Technological Institute of the Philippines	Quezon City - Computer Engineering
Course Code:	CPE 019
Code Title:	Emerging Technologies in CpE 2
2nd Semester	AY 2023-2024

ACTIVITY	Prelim Examination
Name	Cuevas, Christian Jay L.
Section	CPE32S3
Date Performed:	3/26/24
Date Submitted:	3/26/24
Instructor:	Engr. Roman M. Richard

INSTRUCTIONS:

In this assignment, you are task to build a multilayer perceptron model. The following are the requirements:

- Choose any dataset
- Explain the problem you are trying to solve
- Create your own model
- Evaluate the accuracy of your model

DATASET:

- The dataset that I am using is from UCI Machine Learning Repository and it is about identifying spam messages using different combinations of words. These words are the features of the dataset, they were collected in spam emails and are the features of this dataset. This dataset was collected from a personal email so the indicator of "not spam" is the word "george" and the area code "650".

Image for visualization only

PROBLEM:

- The problem we're trying to solve is the identification of spam emails and legitimate emails by using the multilayer perceptron model and the different words as features that will be used in training the model. This can be useful in creating a personal spam filter that can be used in personal emails.

CODING:

IMPORTING LIBRARIES AND DATASET:

```
pip install ucimlrepo
```

```
Collecting ucimlrepo
  Downloading ucimlrepo-0.0.6-py3-none-any.whl (8.0 kB)
Installing collected packages: ucimlrepo
Successfully installed ucimlrepo-0.0.6
```

```
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
import matplotlib.pyplot as plt
from ucimlrepo import fetch_ucirepo
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Activation

```

```
#Fetch the dataset from UCI repository
#94 is the id of our dataset Spambase
spambase = fetch_ucirepo(id=94)
```

```
#Get the original data from the UCI Repository
#This contains all the labels, features, and target variable
all_data = spambase.data.original
```

EXPLORATORY DATA ANALYSIS:

- The first step is to look at the number of features, number of columns, and data types.

```
all_data.info()
```

```
 0   word_freq_make              4601 non-null   float64
 1   word_freq_address           4601 non-null   float64
 2   word_freq_all               4601 non-null   float64
 3   word_freq_3d                4601 non-null   float64
 4   word_freq_our               4601 non-null   float64
 5   word_freq_over              4601 non-null   float64
 6   word_freq_remove            4601 non-null   float64
 7   word_freq_internet          4601 non-null   float64
 8   word_freq_order             4601 non-null   float64
 9   word_freq_mail              4601 non-null   float64
 10  word_freq_receive           4601 non-null   float64
 11  word_freq_will              4601 non-null   float64
 12  word_freq_people            4601 non-null   float64
 13  word_freq_report            4601 non-null   float64
 14  word_freq_addresses         4601 non-null   float64
 15  word_freq_free              4601 non-null   float64
 16  word_freq_business          4601 non-null   float64
 17  word_freq_email             4601 non-null   float64
 18  word_freq_you               4601 non-null   float64
 19  word_freq_credit            4601 non-null   float64
 20  word_freq_your              4601 non-null   float64
 21  word_freq_font              4601 non-null   float64
 22  word_freq_000               4601 non-null   float64
 23  word_freq_money             4601 non-null   float64
 24  word_freq_hp                4601 non-null   float64
 25  word_freq_hpl               4601 non-null   float64
 26  word_freq_george            4601 non-null   float64
 27  word_freq_650               4601 non-null   float64
 28  word_freq_lab               4601 non-null   float64
 29  word_freq_labs              4601 non-null   float64
 30  word_freq_telnet            4601 non-null   float64
 31  word_freq_857               4601 non-null   float64
 32  word_freq_data              4601 non-null   float64
 33  word_freq_415               4601 non-null   float64
 34  word_freq_85                4601 non-null   float64
 35  word_freq_technology        4601 non-null   float64
 36  word_freq_1999              4601 non-null   float64
 37  word_freq_parts             4601 non-null   float64
 38  word_freq_pm                4601 non-null   float64
 39  word_freq_direct            4601 non-null   float64
 40  word_freq_cs                4601 non-null   float64
 41  word_freq_meeting           4601 non-null   float64
 42  word_freq_original          4601 non-null   float64
```

```
 48  char_freq_;                  4601 non-null   float64
 49  char_freq_(                  4601 non-null   float64
 50  char_freq_[                  4601 non-null   float64
 51  char_freq_!                  4601 non-null   float64
 52  char_freq_$                  4601 non-null   float64
 53  char_freq_#                  4601 non-null   float64
 54  capital_run_length_average  4601 non-null   float64
 55  capital_run_length_longest  4601 non-null   int64
 56  capital_run_length_total    4601 non-null   int64
 57  Class                       4601 non-null   int64
```

Observation:

- As you can see above, this dataset has a total of 58 attributes and 55 of them are float64 and 3 are int64 data types. If we scan the null count, we can see that there are no missing data, but we will further confirm that later. Additionally, we have a total of 4601 data entries.

- Check if there is a null value in the dataset.

```
all_data.isnull().sum()
```

```
word_freq_make                0
word_freq_address             0
word_freq_all                 0
word_freq_3d                  0
word_freq_our                 0
word_freq_over                0
word_freq_remove              0
word_freq_internet            0
word_freq_order               0
word_freq_mail                0
word_freq_receive             0
word_freq_will                0
word_freq_people              0
word_freq_report              0
word_freq_addresses           0
word_freq_free                0
word_freq_business            0
word_freq_email               0
word_freq_you                 0
word_freq_credit              0
word_freq_your                0
word_freq_font                0
word_freq_000                 0
word_freq_money               0
word_freq_hp                  0
word_freq_hpl                 0
word_freq_george              0
word_freq_650                 0
word_freq_lab                 0
word_freq_labs                0
word_freq_telnet              0
word_freq_857                 0
word_freq_data                0
word_freq_415                 0
word_freq_85                  0
word_freq_technology          0
word_freq_1999                0
word_freq_parts               0
word_freq_pm                  0
word_freq_direct              0
word_freq_cs                  0
word_freq_meeting             0
word_freq_original            0
word_freq_project             0
word_freq_re                  0
word_freq_edu                 0
word_freq_table               0
word_freq_conference          0
char_freq_;                   0
char_freq_(                   0
char_freq_[                   0
char_freq_!                   0
char_freq_$                   0
char_freq_#                   0
capital_run_length_average    0
capital_run_length_longest    0
capital_run_length_total      0
Class                         0
dtype: int64
```

Observation:

- You can observe above that there is no null values in the given dataset. This makes it easier because we do not have to fill or drop missing values.

- Look at the first 5 rows of the dataset to check the values of the data.

```
1 all_data.head()
```

et	word_freq_order	word_freq_mail	...	char_freq_;	char_freq_(	char_freq_[	char_freq_!	char_freq_$	char_freq_#	capital_run_length_average
00	0.00	0.00	...	0.00	0.000	0.0	0.778	0.000	0.000	3.756
07	0.00	0.94	...	0.00	0.132	0.0	0.372	0.180	0.048	5.114
12	0.64	0.25	...	0.01	0.143	0.0	0.276	0.184	0.010	9.821
63	0.31	0.63	...	0.00	0.137	0.0	0.137	0.000	0.000	3.537
63	0.31	0.63	...	0.00	0.135	0.0	0.135	0.000	0.000	3.537

Observation:

- here we can increase our understanding with our data, by looking at the value of our data, we can see that it contains floating values. These floating values represent the percentage of words that match the specified word or char in the e-mail. For example the "word_freq_all" column has a value of 0.64 in the first column, this means that the word "all" appeared in the e-mail with a frequency of 64%. There are words that have 0%, this means that they did not appear in that e-mail message. Also, the class has binary value of 0 and 1, 0 means it is not spam while 1 means it is a spam.

- To further observe the data, let us use .describe(). This returns count, mean, std, min, 25%, 50%, 75% and max.
- This can help in proving that all of the values are in the range of 0 - 1, which can help in identifying outliers.

```
1 all_data.describe()
```

ɔrd_freq_order	word_freq_mail	...	char_freq_;	char_freq_(	char_freq_[	char_freq_!	char_freq_$	char_freq_#	capital_run_length_average	capit
4601.000000	4601.000000	...	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	
0.090067	0.239413	...	0.038575	0.139030	0.016976	0.269071	0.075811	0.044238	5.191515	
0.278616	0.644755	...	0.243471	0.270355	0.109394	0.815672	0.245882	0.429342	31.729449	
0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	
0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.588000	
0.000000	0.000000	...	0.000000	0.065000	0.000000	0.000000	0.000000	0.000000	2.276000	
0.000000	0.160000	...	0.000000	0.188000	0.000000	0.315000	0.052000	0.000000	3.706000	
5.260000	18.180000	...	4.385000	9.752000	4.081000	32.478000	6.003000	19.829000	1102.500000	

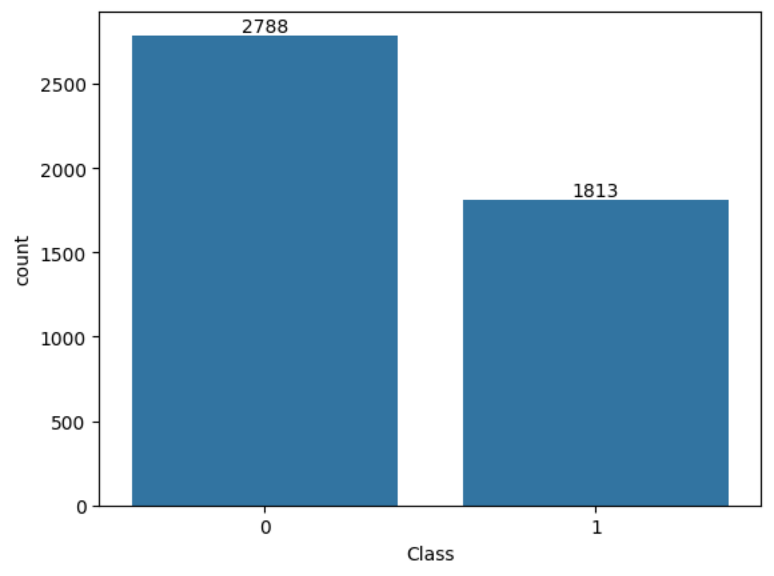
Observation:

- We can observe that the minimum and maximum value of our data is in the range of 0 - 1, which proves the claim and validity of this dataset.

- Now, after observing the features or attributes of this dataset, we also need to look at the target variables.

```
1 ax = sns.countplot(data=all_data, x='Class')
2 ax.bar_label(ax.containers[0])
```

```
[Text(0, 0, '2788'), Text(0, 0, '1813')]
```



Observation:

- You can observe that the instance of the class "not spam" is higher than the instance of the class "spam". The number of not spam instances is 2788 while the number of mines instances is 1813.
- Why do we need to check for imbalance? We check for class imbalance because this can affect the predictive performance of our model [2] and it can also produce bias when the model is making a decision [1,2].
- To check for the degree of imbalance, we can use the Imbalance Ratio, which is denoted by IR = total of negative class / total of positive class [1]. Positive class is the class we're testing for [3] which is the "spam" class, and the negative class is the class is the other possibility that our model is also testing [4].
- In our case, IR = 2788 / 1813 is equal to 1.5377, which is close to 1. When the result of the ratio is close to 1, this means that our class is not imbalanced [5].

- We need to also check for their correlation to find out if there is a correlation between the features and the target variable.

```
all_data.corr(method="pearson", numeric_only = True)
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet	word_freq
	1.000000	-0.016759	0.065627	0.013273	0.023119	0.059674	0.007669	-0.003950	0
	-0.016759	1.000000	-0.033526	-0.006923	-0.023760	-0.024840	0.003918	-0.016280	-0
	0.065627	-0.033526	1.000000	-0.020246	0.077734	0.087564	0.036677	0.012003	0
	0.013273	-0.006923	-0.020246	1.000000	0.003238	-0.010014	0.019784	0.010268	-0
	0.023119	-0.023760	0.077734	0.003238	1.000000	0.054054	0.147336	0.029598	0
	0.059674	-0.024840	0.087564	-0.010014	0.054054	1.000000	0.061163	0.079561	0
	0.007669	0.003918	0.036677	0.019784	0.147336	0.061163	1.000000	0.044545	0
	-0.003950	-0.016280	0.012003	0.010268	0.029598	0.079561	0.044545	1.000000	0
	0.106263	-0.003826	0.093786	-0.002454	0.020823	0.117438	0.050786	0.105302	1
	0.041198	0.032962	0.032075	-0.004947	0.034495	0.013897	0.056809	0.083129	0
	0.188459	-0.006864	0.048254	-0.012976	0.068382	0.053900	0.159578	0.128495	0
	0.105801	-0.040398	0.083210	-0.019221	0.066788	0.009264	-0.001461	-0.002973	0
	0.066438	-0.018858	0.047593	-0.013199	0.031126	0.077631	0.013295	0.026274	0
	0.036780	-0.009206	0.008552	0.012008	0.003445	0.009673	-0.022723	0.012426	0
s	0.028439	0.005330	0.122113	0.002707	0.056177	0.173066	0.042904	0.072782	0
	0.059386	-0.009117	0.063906	0.007432	0.083024	0.019865	0.128436	0.051115	0
s	0.081928	-0.018370	0.036262	0.003470	0.143443	0.064137	0.187981	0.216422	0
	0.053324	0.033500	0.121923	0.019391	0.062344	0.078350	0.122011	0.037738	0
	0.128243	-0.055476	0.139329	-0.010834	0.098510	0.095505	0.111792	0.020641	0
	0.021295	-0.015806	0.031111	-0.005381	0.031526	0.058979	0.046134	0.109163	0
	0.197049	-0.018191	0.156651	0.008176	0.136605	0.106833	0.130794	0.156905	0
	-0.024349	-0.008850	-0.035681	0.028102	-0.020207	0.007956	-0.002093	-0.016192	-0
	0.134072	-0.020502	0.123671	0.011368	0.070037	0.211455	0.064795	0.089226	0
	0.188155	0.001984	0.041145	0.035360	0.000039	0.059329	0.030575	0.034127	0
	-0.072504	-0.043483	-0.087924	-0.015181	-0.072502	-0.084402	-0.089494	-0.053038	-0
	-0.061686	-0.038211	-0.062459	-0.013708	-0.075456	-0.087271	-0.080330	-0.041450	-0
	-0.066424	-0.030307	-0.108886	-0.010684	-0.088011	-0.069051	-0.065893	-0.057189	-0
	-0.048680	-0.029221	-0.050648	-0.010368	-0.061501	-0.066223	-0.066947	-0.049988	-0
	-0.041251	-0.021940	-0.057726	-0.007798	0.032048	-0.048673	-0.048482	-0.037047	-0
	-0.052799	-0.027508	-0.032547	-0.010476	-0.052066	-0.048127	-0.058101	-0.043405	-0
	-0.039066	-0.018097	-0.038927	-0.007529	-0.042535	-0.046383	-0.046280	-0.035816	-0
	-0.032058	-0.003326	-0.061870	-0.006717	-0.026748	-0.036835	-0.040538	-0.034276	-0
	-0.041014	-0.024903	-0.054759	-0.008075	-0.031998	-0.034164	-0.041372	-0.039220	-0
	-0.027690	-0.004303	-0.061706	-0.006729	-0.026960	-0.037315	-0.040910	-0.034811	-0
	-0.044954	-0.024058	-0.048335	-0.006122	-0.049732	-0.054315	-0.053202	-0.035174	-0
gy	-0.054673	-0.028198	-0.046504	-0.006515	-0.048844	-0.052819	-0.053978	-0.033747	-0
	-0.057312	-0.024013	-0.067015	-0.007761	-0.072599	-0.057465	-0.052035	-0.017466	-0
	-0.007960	-0.008922	0.032407	-0.002669	0.130812	-0.017918	-0.014781	-0.012119	-0
	-0.011134	-0.019124	-0.014809	-0.004602	-0.042044	-0.047619	-0.046978	-0.030392	-0
	-0.036095	-0.014821	-0.047066	-0.007643	-0.021442	-0.029866	-0.022121	-0.005988	-0
	-0.009703	-0.015420	-0.030956	-0.005670	-0.047505	-0.029457	-0.033120	-0.003884	-0
	-0.026070	-0.025177	-0.005811	-0.008095	0.115041	-0.054812	-0.049664	-0.043626	-0
	-0.024292	-0.002370	-0.044325	-0.009268	-0.048879	-0.030616	-0.049079	-0.004542	-0
	-0.022116	-0.019739	-0.053464	-0.005933	0.015234	-0.028826	-0.034461	-0.030134	-0
	-0.037105	-0.016418	-0.050664	-0.012957	-0.042336	-0.053637	-0.050811	-0.002423	-0
	-0.034056	-0.023858	-0.056655	-0.009181	-0.077986	-0.033046	-0.056166	-0.037916	-0
	-0.000953	-0.009818	0.029339	-0.003348	-0.026900	-0.014343	-0.017512	-0.006397	0
ce	-0.017755	-0.015747	-0.026344	-0.001924	-0.032005	-0.031693	-0.031408	-0.021224	-0
	-0.026505	-0.007282	-0.033213	-0.000591	-0.032759	-0.019119	-0.033089	-0.027432	-0

	-0.021196	-0.049837	-0.016495	-0.012370	-0.046361	-0.008705	-0.051885	-0.032494	-0
	-0.033301	-0.018527	-0.033120	-0.007148	-0.026390	-0.015133	-0.027653	-0.019548	0
	0.058292	-0.014461	0.108140	-0.003138	0.025509	0.065043	0.053706	0.031454	0
	0.117419	-0.009605	0.087618	0.010862	0.041582	0.105692	0.070127	0.057910	0
	-0.008844	0.001946	-0.003336	-0.000298	0.002016	0.019894	0.046612	-0.008012	-0
rage	0.044491	0.002083	0.097398	0.005260	0.052662	-0.010278	0.041565	0.011254	0
gest	0.061382	0.000271	0.107463	0.022081	0.052290	0.090172	0.059677	0.037575	0
tal	0.089165	-0.022680	0.070114	0.021369	0.002492	0.082089	-0.008344	0.040252	0
	0.126208	-0.030224	0.196988	0.057371	0.241920	0.232604	0.332117	0.206808	0

```
all_data.corr(method="pearson", numeric_only = True)['Class'].sort_values(ascending=False)
```

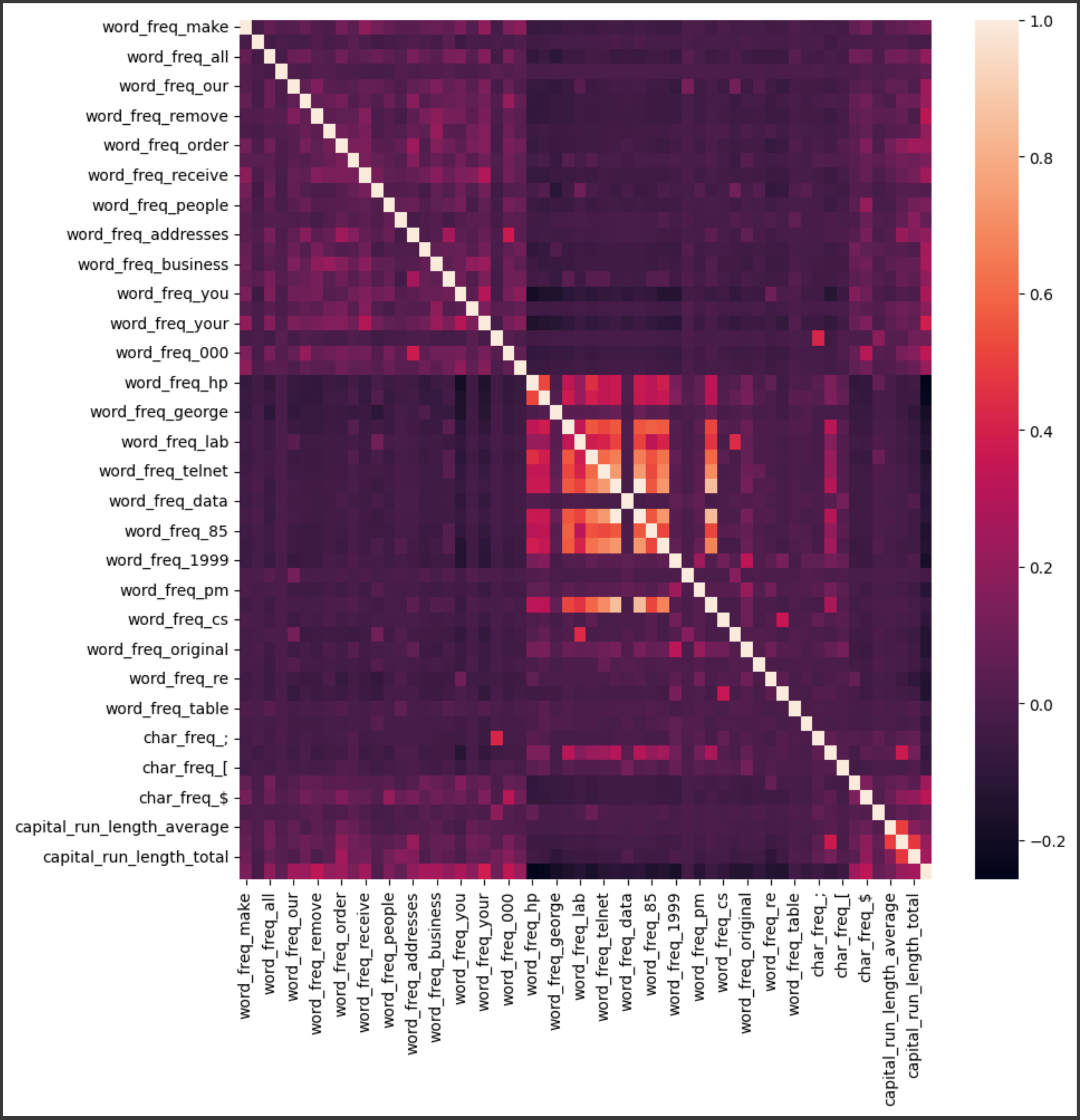
```
Class                         1.000000
word_freq_your                0.383234
word_freq_000                 0.334787
word_freq_remove              0.332117
char_freq_$                   0.323629
word_freq_you                 0.273651
word_freq_free                0.263215
word_freq_business            0.263204
capital_run_length_total      0.249164
word_freq_our                 0.241920
char_freq_!                   0.241888
word_freq_receive             0.234529
word_freq_over                0.232604
word_freq_order               0.231551
word_freq_money               0.216111
capital_run_length_longest    0.216097
word_freq_internet            0.206808
word_freq_email               0.204208
word_freq_all                 0.196988
word_freq_addresses           0.195902
word_freq_credit              0.189761
word_freq_mail                0.138962
word_freq_people              0.132927
word_freq_make                0.126208
capital_run_length_average    0.109999
word_freq_font                0.091860
char_freq_#                   0.065067
word_freq_report              0.060027
word_freq_3d                  0.057371
word_freq_will                0.007741
word_freq_address            -0.030224
word_freq_parts              -0.031035
word_freq_table              -0.044679
char_freq_;                  -0.059630
char_freq_[                  -0.064709
word_freq_direct             -0.064801
word_freq_conference         -0.084020
char_freq_(                  -0.089672
word_freq_project            -0.094594
word_freq_cs                 -0.097375
word_freq_415                -0.112754
word_freq_857                -0.114214
word_freq_data               -0.119931
word_freq_pm                 -0.122831
word_freq_telnet             -0.126912
word_freq_lab                -0.133523
word_freq_original           -0.135664
word_freq_technology         -0.136134
word_freq_meeting            -0.136615
word_freq_re                 -0.140408
word_freq_edu                -0.146138
word_freq_85                 -0.149225
word_freq_650                -0.158800
word_freq_labs               -0.171095
word_freq_1999               -0.178045
word_freq_george             -0.183404
word_freq_hpl                -0.232968
word_freq_hp                 -0.256723
Name: Class, dtype: float64
```

Observation:

- We can see in the results above that there is a medium positive correlation between the attributes which is between the values 0.3 - 0.5 [6]. There is also a weak negative correlation in certain attributes ranging from -0.1 - -0.3 [6]. This means that there is a weak correlation between the attributes and the target variable "class".

```
1 fig = plt.figure(figsize=(10,10), dpi=100)
2
3 sns.heatmap(all_data.corr())
```

<Axes: >



Observation:

- Here, we observe in this heatmap that it is brighter in the middle part of the heatmap along the diagonal line. This means that there is a high correlation between the attributes closer to each other. Regarding the target variable "Class", we can't see it in the heatmap but according to the descending order of correlation displayed above, word_freq_your has the highest correlation with it with 38% frequency.

SPLITTING THE DATASET INTO TRAINING, VALIDATION AND TEST DATA:

- Splitting the dataset into training and test data is very important because it will show the predictive power of our model. The training data will be used to train the model and the testing data will be used to test the final model [7]. Validation data on the other hand is used during the training of our model and it is the subset where we initially validate the trained model, before using the test data [8].
- The data will be split to 60/20/20, where 60 will be the training data, 20 will be the validation data, and 20 will be the test data [9, 10].

```
X = all_data.drop(["Class"], axis = 1)
y = all_data["Class"]

X, X_test, y, y_test = train_test_split(X, y, test_size =0.2, random_state=10)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size =0.2, random_state=10)
```

```
X_train.shape, X_test.shape, X_val.shape
```

```
((2944, 57), (921, 57), (736, 57))
```

```
y_train.shape, y_test.shape, y_val.shape
```

```
((2944,), (921,), (736,))
```

TRAINING THE MODEL:

- We're using Multilayer Perceptron model to train our training dataset. MLP is a very powerful tool that is effective in non linear dataset [11]. It consists of the input layer, hidden layer(s), and the output layer [11]. There are nodes in each layer which has values and weights, the values are from the dataset but the weights are composed of heuristics or random values [12]. The model learns with back propagation and gradient descent.
- Our input layer will be consisting of 60 nodes which are the 60 features that we have, and the weights will be random.
- How do we decide the number of nodes in the hidden layers? The rule of thumb is that the number of nodes in the hidden layer is 2/3 of the input layer and less than of the twice the input layer [13]. The number of the nodes in the hidden layer is 40.
- The number of the nodes in our output layer is 1, if the value is 0 it means it is the class "Rock" and if the value is 1 it means it is the class "Mines".
- I used Rectified Linear Unit because it is the most recommended one in multilayer perceptron model [14].

```
#Initial model
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(57),
    tf.keras.layers.Dense(38, activation = "relu"),
    tf.keras.layers.Dense(1, activation = "sigmoid")
])
```

```
model.compile(optimizer = "adam",
              loss = "binary_crossentropy",
              metrics=["accuracy"]
              )
```

- The number of epochs is arbitrary and will be adjusted accordingly after observing the graph of loss.
- The number of batch size is 32 because it is a good default value [16].

```
spam_data = model.fit(X_train, y_train, validation_data = (X_val, y_val), epochs = 50, batch_size = 32)
```

```
Epoch 14/50
92/92 [==============================] - 0s 2ms/step - loss: 1.4268 - accuracy: 0.8614 - val_loss: 0.6378 - val_accuracy: 0.8913
Epoch 15/50
92/92 [==============================] - 0s 3ms/step - loss: 0.8930 - accuracy: 0.8692 - val_loss: 0.6224 - val_accuracy: 0.8981
```

```
92/92 [==============================] - 0s 3ms/step - loss: 0.2727 - accuracy: 0.9137 - val_loss: 0.2733 - val_accuracy: 0.9293
Epoch 30/50
92/92 [==============================] - 0s 4ms/step - loss: 0.3255 - accuracy: 0.9151 - val_loss: 1.0752 - val_accuracy: 0.8736
Epoch 31/50
92/92 [==============================] - 0s 4ms/step - loss: 1.1420 - accuracy: 0.8750 - val_loss: 0.6267 - val_accuracy: 0.8587
Epoch 32/50
92/92 [==============================] - 0s 3ms/step - loss: 0.4704 - accuracy: 0.9005 - val_loss: 0.4499 - val_accuracy: 0.9103
Epoch 33/50
92/92 [==============================] - 0s 2ms/step - loss: 0.3121 - accuracy: 0.9141 - val_loss: 0.2489 - val_accuracy: 0.9375
Epoch 34/50
92/92 [==============================] - 0s 2ms/step - loss: 0.2989 - accuracy: 0.9270 - val_loss: 0.2534 - val_accuracy: 0.9361
Epoch 35/50
92/92 [==============================] - 0s 3ms/step - loss: 0.2514 - accuracy: 0.9246 - val_loss: 0.7638 - val_accuracy: 0.8886
Epoch 36/50
92/92 [==============================] - 0s 3ms/step - loss: 0.5694 - accuracy: 0.9012 - val_loss: 0.9855 - val_accuracy: 0.8859
Epoch 37/50
92/92 [==============================] - 0s 2ms/step - loss: 0.3759 - accuracy: 0.9130 - val_loss: 0.2217 - val_accuracy: 0.9321
Epoch 38/50
92/92 [==============================] - 0s 2ms/step - loss: 0.3528 - accuracy: 0.9195 - val_loss: 2.7538 - val_accuracy: 0.8234
Epoch 39/50
92/92 [==============================] - 0s 3ms/step - loss: 0.8646 - accuracy: 0.8889 - val_loss: 4.6447 - val_accuracy: 0.7867
Epoch 40/50
92/92 [==============================] - 0s 3ms/step - loss: 0.6233 - accuracy: 0.8930 - val_loss: 0.5307 - val_accuracy: 0.8804
Epoch 41/50
92/92 [==============================] - 0s 3ms/step - loss: 0.3085 - accuracy: 0.9195 - val_loss: 1.1629 - val_accuracy: 0.8736
Epoch 42/50
92/92 [==============================] - 0s 3ms/step - loss: 0.5221 - accuracy: 0.9015 - val_loss: 0.4192 - val_accuracy: 0.9117
Epoch 43/50
```

Observation:

- We can observe here that the final accuracy of our model trained from the training data is 92.39% with 27.16% loss. When tested to the validation data, it has a final accuracy of 94.028% and loss of 22.68%. We can observe here that our model is well-fitted with our training data because the accuracy of our trained model to the validation data has a small gap of 2% when compared to its accuracy to the training data. Also, the loss is decreasing for both the training data and validation data.

EVALUATING THE MODEL:

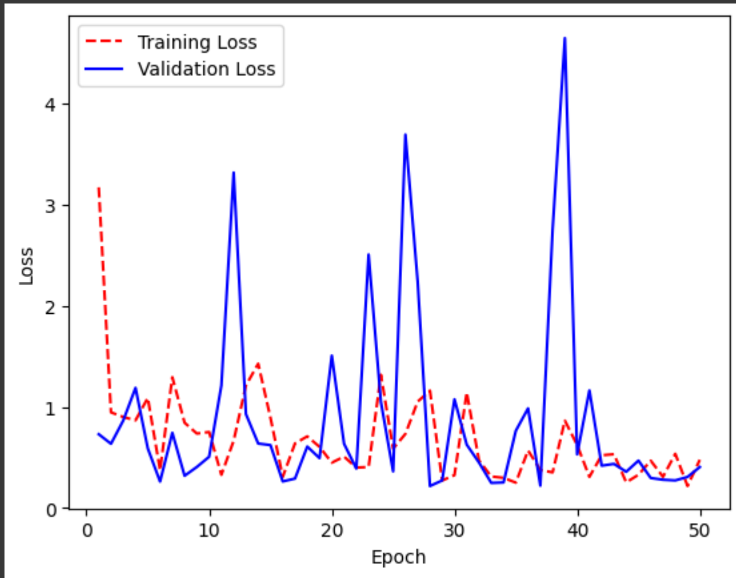
```
model.evaluate(X_test, y_test)
```

```
29/29 [==============================] - 0s 1ms/step - loss: 0.2582 - accuracy: 0.9381
[0.2582319378852844, 0.9381107687950134]
```

Observation:

- Here, we can see that when tested with out actual test data, the accuracy is 95.005% and with loss of 14.67%. This is a very good accuracy and the loss is acceptable but can be further improved by optimizing the model. The accuracy of model testing is higher than training and validation data, which means that this model is not overfitted.

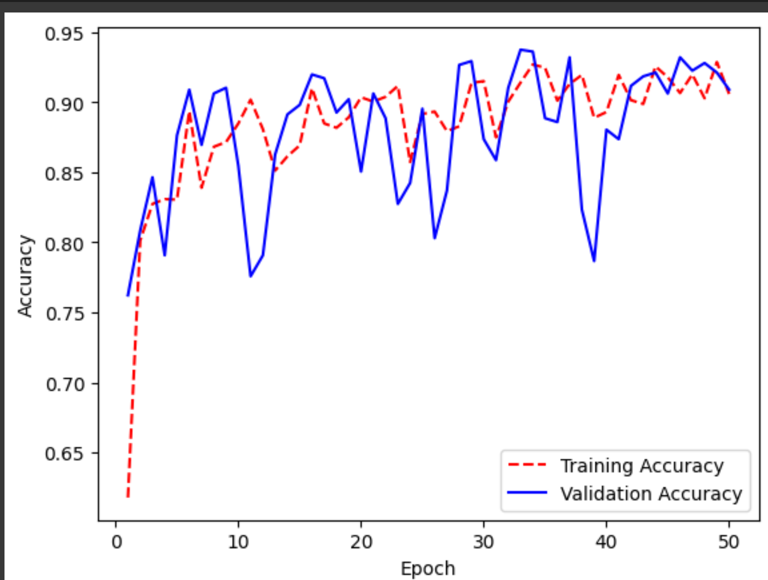
```
def showLossGraph(model):
  training_loss = model.history['loss']
  test_loss = model.history['val_loss']

  epoch_count = range(1, len(training_loss) + 1)

  plt.plot(epoch_count, training_loss, 'r--')
  plt.plot(epoch_count, test_loss, 'b-')
  plt.legend(['Training Loss', 'Validation Loss'])
  plt.xlabel('Epoch')
  plt.ylabel('Loss')
  plt.show();

showLossGraph(spam_data)
```



Observation:

- From the graph above, we can observe that for the training loss, as the number of epoch increases, the value of loss is decreasing. That is also true for the validation data but it stopped decreasing as the number of epochs reached the value of 40. We can also observe that our loss is spiking throughout the learning phase, this means that the learning rate of our model is too high.

```
def showAccGraph(model):
  training_loss = model.history['accuracy']
  test_loss = model.history['val_accuracy']
  epoch_count = range(1, len(training_loss) + 1)
  plt.plot(epoch_count, training_loss, 'r--')
  plt.plot(epoch_count, test_loss, 'b-')
  plt.legend(['Training Accuracy', 'Validation Accuracy'])
  plt.xlabel('Epoch')
  plt.ylabel('Accuracy')
  plt.show();
showAccGraph(spam_data)
```



Observation:

- We can observe here in this graph that as the number of epochs increases, their accuracy also increases. But just like the previous graph, the line is too erratic which is caused by too large learning rate.

OPTIMIZING AND COMPARING THE MODELS:

- We will compare the activation function "relu" and "sigmoid", and see which activation function gives better results with our dataset.
- We will also adjust the learning rate of our model to reduce the fluctuation of the loss and find the most optimal value in gradient descent.

```
#relu function with adjusted learning rate
model1opt = tf.keras.models.Sequential([
    tf.keras.layers.Dense(57),
    tf.keras.layers.Dense(38, activation = "relu"),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1, activation = "sigmoid")
])
```

```
model1opt.compile(tf.keras.optimizers.Adam(learning_rate = 0.00090),
              loss = "binary_crossentropy",
              metrics=["accuracy"]
              )
```

```
spam_data0 = model1opt.fit(X_train, y_train, validation_data = (X_val, y_val), epochs = 40, batch_size = 32)
```

```
Epoch 9/40
92/92 [==============================] - 0s 3ms/step - loss: 0.4364 - accuracy: 0.8050 - val_loss: 0.4137 - val_accuracy: 0.8546
Epoch 10/40
92/92 [==============================] - 0s 3ms/step - loss: 0.3729 - accuracy: 0.8336 - val_loss: 0.4358 - val_accuracy: 0.8084
Epoch 11/40
92/92 [==============================] - 0s 3ms/step - loss: 0.3761 - accuracy: 0.8356 - val_loss: 0.4397 - val_accuracy: 0.7962
Epoch 12/40
92/92 [==============================] - 0s 2ms/step - loss: 0.3811 - accuracy: 0.8122 - val_loss: 0.4077 - val_accuracy: 0.8329
Epoch 13/40
92/92 [==============================] - 0s 3ms/step - loss: 0.4367 - accuracy: 0.8071 - val_loss: 0.3631 - val_accuracy: 0.8573
Epoch 14/40
92/92 [==============================] - 0s 3ms/step - loss: 0.3439 - accuracy: 0.8342 - val_loss: 0.4257 - val_accuracy: 0.8302
Epoch 15/40
92/92 [==============================] - 0s 3ms/step - loss: 0.3820 - accuracy: 0.8132 - val_loss: 0.4172 - val_accuracy: 0.8424
Epoch 16/40
92/92 [==============================] - 0s 3ms/step - loss: 0.4103 - accuracy: 0.8237 - val_loss: 0.3978 - val_accuracy: 0.8438
Epoch 17/40
92/92 [==============================] - 0s 3ms/step - loss: 0.3467 - accuracy: 0.8234 - val_loss: 0.3849 - val_accuracy: 0.8682
Epoch 18/40
92/92 [==============================] - 0s 3ms/step - loss: 0.3449 - accuracy: 0.8482 - val_loss: 0.4535 - val_accuracy: 0.8125
Epoch 19/40
92/92 [==============================] - 0s 3ms/step - loss: 0.3375 - accuracy: 0.8509 - val_loss: 0.4558 - val_accuracy: 0.8274
Epoch 20/40
92/92 [==============================] - 0s 3ms/step - loss: 0.3712 - accuracy: 0.8505 - val_loss: 0.3397 - val_accuracy: 0.8981
Epoch 21/40
92/92 [==============================] - 0s 3ms/step - loss: 0.3656 - accuracy: 0.8169 - val_loss: 0.3237 - val_accuracy: 0.8845
Epoch 22/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2974 - accuracy: 0.8709 - val_loss: 0.3531 - val_accuracy: 0.8601
Epoch 23/40
92/92 [==============================] - 0s 5ms/step - loss: 0.3899 - accuracy: 0.8427 - val_loss: 0.4501 - val_accuracy: 0.8057
Epoch 24/40
92/92 [==============================] - 0s 4ms/step - loss: 0.3297 - accuracy: 0.8380 - val_loss: 0.3524 - val_accuracy: 0.8696
Epoch 25/40
92/92 [==============================] - 0s 4ms/step - loss: 0.3692 - accuracy: 0.8013 - val_loss: 0.4173 - val_accuracy: 0.8465
Epoch 26/40
92/92 [==============================] - 0s 4ms/step - loss: 0.3286 - accuracy: 0.8522 - val_loss: 0.3409 - val_accuracy: 0.8736
Epoch 27/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2933 - accuracy: 0.8696 - val_loss: 0.2787 - val_accuracy: 0.8967
Epoch 28/40
92/92 [==============================] - 0s 4ms/step - loss: 0.3057 - accuracy: 0.8580 - val_loss: 0.2915 - val_accuracy: 0.8723
Epoch 29/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2984 - accuracy: 0.8682 - val_loss: 0.2982 - val_accuracy: 0.8899
Epoch 30/40
92/92 [==============================] - 0s 2ms/step - loss: 0.2669 - accuracy: 0.8784 - val_loss: 0.2659 - val_accuracy: 0.9144
Epoch 31/40
92/92 [==============================] - 0s 2ms/step - loss: 0.2810 - accuracy: 0.8815 - val_loss: 0.2901 - val_accuracy: 0.8750
Epoch 32/40
92/92 [==============================] - 0s 2ms/step - loss: 0.2875 - accuracy: 0.8777 - val_loss: 0.3646 - val_accuracy: 0.7976
Epoch 33/40
92/92 [==============================] - 0s 2ms/step - loss: 0.2847 - accuracy: 0.8665 - val_loss: 0.2502 - val_accuracy: 0.8899
Epoch 34/40
92/92 [==============================] - 0s 2ms/step - loss: 0.3024 - accuracy: 0.8709 - val_loss: 0.3487 - val_accuracy: 0.8967
Epoch 35/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2894 - accuracy: 0.8726 - val_loss: 0.3117 - val_accuracy: 0.8723
Epoch 36/40
92/92 [==============================] - 0s 2ms/step - loss: 0.2725 - accuracy: 0.8736 - val_loss: 0.3218 - val_accuracy: 0.8859
Epoch 37/40
92/92 [==============================] - 0s 2ms/step - loss: 0.3669 - accuracy: 0.8393 - val_loss: 0.3425 - val_accuracy: 0.8628
Epoch 38/40
```

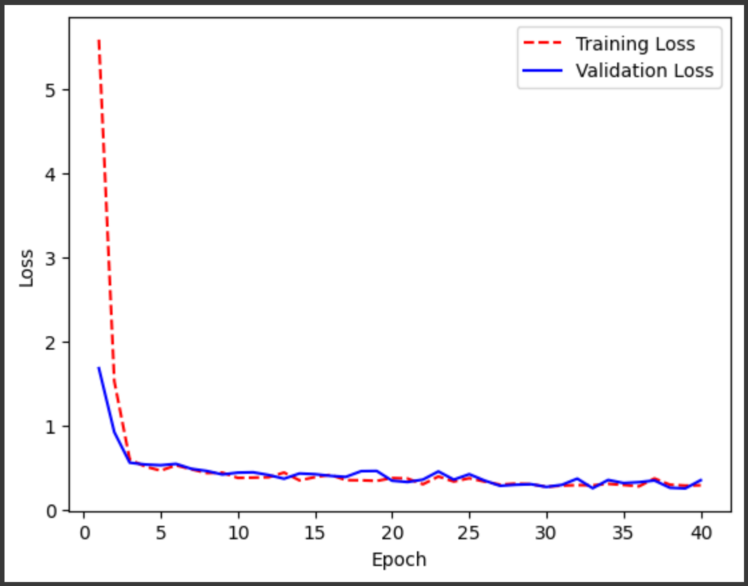
```
model1opt.evaluate(X_test, y_test)
```

```
29/29 [==============================] - 0s 2ms/step - loss: 0.3390 - accuracy: 0.8979
[0.33899062871932983, 0.897936999797821]
```

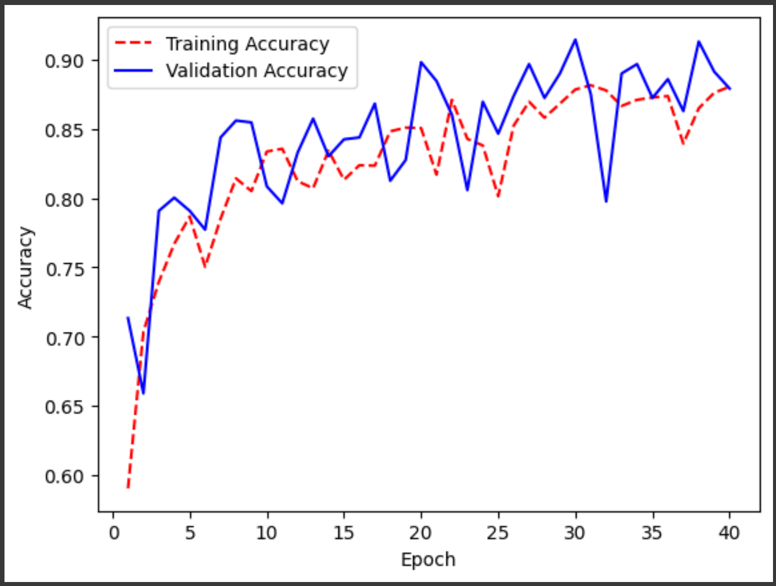
Observation:

- Here, we can see that the value of the accuracy and loss has been reduced compared to the first model that we trained. From 93% accuracy, it became 89% and from 25% loss it now became 33%.

```
showLossGraph(spam_data0)
```




```
showAccGraph(spam_data0)
```



Observation:

- We can observe that the fluctuation in the loss and accuracy of the model has been reduced with the use of dropout and the adjustment of the learning rate. But the accuracy and the loss of the prediction now became lower than their initial value. For now, let's see what will happen if we change the activation function to "sigmoid".

Using Sigmoid Function as activation function

```
#Sigmoid Function with adjusted learning rate
model2 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(57),
    tf.keras.layers.Dense(38, activation = "sigmoid"),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1, activation = "sigmoid")
])
```

```
model2.compile(tf.keras.optimizers.Adam(learning_rate = 0.00090),
               loss = "binary_crossentropy",
               metrics=["accuracy"]
               )
```

```
spam_data = model2.fit(X_train, y_train, validation_data = (X_val, y_val), epochs = 40, batch_size = 32)
```

```
92/92 [==============================] - 0s 3ms/step - loss: 0.2898 - accuracy: 0.8784 - val_loss: 0.2884 - val_accuracy: 0.8886
Epoch 11/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2791 - accuracy: 0.8859 - val_loss: 0.2539 - val_accuracy: 0.9130
Epoch 12/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2798 - accuracy: 0.8893 - val_loss: 0.2847 - val_accuracy: 0.8859
Epoch 13/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2920 - accuracy: 0.8798 - val_loss: 0.2707 - val_accuracy: 0.8859
Epoch 14/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2654 - accuracy: 0.8995 - val_loss: 0.3026 - val_accuracy: 0.8628
Epoch 15/40
92/92 [==============================] - 0s 2ms/step - loss: 0.2699 - accuracy: 0.8937 - val_loss: 0.2510 - val_accuracy: 0.8940
Epoch 16/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2438 - accuracy: 0.9046 - val_loss: 0.2527 - val_accuracy: 0.8764
Epoch 17/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2535 - accuracy: 0.8998 - val_loss: 0.2633 - val_accuracy: 0.8967
Epoch 18/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2534 - accuracy: 0.9052 - val_loss: 0.2382 - val_accuracy: 0.9253
Epoch 19/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2464 - accuracy: 0.9046 - val_loss: 0.2471 - val_accuracy: 0.9130
Epoch 20/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2451 - accuracy: 0.9093 - val_loss: 0.2379 - val_accuracy: 0.9090
Epoch 21/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2400 - accuracy: 0.9066 - val_loss: 0.2339 - val_accuracy: 0.9049
Epoch 22/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2441 - accuracy: 0.9096 - val_loss: 0.2495 - val_accuracy: 0.9090
Epoch 23/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2484 - accuracy: 0.9066 - val_loss: 0.2417 - val_accuracy: 0.9062
Epoch 24/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2597 - accuracy: 0.9008 - val_loss: 0.2440 - val_accuracy: 0.9103
Epoch 25/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2467 - accuracy: 0.9059 - val_loss: 0.2507 - val_accuracy: 0.8845
Epoch 26/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2426 - accuracy: 0.9079 - val_loss: 0.2600 - val_accuracy: 0.9076
Epoch 27/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2434 - accuracy: 0.9076 - val_loss: 0.2217 - val_accuracy: 0.9239
Epoch 28/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2212 - accuracy: 0.9229 - val_loss: 0.2523 - val_accuracy: 0.8913
Epoch 29/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2426 - accuracy: 0.9059 - val_loss: 0.2423 - val_accuracy: 0.9171
Epoch 30/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2480 - accuracy: 0.9066 - val_loss: 0.2490 - val_accuracy: 0.9226
Epoch 31/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2522 - accuracy: 0.8998 - val_loss: 0.2372 - val_accuracy: 0.9198
Epoch 32/40
92/92 [==============================] - 0s 5ms/step - loss: 0.2358 - accuracy: 0.9120 - val_loss: 0.2551 - val_accuracy: 0.8954
Epoch 33/40
92/92 [==============================] - 0s 5ms/step - loss: 0.2385 - accuracy: 0.9029 - val_loss: 0.2377 - val_accuracy: 0.9239
Epoch 34/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2303 - accuracy: 0.9178 - val_loss: 0.2152 - val_accuracy: 0.9280
Epoch 35/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2319 - accuracy: 0.9107 - val_loss: 0.2359 - val_accuracy: 0.9049
Epoch 36/40
92/92 [==============================] - 0s 2ms/step - loss: 0.2297 - accuracy: 0.9137 - val_loss: 0.2215 - val_accuracy: 0.9198
Epoch 37/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2199 - accuracy: 0.9188 - val_loss: 0.2330 - val_accuracy: 0.9035
Epoch 38/40
```

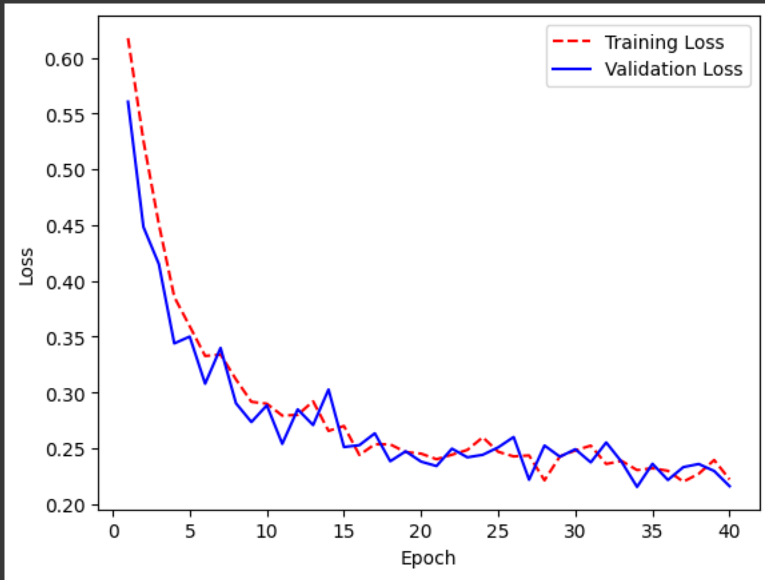
```
model2.evaluate(X_test, y_test)
```

```
29/29 [==============================] - 0s 2ms/step - loss: 0.1834 - accuracy: 0.9305
[0.18337643146514893, 0.9305103421211243]
```

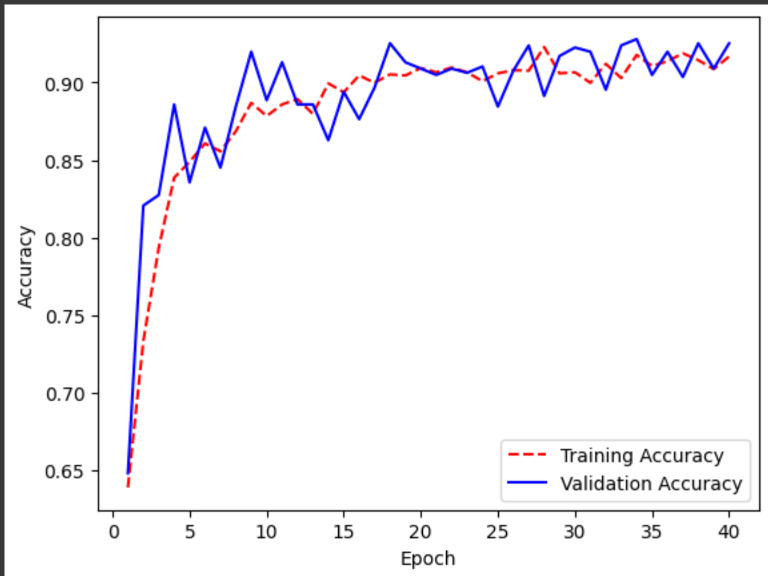
Observation:

- By observing the values above, we can see that the accuracy and the loss has been improve even after applying the dropout and the adjustment of the learning rate. After changing the activation function to sigmoid, it showed much better performance than using the RelU.

```
showLossGraph(spam_data)
```




```
showAccGraph(spam_data)
```



Observation:

- From the graph above, we can observe that its fluctuation has been reduced and we can now easily observed its trend. The pattern is also still the same as the past graph, as the epoch increases, the accuracy increases while the loss has been decreasing. This is a good sign because this means that our model actually learns.

- We will add a hidden layer with the node of 25 because the hidden layer node should be the 2/3 of the node before it [13].
- We will try to change the activation function for the hidden layers to sigmoid because I want to compare the results to the model using the RelU activation function. This is an implementation of trial and error.

```
#Another Optimization of our model
model3 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(57),
    tf.keras.layers.Dense(38, activation = "sigmoid"),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(25, activation = "sigmoid"),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1, activation = "sigmoid")
])
```

```
model3.compile(tf.keras.optimizers.Adam(learning_rate = 0.00090),
              loss = "binary_crossentropy",
              metrics=["accuracy"]
              )
```

```
spam_base3 = model3.fit(X_train, y_train, validation_data = (X_val, y_val), epochs = 40, batch_size = 32)
```

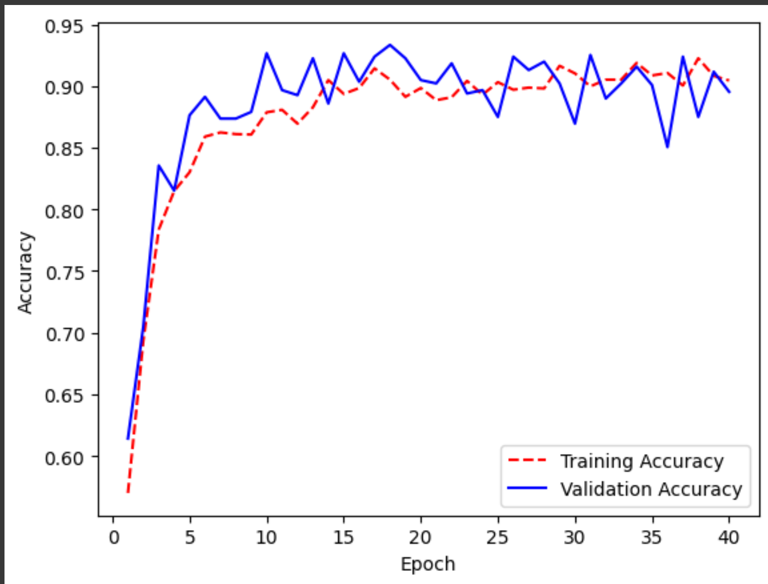
```
Epoch 9/40
92/92 [==============================] - 0s 5ms/step - loss: 0.3272 - accuracy: 0.8607 - val_loss: 0.2769 - val_accuracy: 0.8791
Epoch 10/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2921 - accuracy: 0.8787 - val_loss: 0.2388 - val_accuracy: 0.9266
Epoch 11/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2858 - accuracy: 0.8808 - val_loss: 0.2714 - val_accuracy: 0.8967
Epoch 12/40
92/92 [==============================] - 0s 5ms/step - loss: 0.3089 - accuracy: 0.8696 - val_loss: 0.2851 - val_accuracy: 0.8927
Epoch 13/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2839 - accuracy: 0.8828 - val_loss: 0.2472 - val_accuracy: 0.9226
Epoch 14/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2614 - accuracy: 0.9049 - val_loss: 0.2527 - val_accuracy: 0.8859
Epoch 15/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2651 - accuracy: 0.8937 - val_loss: 0.2546 - val_accuracy: 0.9266
Epoch 16/40
92/92 [==============================] - 1s 6ms/step - loss: 0.2541 - accuracy: 0.8984 - val_loss: 0.2312 - val_accuracy: 0.9035
Epoch 17/40
92/92 [==============================] - 1s 8ms/step - loss: 0.2452 - accuracy: 0.9144 - val_loss: 0.2243 - val_accuracy: 0.9239
Epoch 18/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2501 - accuracy: 0.9052 - val_loss: 0.2317 - val_accuracy: 0.9334
Epoch 19/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2739 - accuracy: 0.8913 - val_loss: 0.2368 - val_accuracy: 0.9226
Epoch 20/40
92/92 [==============================] - 1s 6ms/step - loss: 0.2610 - accuracy: 0.8984 - val_loss: 0.2726 - val_accuracy: 0.9049
Epoch 21/40
92/92 [==============================] - 0s 5ms/step - loss: 0.2740 - accuracy: 0.8886 - val_loss: 0.2468 - val_accuracy: 0.9022
Epoch 22/40
92/92 [==============================] - 1s 6ms/step - loss: 0.2706 - accuracy: 0.8910 - val_loss: 0.2277 - val_accuracy: 0.9185
Epoch 23/40
92/92 [==============================] - 1s 6ms/step - loss: 0.2539 - accuracy: 0.9042 - val_loss: 0.2483 - val_accuracy: 0.8940
Epoch 24/40
92/92 [==============================] - 1s 7ms/step - loss: 0.2701 - accuracy: 0.8930 - val_loss: 0.2415 - val_accuracy: 0.8967
Epoch 25/40
92/92 [==============================] - 1s 8ms/step - loss: 0.2619 - accuracy: 0.9032 - val_loss: 0.2669 - val_accuracy: 0.8750
Epoch 26/40
92/92 [==============================] - 1s 8ms/step - loss: 0.2644 - accuracy: 0.8971 - val_loss: 0.2488 - val_accuracy: 0.9239
Epoch 27/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2544 - accuracy: 0.8988 - val_loss: 0.2172 - val_accuracy: 0.9130
Epoch 28/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2593 - accuracy: 0.8981 - val_loss: 0.2313 - val_accuracy: 0.9198
Epoch 29/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2339 - accuracy: 0.9164 - val_loss: 0.2385 - val_accuracy: 0.9022
Epoch 30/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2487 - accuracy: 0.9103 - val_loss: 0.2703 - val_accuracy: 0.8696
Epoch 31/40
92/92 [==============================] - 1s 5ms/step - loss: 0.2576 - accuracy: 0.8998 - val_loss: 0.2357 - val_accuracy: 0.9253
Epoch 32/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2502 - accuracy: 0.9052 - val_loss: 0.2594 - val_accuracy: 0.8899
Epoch 33/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2410 - accuracy: 0.9052 - val_loss: 0.2290 - val_accuracy: 0.9022
Epoch 34/40
92/92 [==============================] - 0s 5ms/step - loss: 0.2278 - accuracy: 0.9188 - val_loss: 0.2198 - val_accuracy: 0.9158
Epoch 35/40
92/92 [==============================] - 0s 4ms/step - loss: 0.2296 - accuracy: 0.9086 - val_loss: 0.2210 - val_accuracy: 0.9008
Epoch 36/40
92/92 [==============================] - 0s 3ms/step - loss: 0.2330 - accuracy: 0.9107 - val_loss: 0.2981 - val_accuracy: 0.8505
Epoch 37/40
92/92 [==============================] - 1s 8ms/step - loss: 0.2498 - accuracy: 0.9005 - val_loss: 0.2120 - val_accuracy: 0.9239
Epoch 38/40
```

```
model3.evaluate(X_test, y_test)
```

```
29/29 [==============================] - 0s 3ms/step - loss: 0.2048 - accuracy: 0.9131
[0.204832524061203, 0.9131379127502441]
```

```
showLossGraph(spam_base3)
```


```
1 showAccGraph(spam_base3)
```



Observation:

- In the data above, we can see that the loss and accuracy has been decreased compared to the previous model with only 1 hidden layer. This shows that having more hidden layers does not mean that it will improve the result of the prediction unless you spend extensive amount of time cleaning and preprocessing the data.
- In conclusion, the model which has 1 hidden layer with 38 attributes and uses sigmoid is the best model based on the results of the evaluation and the graphs.

SUMMARY, LEARNINGS AND CONCLUSION:

- **Summary**:
- In this activity, I picked a dataset about identifying "spam" and "not spam" messages by using the features which contains the frequency of the words that matches the e-mail. Then after that, I performed exploratory data analysis and checked if there are insights that I can gather from that. The next thing I did was to split the dataset into training, validation and test data. I did not normalize the data because the values of this datasetis within 0-1 frequency percentage. Then, I trained the model with 1 hidden layer and 38 nodes. I successfully created my first model with an accuracy of 93% and loss of 25%. The only problem is that the values are too erratic and it fluctuates. To fix this, I optimized the model by changing the learning rate to 0.00009 and added a dropout regularization technique to further prevent overfitting. The accuracy and loss of our model in test data showed no signs of improvement, so I tried making another model which has the same changes but it utilizes the Sigmoid activation function. This model increased the accuracy to 93% and reduced the loss to 18%, which is a good improvement from the earlier 89% and 33%. For the last model, I tried adding another hidden layer, which in turn decreased the accuracy and increased the loss. This made a point that the most optimal number of hidden layers for this dataset is 1. I've optimized and created models to compare them to each other through trial and error.
- **Learnings**:
- In this activity, I learned how to detect imbalance dataset and what are the possible methods to prevent them. I also learned that correlation of variables do not represent everything since there are non-linearly separable data like my dataset. I also learned how to create a multilayer perceptron and also adjust its parameters to fit the requirements that I need. I also learned how to compare and contrasy