Foundations of Machine Learning
DA5400
ASSIGNMENT 2

SUBMITTED BY

GLEN PHILIP SEQUEIRA
DA24C005

Contents

1. Folder Structure and Instructions

As part of this assignment, we were asked to build a spam classifier from scratch. I have implemented the bernoulli Naive Bayes algorithm to implement the spam classifier. The folder named `solutions_da24c005` contains:

- `analysis.py`
- `analysis_test.csv`
- `helper_functions.py`
- `main.py`
- `naive_bayes.py`
- `train.csv`

1.1. Code.

- `analysis.py`: It contains the code to analyse the impact of data-preprocessing and feature-extraction on accuracy of the naive bayes classifier.

- `helper_functions.py`: It contains the code of several helper functions for the smooth functioning of `naive_bayes.py` and `analysis.py` files

- `main.py`: It contains the code to obtain predictions for a test folder (containing emails in .txt files). It returns `test_predictions.csv` file in `solutions_da24c005`, which contains the predictions for emails in the test folder.

- `naive_bayes.py`: It contains the code for implementation of the naive bayes alorithm from scratch

1.2. Datasets.

- `train.csv` is a dataset used for training our naive_bayes model. This dataset contains 2820 emails, with equal number of spam and ham emails. I have created this dataset using:
  (1) source 1
  (2) source 2
  (3) Emails generated by an LLM model

- I have used `analysis_test.csv` to test the model's performance for analysing the importance of data preprocessing and feature extraction. I have created this dataset using 5000 emails from source 3

1.3. Instructions.

- All the above files are saved in `solutions_da24c005` zip folder. After extracting the files, run python from the same directory/folder where all these files are located, otherwise it will give error.

- Kindly ensure that the following python libraries have been installed on your device:
  (1) `pandas` for data handling
  (2) `matplotlib.pyplot` for visualisation
  (3) `re` for text manipulation

(4) `CountVectorizer` from `sklearn.feature_extraction` for feature extraction `os` for file handling
(5) `numpy` for matrix operations

- Keep the `test` folder containing the .txt email files, in the same directory where these files are present.

- Run `analysis.py` to get analysis using visualisation of the impact of data processing and feature extraction on model performance.

- Run `main.py` to get predictions for `test` folder containing `.txt` files. This `main.py` file generates a `test_predictions.csv` file in the current directory. `main.py` also saves the array of predictions for the `test` emails in the variable `predictions`. This csv file contains three columns:
  (1) `email_name`: the `.txt` file name
  (2) `email`: the content of the corresponding `.txt` file
  (3) `Predictions`: predictions of our naive bayes classifier in 1 (spam), 0(ham) format

2. Naive Bayes Classifier

2.1. Introduction. For this spam classification problem, I have chosen naive bayes algorithm as my binary classifier. This is a good classifier because it is simple but efficient. By the class conditional independence assumption, naive bayes algorithm is computationally efficient as it reduces the number of parameters significantly by this assumption. Though this assumption is simplistic as it assumes that there is no relationship between the words appearing in an email, naive bayes performs relatively well and quickly in practice. I have not used SVM and logistic regression because these classifiers are complex, require relatively more time to predict and require hyperparameter tuning. KNN cannot be used for spam classification because calculation of euclidean distance in high dimension datasets like email datasets is not practical.

The Naive Bayes algorithm is a generative model. It models the joint probability:

$$P(x, y)$$

where:

- $x_i \in \{0, 1\}^d$ represents a data point (email), and each component in the vector corresponds to the presence (1) or absence (0) of a word.
- $y_i \in \{0, 1\}$ represents the class label: spam - 1, non-spam - 0.

Naive bayes models $P(x, y)$ by modelling the probabilty of a spam or non-spam email $P(y)$ and probability of the feature vector given a spam or non-spam email, $P(x \mid y)$

$$P(x, y) = P(y)P(x \mid y)$$

To calculate $P(y \mid x_{\text{test}})$ for prediction, Naive Bayes algorithm uses Bayes' rule:

$$P(y \mid x) = \frac{P(x \mid y) \cdot P(y)}{P(x)}$$

Also, it assumes class-conditional independence, which means that each feature f_j in x is independent of other features given the label y. This lets us write $P(x \mid y)$ as a product of the individual feature probabilities:

$$P(x \mid y) = \prod_{j=1}^{d} P(f_j \mid y)$$

This is actually a simplistic or "naive" assumption as in real-life it is not true that the features are independent of each other. But still, this assumption reduces computational complexity by reducing the number of parameters significantly from 2^{d-1} to $2d + 1$, where d is the length of the dictionary assumed for the model. Even with this assumption, naive bayes works well for binary classification tasks like spam classification.

The Naive Bayes model includes parameters like,

- $\hat{p} = P(y = 1)$: the probability of a spam email.
- $\hat{p}_j^1 = P(f_j = 1 \mid 1)$: the probability of the j-th word appearing in a spam email
- $\hat{p}_j^0 = P(f_j = 1 \mid 0)$: the probability of the j-th word appearing in a non-spam email

2.2. naivebayes_classifier class. :

Attributes

- `prob_spam`: it is the estimated probability of an email being spam, $\hat{p}$.
- `prob_spam_words`: it is an array having the probabilities of each word given that the email is spam, $\hat{p}_j^1$.
- `prob_ham_words`: it is an array having the probabilities of each word given that the email is non-spam, $\hat{p}_j^0$.

Methods

`fit(self, X_train, y_train)`

- `X_train`: is a sparse matrix representation of training emails, where each row represents an email and each column represents a feature. We do the sparse implementation to save memory and time. `y_train`: A vector of labels (0 for non-spam, 1 for spam) corresponding to each email in `X_train`.
- This method learns a model from the training data. Using `get_wordcount` method, it gets the frequency of the words for spam and non-spam emails. It applies `laplace_smoothing` method to deal with zero probability corner cases. It finally calculates `self.prob_spam`,$\hat{p}$ textttself.prob_spam_words, $\hat{p}_j^1$ and self.prob_ham_words, $\hat{p}_j^0$ using the calculation given below.

  Using maximum likelihood estimation:

  1. $\hat{p}$: the fraction of spam emails in the dataset,

$$\hat{p} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

  where y_i is the label of the i-th email, and n is the total number of emails.

  2. $\hat{p}_j^y$: the fraction of y-labelled emails that contain the j-th word,

$$\hat{p}_j^y = \frac{\sum_{i=1}^{n} \mathbb{I}(f_{j,i} = 1, y_i = y)}{\sum_{i=1}^{n} \mathbb{I}(y_i = y)}$$

  where $\mathbb{I}$ is the indicator function

`get_wordcount(self, X, X_spam, X_ham)`

- `X` is the sparse matrix of training emails,`X_spam` is the list of spam emails and `X_ham` is the list of non-spam emails.
- this method returns `spam_words` and `ham_words` which are arrays containing the frequency of each word present in spam and non-spam emails, respectively.

`laplace_smoothing(self, spam_words, ham_words, spam_num, total)`

- `spam_words` and `ham_words` are arrays of frequencies of each word in spam emails and ham emails. `spam_num` is the number of spam emails and `total` is the total number of emails.
- In this function, we add 1 to each element of the arrays, spam_words and `ham_words`. This is equivalent to adding spam and non-spam email containing all the words. We add 4 and 2 respectively to `total` and `spam_num`. This means that we have added a spam and non-spam email which does not contain any of the words
- the method returns the updated frequency counts and the updated numbers of spam and non-spam emails.

We are adding 4 pseudo-emails (laplace smoothing) in the dataset, one spam and non-spam email which contains all the words. One spam and non-spam email which doesn't contain any word. By doing this we ensure that no $\hat{p}_j^y$ becomes zero, which could lead to comparision errors during prediction.

$$\hat{p}_j^y = \frac{\sum_{i=1}^{n} \mathbb{I}(f_{j,i} = 1, y_i = y) + 1}{\sum_{i=1}^{n} \mathbb{I}(y_i = y) + 2}$$

`log_probability(self, email, class_word_probs, class_prob)`

- `email` is a single email represented as a sparse feature vector, `class_word_probs` are the probabilities of words given the class and `class_prob` is the prior probability of the class (either $\hat{p}$ or $1 - \hat{p}$).
- this method returns $\log(P(y_i \mid x_i))$ so that we can perform the final prediction

We make use of logarithms over the direct probabilities because it simplifies the calculation of probabilities by converting products into sums. Also, the log transformation makes the final decision boundary of naive bayes a linear function as shown below.

`predict(self, X)`

- `X` is the sparse matrix representation of the testing emails.
- This method returns an array of predicted labels (0 for non-spam, 1 for spam) by performing these calculations:

To classify a tes email x_{test}, we calculate the probabilities $P(y = 1 \mid x_{\text{test}})$ and $P(y = 0 \mid x_{\text{test}})$ and choose the class with the higher probability. Using Bayes' rule, we compare:

$$\frac{P(y = 1 \mid x_{\text{test}})}{P(y = 0 \mid x_{\text{test}})} \geq 1 \implies \frac{P(x_{\text{test}} \mid y = 1) \cdot P(y = 1)}{P(x_{\text{test}} \mid y = 0) \cdot P(y = 0)} \geq 1$$

Taking logs of both sides:

$$\log\left(\frac{P(x_{\text{test}} \mid y = 1)}{P(x_{\text{test}} \mid y = 0)}\right) + \log\left(\frac{P(y = 1)}{P(y = 0)}\right) \geq 0$$

$$\sum_{j=1}^{d} \left[f_j \cdot \log\left(\frac{\hat{p}_j^1}{\hat{p}_j^0}\right) + (1 - f_j) \cdot \log\left(\frac{1 - \hat{p}_j^1}{1 - \hat{p}_j^0}\right) \right] + \log\left(\frac{\hat{p}}{1 - \hat{p}}\right) \geq 0$$

Taking

$$w_j = \log\left(\frac{\hat{p}_j^1(1 - \hat{p}_j^0)}{\hat{p}_j^0(1 - \hat{p}_j^1)}\right)$$

and

$$b = \log\left(\frac{1 - \hat{p}_j^1}{1 - \hat{p}_j^0}\right) + \log\left(\frac{\hat{p}}{1 - \hat{p}}\right),$$

we get:

$$w^T x_{\text{test}} + b \geq 0$$

Thus, we predict $y = 1$ if $w^T x_{\text{test}} + b \geq 0$, indicating that Naive Bayes has a linear decision boundary.

3. Code Overview

3.1. helper_functions.py. :

This python file contains helper functions so that `analysis.py` and `main.py` run smoothly. This file contains the following:
`common_words list`: This list contains common words like 'a', 'an', 'the' which are not informative in deciding whether an email is spam or not.

`process_dataset(email)`: This function processes an email string to make it more informative.

- `email` is a string, which is a single row of the panda series which contains the content of the email.

- It converts the entire email into lowercase in order to maintain consistency.
- Makes use of regular expressions to split the email into individual words and punctuation marks (`!` and `?`). It is important to capture punctuation marks like ? and ! because they contain useful information about an email being spam or not.
- It removes the words from the email string which are in `common_words` list and also those which are shorter than 3 letters. Doing this helps to improve the space and time complexity because we reduce the length of the processed email string
- This function finally returns the processed email string.

`most_frequent(emails, nrange)`:
This function gives the 20 most frequently occuring n-grams in an list of emails.

- `emails` is of panda series datatype. We pass the column of an email dataset, which contains the content of the emails. `nrange` specifies the range of n-grams for `CountVectorizer`

- We use `CountVectorizer` to count the number of n-grams in `emails` series. We include a custom `token_pattern` which ensures that punctuation marks (like `!` and `?`) are counted.

- This function returns a list of 20 tuples sorted as most to least frequent where each tuple contains a word and it's frequency.

`plot_graph(words, title)`

- `words` is a list of 20 tuples sorted as most to least frequent where each tuple contains a word and it's frequency and `title` provides the title for the plot, for example: "20 Most Frequent unigrams in Ham emails"

- This function uses `matplotlib.pyplot` to give a horizontal bar graph showing the frequencies of the 20 most common n-grams.

- The y-axis represents the words and x-axis represents their corresponding frequencies

`calculate_accuracy(clf, X, y)`

- `clf` is the classifier object which has already been fit on the training data. `X` is a binary sparse matrix which represents the words of the testing emails and `y` is a numpy array of the true predicitions
- This function calculates the accuracy of a classifier's predictions on a given dataset.

- It returns the accuracy of the classifier as the percentage of correctly predicted emails rounded to three decimal places

3.2. analysis.py. :

This python file contains the code which analyses data preprocessing and feature extraction. It shows the improvement in the accuracy of the naive bayes model on the email dataset after applying data preprocessing and feature extraction.

- In this file, I have imported libraries: (`pandas`, `numpy`), (`CountVectorizer` from `sklearn`), (`matplotlib.pyplot`). I have also imported the custom Naive Bayes classifier (`naive_bayes.py`) and helper functions (`helper_functions.py`), which will be useful for the smooth functioning of this python file.

- For the analysis, I have used `train.csv` and `analysis_test.csv`.

- Using the function `process_dataset`, we preprocess the emails of both the train and test datasets and store them in a column called `processed_email` in the `train` and `analysis_test` dataframes.

- We use `Binary = True` parameter in `CountVectorizer`. This counts only the presence and absence of the word in the emails and not the total frequency. We extract features from the emails in three ways:
  (1) `X_train1`, `X_test1`: created from unprocessed `email` text
  (2) `X_train2`, `X_test2`: created from processed email text using unigrams (single words).
  (3) `X_train3`, `X_test3`: created from processed email text using unigrams and bigrams (word pairs).

- I have divided the emails of `train.csv` into spam and ham emails for both unprocessed and processed text.Using `most_frequent` and `plot_graph` helper functions, we plot horizontal bar graphs of:

  (1) Top 20 unigrams in ham and spam emails (both raw and processed).
  (2) Top 20 bigrams in ham and spam emails (both raw and processed).
- This visualisation gives us insight that data pre-processing and using bigrams helps us to capture the most likely "spam" and "ham" phrases.

- (`naivebayes_classifier`) is trained and tested using
  (1) Unprocessed/raw emails using unigrams
  (2) Processed Emails using Unigrams
  (3) Processed emails using unigrams and bigrams

- We then plot a final bar graph of the accuracies calculated using `calculate_accuracy` function over the different raw and processed datasets using unigrams, bigrams. This highlights how data preprocessing and usage of bigrams affects model performance

3.3. main.py. :

This python file fufills the primary requirement of the assignemnent. If a folder named `test` containing emails as .txt files is present in the `solutions_da24c005` folder, this python file trains on `train.csv`, creates and generates `test_predictions.csv` file in the current directory. This file uses a `get_predictions(train)` function to do this:

- Emails from `train.csv` are loaded into a DataFrame named `train` with columns `email` and `label`. This is the training dataset on which our Naive Bayes model is trained on.
- Using python's `os` library, we perform file handlling and read the emails (`.txt` files) in the `test` folder. We prepare a DataFrame named `test` in which we store the name of the `.txt` file in the column `email_name` and content of the `.txt` in the column `email`

- Using `process_email` from `helper_functions.py`,we process the train and test emails like lowercasing, tokening, removing common words while retaining `!` and `?`, punctuation marks and store them in `processed_email` column of their respective DataFrames. For this, we use a `process_dataset` function.

- We use `CountVectorizer(binary=True, ngram_range=(1, 2))` to get binary counts (presence or absence) of unigrams and bigrams. We get a sparse representation of the presence or absence of uni and bigrams in `train['processed_email']` and `test['processed_email']`

- We create an object `nb` of `naivebayes_classifier` from `naive_bayes.py`. For `nb`, we train the model `X_train` and `y_train` using the `fit` method.

- Using the `predict` method of `nb`, we get the predictions of `X_test`. We save the predictions as a column of test_predictions, `test_predictions["Predictions"]`

- The function gives `test_predictions` as`test_predictions.csv` in `solutions_da24c005` folder. It also returns an array of predictions, which is stored in `predictions` variable

3.4. naive_bayes.py. :

This python file implements the `naivebayes_classifier` from scratch. I have explained the code and theory of this python file in section 2: Naive Bayes Classifier

4. Observations and Results

In this section, we focus on how data preprocessing and feature extraction can improve the performance of our naive bayes algorithm

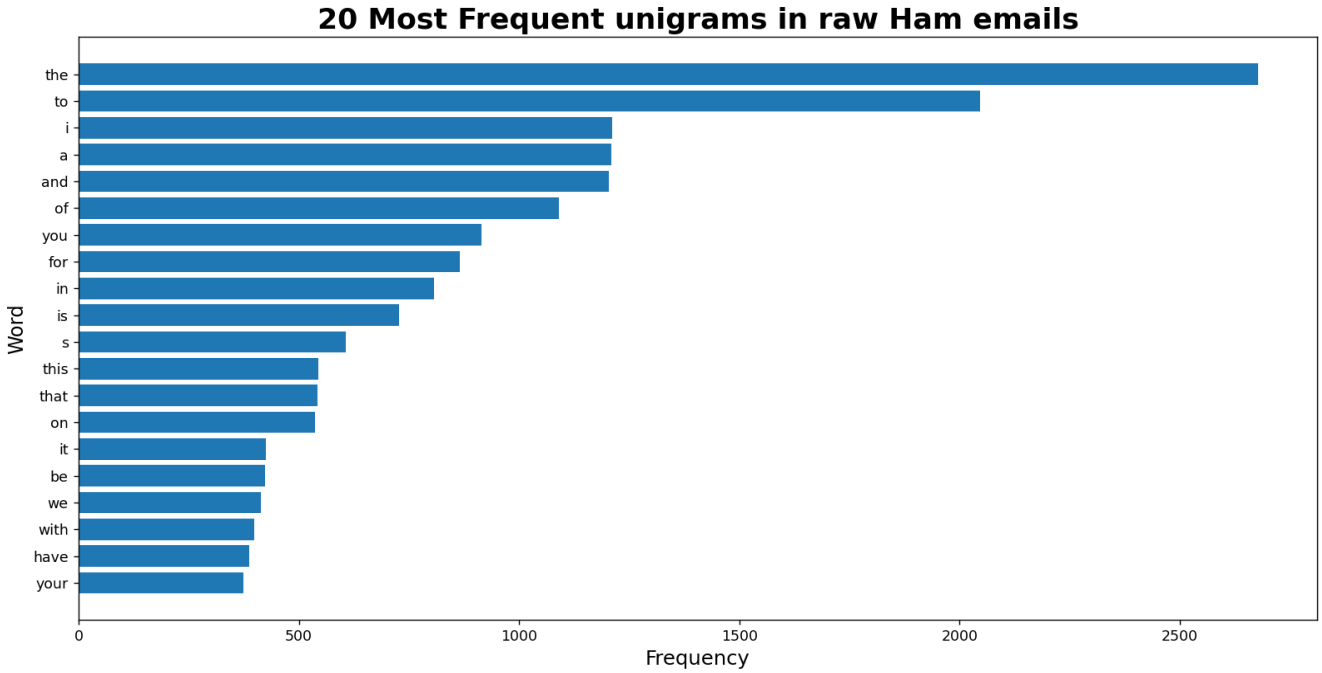

FIGURE 1.

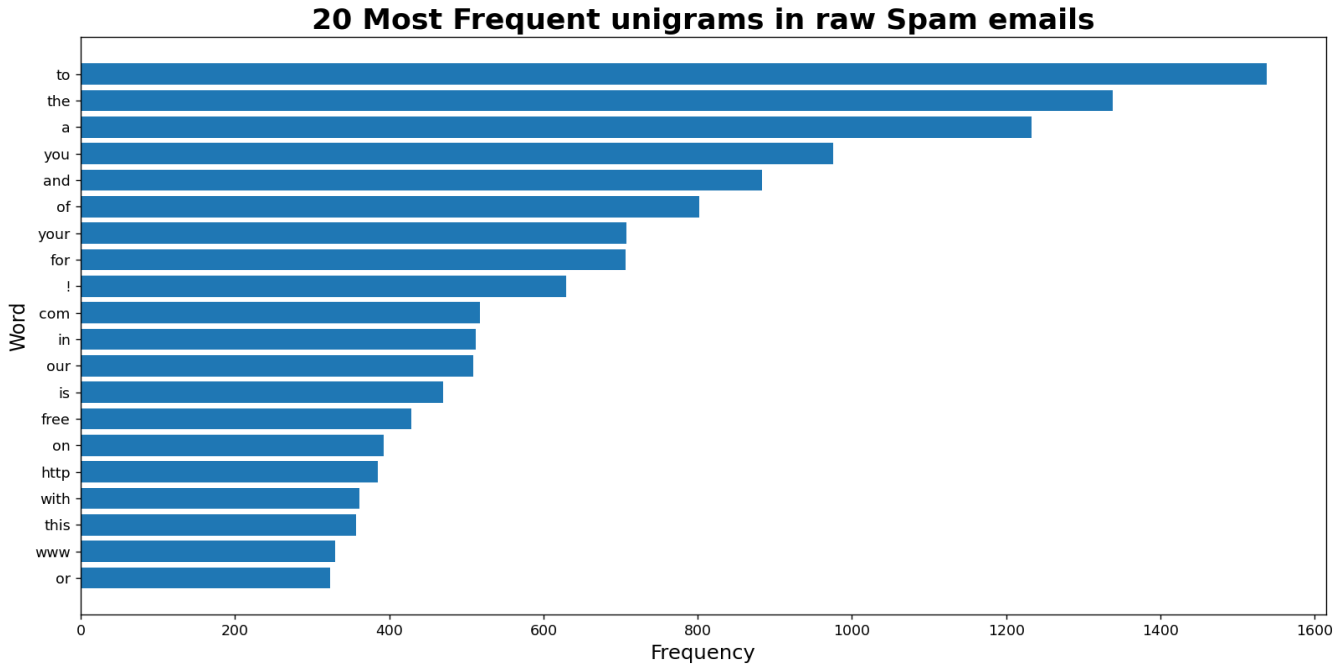

FIGURE 2.

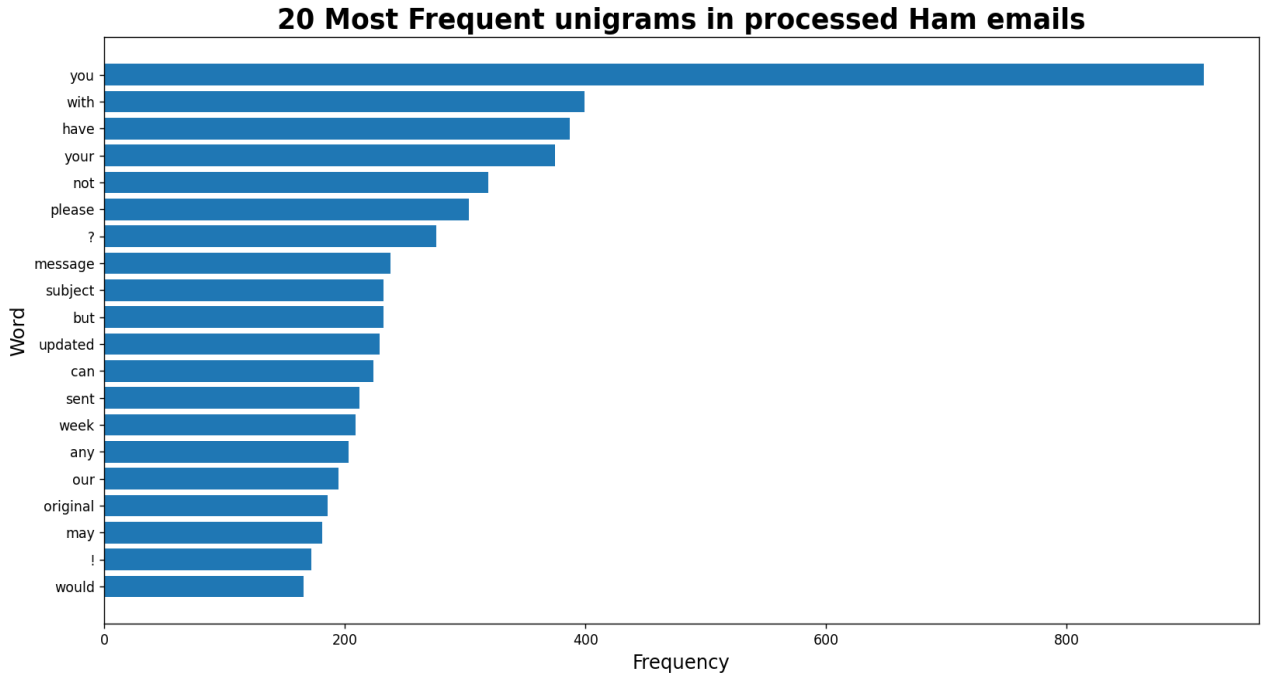

FIGURE 3.

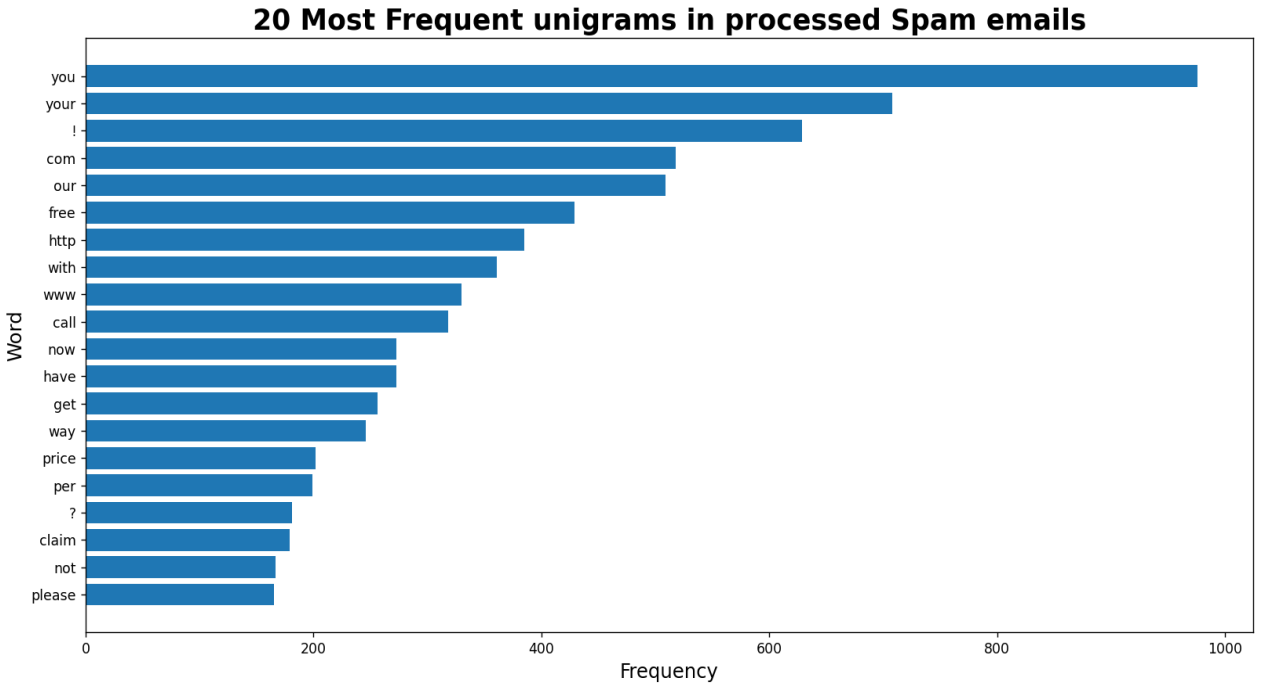

FIGURE 4.

These visualisations help us to gain insights for data preprocessing.

4.1. Data Preprocessing. :

From Figure 1, most frequently appearing unigrams (words) in unprocessed/raw ham emails are:

'the', 'to', 'I', 'a','and','of','you','for','in','is','s','this','that','on','it','be','we', 'with','have','your'.

From figure 2, most frequently appearing unigrams (words) in unprocessed/raw spam emails are:

'to','the','a','you','and','of','your','for','!','com','in','our','is', 'free','on','http', 'with', 'this', 'www','or'

- We observe that the most frequently appearing words in both spam and ham emails are prepositions ('to', 'of', 'for', 'in', 'on', 'with'), articles('the','a'), auxiliary verbs ('is', 'be', 'have'), conjunction ('and'). These words are not informative, they are merely used for grammatical completion of the sentence. Since, they are found both in spam and ham emails, these words will not be very useful to distinguish between spam and ham emails. These words can be regarded as "noise" and not part of the structure of the model.

Based on this observation, we preprocess the emails using `process_email` function. Using this function, we remove the common stop words like 'to', 'of', 'for', 'the','a', 'on', 'all','new', 'had','been'.... Apart from these common words, we also remove words with less than three letters, because these words are highly likely to be typos or some form of 'noise' and they will not help us in distinguishing between ham and spam emails. Also, in this function we use `re` library to extract all words and any '!' or '?' characters from the email string.

From Figure 3, most frequently appearing unigrams (words) in processed ham emails are: 'you','with','have','your','not','please','?','message','subject', but','updated','can','sent', 'week', 'any','our','original','may','!','would'.

From figure 4, most frequently appearing unigrams (words) in processed spam emails are: 'you','your','!','com','our','free','http','with','www','call',now', 'have', 'get', 'way', 'price', 'per', '?', 'claim', 'not','please'

- After removing common stop words and words less than three letters, we get meaningful words which will help us in our spam classification task
- We observe that words like 'you' and 'your' appear frequently in both spam and ham emails. But the contexts they appear in are different.
  - 'You' and 'your' occur commonly in spam emails in phrases like 'Congratulations! You have won a prize!', 'Your account has been hacked; click here to secure it.'

  - They are also used in ham emails by friends, colleagues to address the recipient like, "Will you come to the party this weekend?", "You need to send the report by today."

- We observe that words like 'please', 'would', 'may' appear more frequently in ham emails because ham emails tend to use polite and formal language specially official emails like "Could you please send the updated document?", "Would you be available for a meeting this afternoon?", "You may want to review the attached file."

- We observe that the exclamation mark, '!' appears more frequently than '?' in spam emails. And in ham emails, '?' appears more frequently than '!' in ham emails. In spam emails, '!' is used usually to create a sense of urgency and excitement and to grab the readers attention. In ham emails, "?" appears more frequently than '!' because it involves conversation based messages between friends or colleagues.

- We observe that web related terms like "www," "http," and "com," are appearing more frequently in spam emails which are usually links to external sites. These links usually lead to unwanted or malicious websites.

- We also observe words like "claim," "free," "call," and "now," in spam emails which are found in phrases like "Claim your prize now!", "Get your free trial today!", "Call us now for exclusive offers!"

By preprocessing the emails, we were able to eliminate the common stop words and we could focus on informative words which will really helps us in spam classification. Also, by data preprocessing we can bring down the number of features in our dataset since we are focussing only on the structure of the model (informative words) by ignoring the noise (common words).This helps to reduce overfitting and helps our model to perform well on unseen test data. Reducing the number of features helps to run the model efficiently by saving time and space.

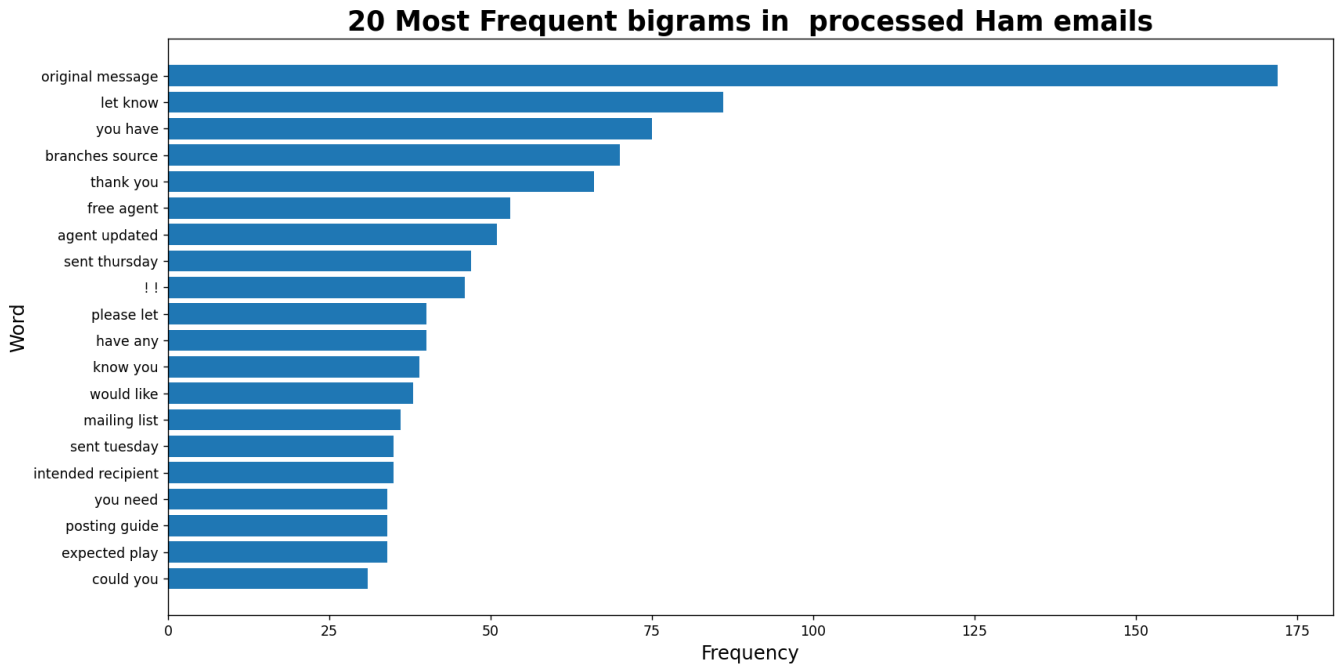

FIGURE 5.

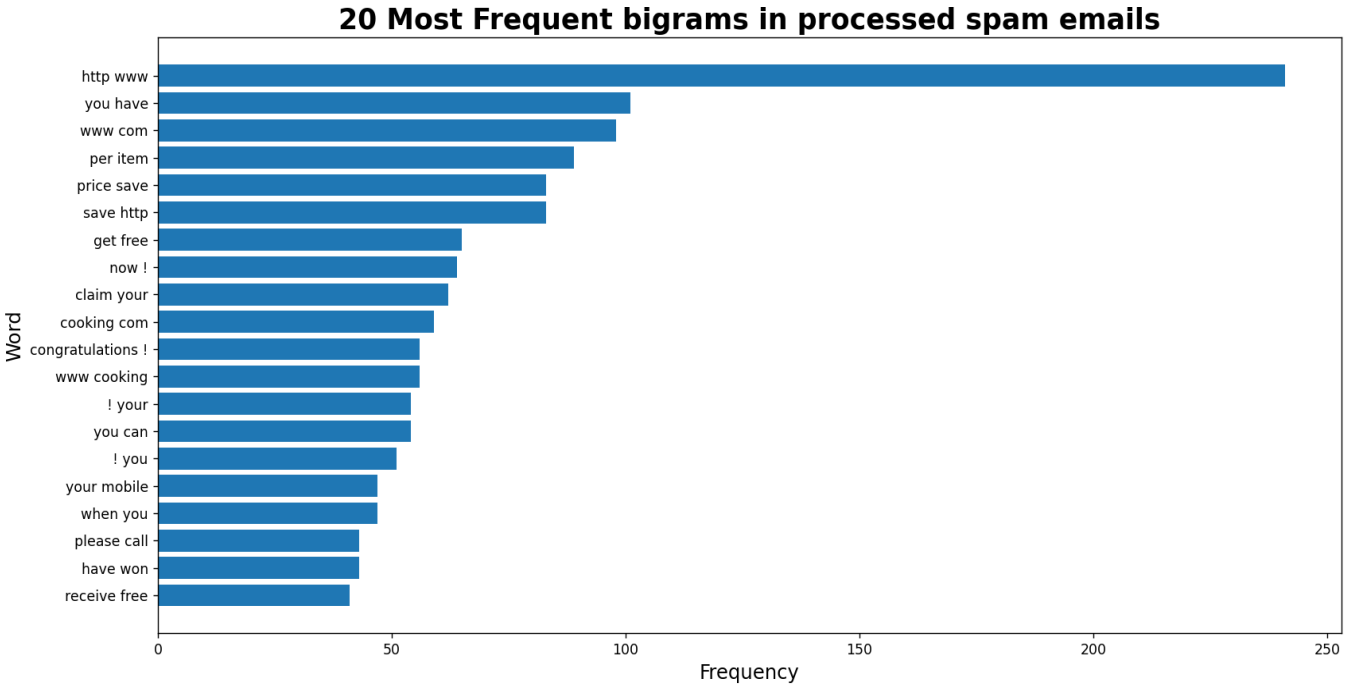

FIGURE 6.

4.2. Feature Extraction. :

From Figure 5, most frequently appearing bigrams (word pairs) in processed ham emails are:
'original message', 'let know', 'you have' 'branches source', 'thank you', 'free agent', 'agent updated', 'sent thursday', '! !', 'please let', 'have any', 'know you', 'would like', 'mailing list', 'sent tuesday', 'intended recipient', 'you need', 'posting guide', 'expected play','could you'

From figure 6, most frequently appearing bigrams (word pairs) in processed spam emails are:
'http www', 'you have', 'www com', 'per item', 'price save', 'save http', 'get free', 'now!', 'claim your', 'cooking com', 'congratulations!', 'www cooking', '!you','your mobile','when you','please call', 'have won','recieve free'

- We observe that ham emails contain frequently occuring bigrams such as 'let (me) know', 'thank you', 'could you', 'please let'. These phrase are used in personal and official communications.

- Spam emails contain bigrams like 'http www', 'www com', 'save http' which are associated with external links to unwanted or malicious websites.

- Spam emails also contain bigrams like 'get free', 'claim you', 'have won', 'recieve free'. This is found in emails like "Get free access to our premium content!, "Claim your prize now!". These phrases are used to make the reader to take an impulsive decision which will benefit the sender

- Using bigrams, we can also obtain the words that commonly apear with exclamation mark, '!' like 'now!', 'congratulations!', '!you'

By using bigrams and unigrams, we can not only get words that appear commonly in spam and ham, we can also capture patterns of words that commonly occur in pairs like 'thank you', 'recieve free'. It also gives insight into the context in which the word used. For example:

- Claim your prize
- Your report is pending

Here, your is used in both emails but using bigrams, we can capture the word pair 'claim your' and 'your report' which helps in clearly distinguishing spam and ham email.

We implement this, by using `CountVectorizer(binary=True, ngram_range=(1, 2))` Here, `ngram_range = (1,2)` specifies that we are counting both uni and bigrams.

Therefore, for our naive bayes classifier, we have extracted features from an email string as a vector, which is a binary representation of the presence or absence of a word (unigram) and word pair (bigram).

4.3. Results. Data preprocessing and appropriate feature extraction helped to increase the performance of my naive bayes model.

- `X_train1` and `X_test1` are train and testing datasets in raw form. We use unigram binary counts on these datasets
- `X_train2` and `X_test2` are train and testing datasets in processed form. We use unigram binary counts on these datasets
- `X_train3` and `X_test3` are train and testing datasets in processed form. We use unigrams and bigrams binary counts on these datasets

These are the accuracies and the corresponding visualisation of my naive bayes model on these different datasets:

```
Accuracy over raw test emails(unigrams), X_test1: 82.356
Accuracy over processed test emails (unigrams), X_test2: 83.957
Accuracy over processed test emails (unigrams, bigrams), X_test3: 87.838
```

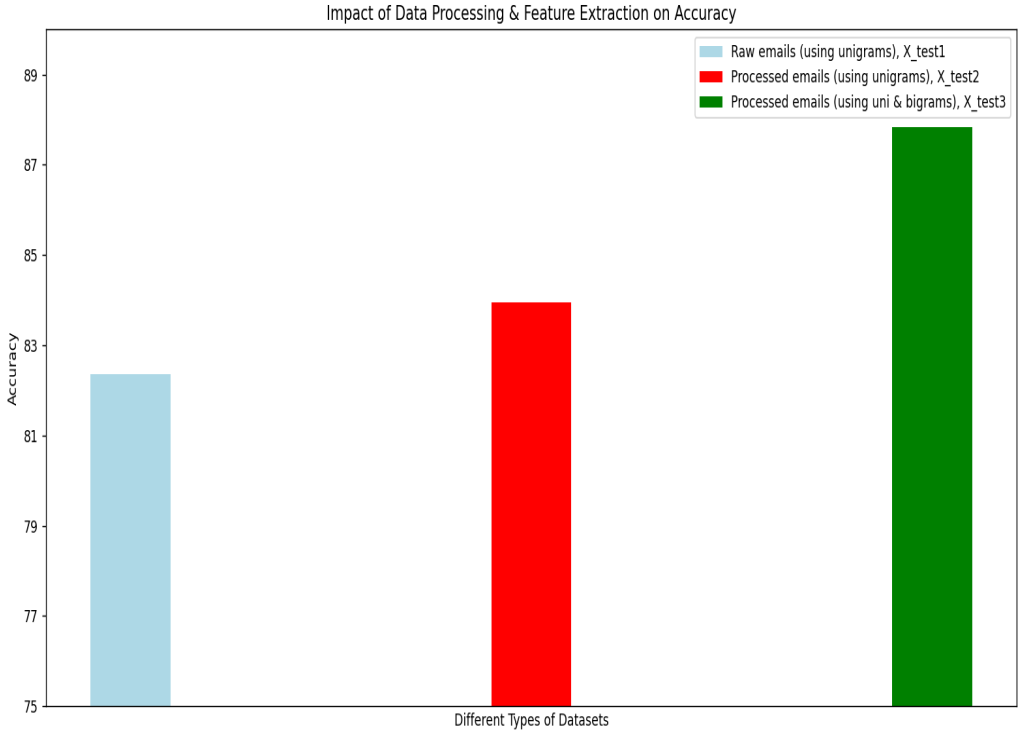

FIGURE 7.