar1.set_xlabel('Top 20 frequently used words',fontsize = 15)
bar1.tick_params(labelsize=15)
bar1.set_title('Spam emails - Top 20 words', fontsize = 15)
plt.savefig('spam_bar.png')
```



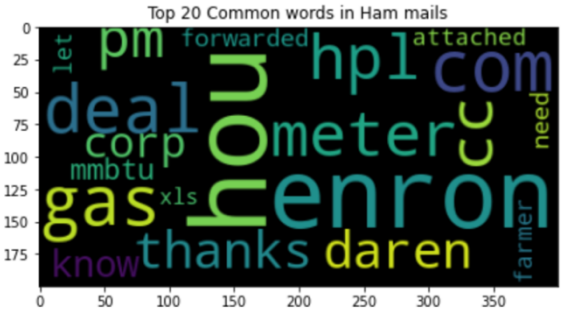
From the above bar plot, the words ‘http’, ‘com’, ‘font’ and ‘company’ are among the most frequently used words in the spam emails given.

Taking top 20 words in word cloud to compare the results of bar plot:

To compare the results from bar plot, a word cloud is also plotted to verify the top 20 words from both the email training datasets. The below lines of codes are used to plot it.

```
#top 20 words in ham and spam emails
topham = topNWords(X_trainham_bagofwords,X_trainham_fit,20)
topspam = topNWords(X_trainspam_bagofwords,X_trainspam_fit,20)
fig = plt.figure(figsize=(15,15))
ax1 = fig.add_subplot(1,2,1)
wordcloud = WordCloud().generate_from_frequencies(dict(topham))
ax1.imshow(wordcloud)
ax1.title.set_text('Top 20 Common words in Ham mails')

ax2 = fig.add_subplot(1,2,2)
wordcloud = WordCloud().generate_from_frequencies(dict(topspam))
ax2.imshow(wordcloud)
ax2.title.set_text('Top 20 Common words in Spam mails')
```

The word cloud is given as follows.



Box plots to compare the email lengths:

The below lines of code are used to create the box plot to compare the lengths of ham and spam training dataset emails. Here a function is defined to calculate the length.

Boxplots to compare the email lengths

```
def emailLengthPlot(data,cls):
    '''
    Function to plot a boxplot for email length distribution'''
    mail_len = []
    for email in data :
        mail_len.append(len(email))

    fig = plt.figure(figsize =(10, 7))

    # Creating plot
    plt.boxplot(mail_len)
    plt.title(f"Mail length distribution of {classes[cls]} mails")

    # show plot
    plt.show()
```

```
print("Length distribution for ham mails")
emailLengthPlot(train_df[train_df['class']==0]['email'],0)
```

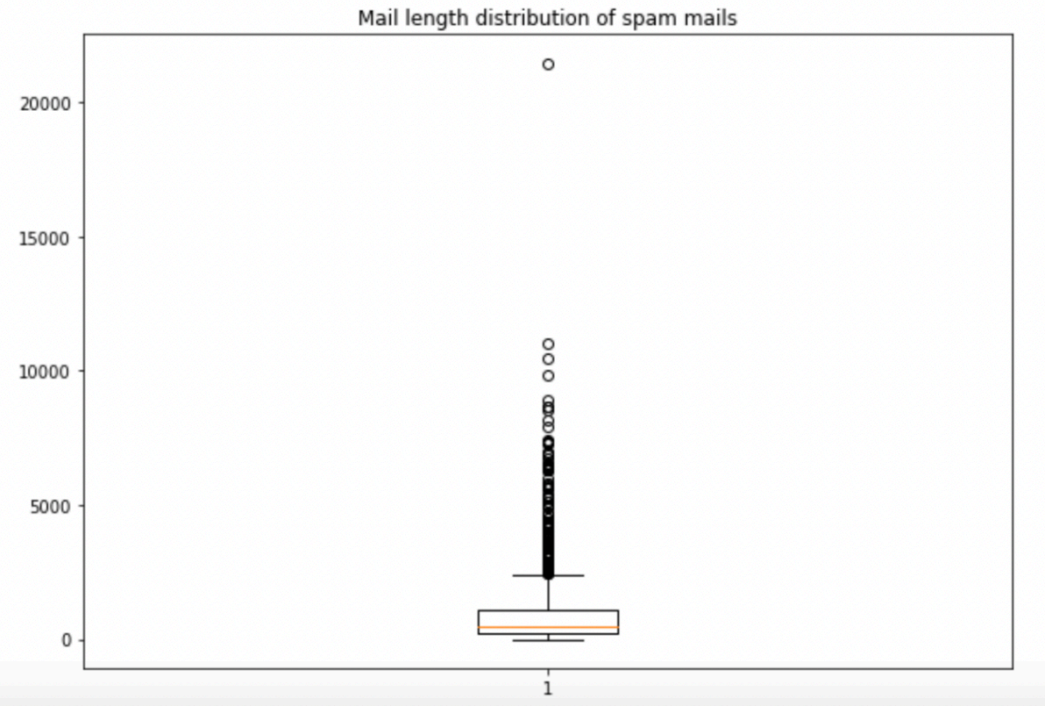
For Ham emails:

```
Length distribution for ham mails
```

For Spam emails:

```
print("Length distribution for spam mails")
emailLengthPlot(train_df[train_df['class']==1]['email'],1)
```

```
Length distribution for spam mails
```



These are few highlights observed from the above box plots:

- The lengths of the emails are almost similar for 75% of the emails.
- Both the box plots are skewed to the right.
- Both the box plots have several outliers where the email lengths are to be very long than usual.

The above plots show that the **distribution of length of the emails in both classes (ham/spam) are similar**.

SUPERVISED CLASSIFICATION

In this section, several classification models are chosen and trained using the training dataset which was split from the give Enron data. Later the model is tested using the unseen test dataset. The performance of these models will be compared based on these categories: Accuracy, Confusion matrix, Precision, F1 score and recall or sensitivity. Based upon those results the best performing model is finalised for classifying the spam and ham emails.

- **Accuracy** – Accuracy is given by the ratio of number of correct predictions to the total number of input samples.
- **Confusion Matrix** – The confusion matrix gives the output in the form of a matrix and describes the complete performance of the designed model. Here there are 4 important terms which is given as follows:
    - **True Positives (TP)**: Predicted value is YES, and the actual output is also YES.
    - **True Negatives (TN):** Predicted value is NO, and the actual output is NO.
    - **False Positives (FP):** Predicted value is YES, and the actual output is NO.
    - **False Negatives (FN):** Predicted value is NO, and the actual output is YES.
- **Precision** – Precision is the ratio of true positives divided by the sum of the true positives and the false positives.
- **F1 Score** – F1 score is given by the weighted harmonic mean of the precision and recall of the test.
- **Recall** – Also called the sensitivity, it is the ratio of the True positive divided by the sum of true positive and false negative.

In this Classification model, there are several classification models used namely Naïve Bayes model, Support Vector Machine, Decision Tree and K Nearest Neighbours algorithms.

Let’s check how each model is build up to classify our emails into ham and spam.

1. NAÏVE BAYES MODEL

Naïve Bayes is a supervised classification model which uses the principle of class conditional independence of the **Bayes Theorem**. It is mostly useful when dealing with large datasets. Here it assumes the presence of a specific feature in a class doesn’t impact the presence of any other features present in it [4]. The **probability of each category is calculated using Bayes theorem** and the output will be the one which has the highest probability.

The values of **accuracy, precision, recall and F1 score** is calculated, and **a confusion matrix** is displayed to r**epresent the counts of FP, FN, TP, TN.**

The below lines of codes were used to implement this algorithm.

Machine learning models

```
# In this classification problem, machine learning models like Naive Bayes,
# Support Vector Machine and Decision tree classifiers are used
```

```
# Final evaluation of models
from sklearn.metrics import precision_recall_fscore_support as score
import time
from sklearn.svm import SVC
import seaborn as sn
from sklearn.metrics import confusion_matrix as cm
from sklearn.metrics import classification_report as cr
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score, fit_time
```

i) Naive Bayes model

```
#Training the Naive Bayes model
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import MultinomialNB
nbmodel = MultinomialNB().fit(X_train_bagofwords, Y_train)

#Making predictions with test data
X_new_counts = count_vector.transform(X_test)
X_new_tfidf = tf_transformer.transform(X_new_counts)
predicted = nbmodel.predict(X_new_tfidf)

# Printing outputs of the models
# Accuracy, precision, recall and f1 score
print(f"Classification using the Naive Bayes Algorithm: ")
print(f"\n------Output of Naive Bayes using the Testing dataset-------")
print(f" ")
print(f" Accuracy   : {round(accuracy_score(Y_test,predicted)*100,2)}")
print(f" Precision  : {round(precision_score(Y_test,predicted),2)}")
print(f" Recall     : {round(recall_score(Y_test,predicted),2)}")
print(f" F1 score   : {round(f1_score(Y_test,predicted),2)}")


print("\nCrosstab of classfied results for Naive Bayes")
pd.crosstab(Y_test,predicted)
```

```
Classification using the Naive Bayes Algorithm:

------Output of Naive Bayes using the Testing dataset-------

 Accuracy   : 97.04
 Precision  : 0.98
 Recall     : 0.92
 F1 score   : 0.95

Crosstab of classfied results for Naive Bayes
```

col_0	0.0	1.0
row_0		
0.0	1094	8
1.0	38	412

The output obtained from the designed **Naïve Bayes Model** is displayed as above. It is inferred that, the **model correctly classified 1506 emails out of 1552 emails tested** which constitutes to 97% of accuracy.

2. SUPPORT VECTOR MACHINE

Support Vector Machines (SVM) is another popular supervised machine learning algorithm developed by Vladimir Vapnik. This model is also used for classification and regression model building. SVM models are **built on the idea of finding a hyperplane** called as the **decision boundary** that divides a given training dataset into two classes. This decision boundary **separates the datapoints** into either side of the classes.

The below lines of codes were used to implement this algorithm.

ii) Support Vector Machine

```
#Training the SVM model
from sklearn import svm
svmmodel = svm.SVC().fit(X_train_bagofwords,Y_train)

#Making predictions with test data
X_new_counts = count_vector.transform(X_test)
X_new_tfidf = tf_transformer.transform(X_new_counts)
predicted = svmmodel.predict(X_new_tfidf)

# Printing outputs of the models
# Accuracy, precision, recall and f1 score
print(f"Classification using the Support Vector Machines: ")
print(f"\n------Output of SVM using the Testing dataset-------")
print(f" ")
print(f" Accuracy   : {round(accuracy_score(Y_test,predicted)*100,2)}")
print(f" Precision  : {round(precision_score(Y_test,predicted),2)}")
print(f" Recall     : {round(recall_score(Y_test,predicted),2)}")
print(f" F1 score   : {round(f1_score(Y_test,predicted),2)}")


print("\nCrosstab of classfied results for Support Vector Machines")
pd.crosstab(Y_test,predicted)
```

```
Classification using the Support Vector Machines:

------Output of SVM using the Testing dataset-------

 Accuracy   : 84.79
 Precision  : 0.66
 Recall     : 1.0
 F1 score   : 0.79

Crosstab of classfied results for Support Vector Machines
```

col_0	0.0	1.0
row_0		
0.0	866	236
1.0	0	450

The output obtained from the designed **Support Vector Machine** Model is displayed as above. It is inferred from the confusion matrix that; **1316 emails were classified correctly out of 1552** emails tested having an **accuracy of 85%**.

3. DECISION TREE

Decision tree model is also one such classifier which can be used for both Classification and Regression. The model **predicts the value of a target test variable by framing decision rules** which are inferred from the training dataset. The **deeper the tree** is, the **more complex** the decision rules are and **fitter the model is** designed.

The below lines of codes were used to implement this algorithm.

iii) Decision Tree

```
from sklearn import tree

#Training the Decision tree
dtree = tree.DecisionTreeClassifier().fit(X_train_bagofwords,Y_train)


#Making predictions with test data
X_new_counts = count_vector.transform(X_test)
X_new_tfidf = tf_transformer.transform(X_new_counts)
predicted = dtree.predict(X_new_tfidf)


# Printing outputs of the models
# Accuracy, precision, recall and f1 score
print(f"Classification using the Decision Tree Model: ")
print(f"\n------Output of Decision Tree using the Testing dataset-------")
print(f" ")
print(f" Accuracy   : {round(accuracy_score(Y_test,predicted)*100,2)}")
print(f" Precision  : {round(precision_score(Y_test,predicted),2)}")
print(f" Recall     : {round(recall_score(Y_test,predicted),2)}")
print(f" F1 score   : {round(f1_score(Y_test,predicted),2)}")
print("\nCrosstab of classfied results for decision tree")
pd.crosstab(Y_test,predicted)
```

```
Classification using the Decision Tree Model:

------Output of Decision Tree using the Testing dataset-------

 Accuracy   : 38.53
 Precision  : 0.32
 Recall     : 1.0
 F1 score   : 0.49

Crosstab of classfied results for decision tree
```

col_0	0.0	1.0
row_0		
0.0	148	954
1.0	0	450

The output obtained from the designed Decision Tree Model is displayed as above. As inferred, decision tree model **delivers a poor performance in terms of classification**. The accuracy of the decision tree model is only **38.53** %.

4. K NEAREST NEIGHBOURS

The K-Nearest Neighbours algorithm is a method which can do both classification and regression. It **estimates the likelihood of one group member** based on what **group the nearest data belongs to**. KNN is also called **the lazy learning algorithm** as it doesn't train when the training data is supplied. Instead, it just **stores the data** and doesn't perform any calculations. It finds the **Euclidean distance** to all the training data samples and stores their values in an ordered list which is then sorted. It then **chooses the top K entries** from the sorted list. When the test data is tested it labels them based on most of the classes present in the selected points.

The below lines of codes were used to implement this algorithm.

iv) K Nearest Neighbours

```
from sklearn.neighbors import NearestNeighbors
from sklearn.neighbors import KNeighborsClassifier

#Training the K Neighbours
knn = KNeighborsClassifier()
knn.fit(X_train_bagofwords,Y_train)


#Making predictions with test data
X_new_counts = count_vector.transform(X_test)
#X_new_tfidf = tf_transformer.transform(X_new_counts)
predicted = knn.predict(X_new_counts)

# Printing outputs of the models
# Accuracy, precision, recall and f1 score
print(f"Classification using the KNN algorithm: ")
print(f"\n------Output of KNN using the Testing dataset-------")
print(f" ")
print(f" Accuracy   : {round(accuracy_score(Y_test,predicted)*100,2)}")
print(f" Precision  : {round(precision_score(Y_test,predicted),2)}")
print(f" Recall     : {round(recall_score(Y_test,predicted),2)}")
print(f" F1 score   : {round(f1_score(Y_test,predicted),2)}")
print("\nCrosstab of classfied results for K Neighbours")
pd.crosstab(Y_test,predicted)
```

```
Classification using the KNN algorithm:

------Output of KNN using the Testing dataset-------

 Accuracy   : 80.73
 Precision  : 0.61
 Recall     : 0.96
 F1 score   : 0.74

Crosstab of classfied results for K Neighbours
```

col_0	0.0	1.0
row_0		
0.0	821	281
1.0	18	432

The output obtained from K Nearest Neighbours is shown above. It is inferred from the confusion matrix that; **1253 emails out of 1552** tested emails **were classified correctly** having an **accuracy of 80%.**

CROSS VALIDATION OF MODELS

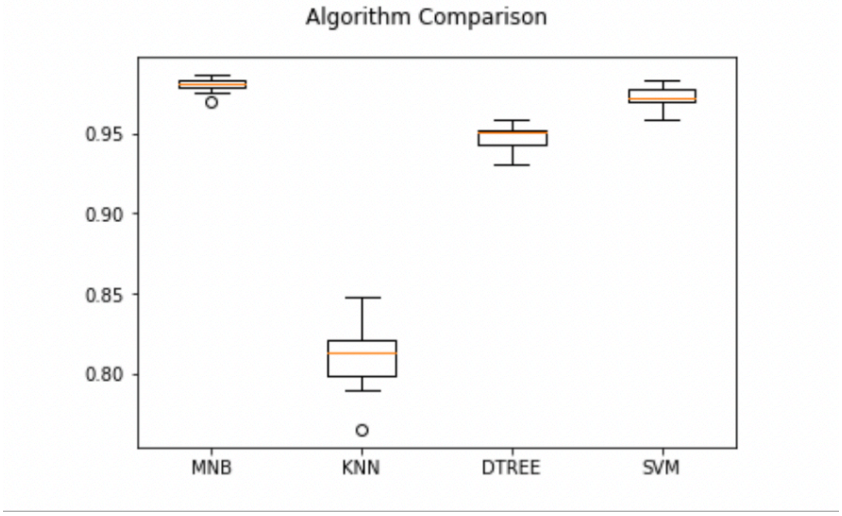
Having designed 4 various classification models, it is equally more **important to validate the stability of the designed models.** Cross-validation helps to overcome this requirement. There are several cross-validation techniques that can be carried out. Some of the common ones used are **K Fold cross validation, Stratified K-Fold Cross validation and Leave-P-Out Cross-validation.**

Here in the designed model cross-validation is carried out as follows:

```
from sklearn import model_selection
models = []
seed = 10
models.append(('MNB', MultinomialNB()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('DTREE', tree.DecisionTreeClassifier()))
models.append(('SVM', svm.SVC()))
# evaluate each model in turn
results = []
names = []
scoring = 'accuracy'
for name, model in models:
    kfold = model_selection.KFold(n_splits=10)
    cv_results = model_selection.cross_val_score(model, X_train_bagofwords, Y_train, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "Accuracy of %s: %f " % (name, cv_results.mean())
    print(msg)
# boxplot algorithm comparison
fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Then a box plot is plotted using the accuracy values to choose the best accurate model for classifying the emails.

```
Accuracy of MNB: 0.980387
Accuracy of KNN: 0.810497
Accuracy of DTREE: 0.947238
Accuracy of SVM: 0.972652
```



From the box plot, it can be concluded that Naïve Bayes algorithm best suited for this classification.

Both Support Vector and the Naïve Bayes are having highest accuracy values of 97%.

The Decision Tree and the KNN models have relatively lower accuracy rates.

SAVING THE BEST FIT MODEL

To save the best model that best fits our classification algorithm, the pickle function can be used. The code is given as follows:

```
# saving the best model using pickle function
```

```
import pickle
with open('model_pkl', 'wb') as files:
    pickle.dump(model, files)
```

REFERENCES:

[1] Fundamentals of NLP – Chapter 1
https://dair.ai/notebooks/nlp/2020/03/19/nlp_basics_tokenization_segmentation.html

[2] Python Graph Gallery - https://python-graph-gallery.com/barplot/

[3] Count Vectorizer, from Scikit-Learning: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

[4] Supervised Learning Algorithms: https://learn.g2.com/supervised-learning

[5] https://www.analyticsvidhya.com/blog/2021/08/quick-hacks-to-save-machine-learning-model-using-pickle-and-joblib/

[6] https://medium.com/syncedreview/applying-multinomial-naive-bayes-to-nlp-problems-a-practical-explanation-4f5271768ebf

[7] https://towardsdatascience.com/tf-idf-explained-and-python-sklearn-implementation-b020c5e83275