In [ ]:
```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
from keras.layers import Dense, Dropout, BatchNormalization
from keras.models import Sequential
from keras.callbacks import EarlyStopping, ReduceLROnPlateau
from keras.regularizers import l1_l2
import pandas as pd
import matplotlib.pyplot as plt
```

Using ANN

In [ ]:
```
# Loading the data
df = pd.read_csv('Phishing_Email.csv', header=None, names=['number', 'content', 'la

# Dropping the data where 'content' is N/A
df.dropna(subset=['content'], inplace=True)

# Only keeping the data where the label is 'Phishing Email' or 'Safe Email'
df = df[df['label'].isin(['Phishing Email', 'Safe Email'])]
df['content'] = df['content'].str.lower()

# Since the mail can be either spam or not, we use binary classification
df['label'] = df['label'].map({'Safe Email': 0, 'Phishing Email': 1}).astype('int32

# Displaying the first few values of the dataset
df.head()

# Splitting the content and labels into X and y
X = df['content']
y = df['label']

# Converting text into numerical data
vectorizer = TfidfVectorizer(max_features=10000)
X = vectorizer.fit_transform(X).toarray()

# Splitting training and testing data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

In [ ]:
```
# Building the model
ANNmodel = Sequential()
ANNmodel.add(Dense(1024, input_dim=X.shape[1], activation='relu', kernel_regularize
ANNmodel.add(BatchNormalization())
ANNmodel.add(Dropout(0.5))
ANNmodel.add(Dense(512, activation='relu', kernel_regularizer=l1_l2(l1=1e-5, l2=1e-
ANNmodel.add(Dense(256, activation='relu', kernel_regularizer=l1_l2(l1=1e-5, l2=1e-
ANNmodel.add(Dropout(0.5))
```

```
ANNmodel.add(Dense(128, activation='relu', kernel_regularizer=l1_l2(l1=1e-5, l2=1e-
ANNmodel.add(Dropout(0.5))
ANNmodel.add(Dense(64, activation='relu', kernel_regularizer=l1_l2(l1=1e-5, l2=1e-4
ANNmodel.add(Dropout(0.5))
ANNmodel.add(Dense(32, activation='relu', kernel_regularizer=l1_l2(l1=1e-5, l2=1e-4
ANNmodel.add(Dense(1, activation='sigmoid'))  # For binary classification

# Compiling the model
ANNmodel.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']

# Callbacks for early stopping and learning rate reduction
early_stopping = EarlyStopping(monitor='val_loss', patience=10, verbose=1)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.

# Training the model
history = ANNmodel.fit(X_train, y_train, epochs=60, batch_size=32, validation_split

# Evaluating the model
y_pred_prob = ANNmodel.predict(X_test)

# Converting probabilities to binary labels
y_pred = (y_pred_prob > 0.5).astype('int32')

# Ensuring y_test is an integer array
y_test = y_test.astype('int32')

print(classification_report(y_test, y_pred))
print("Accuracy:", accuracy_score(y_test, y_pred))
```

```
Epoch 1/60
373/373 [==============================] - 8s 13ms/step - loss: 1.0751 - accuracy:
0.9086 - val_loss: 0.9743 - val_accuracy: 0.9691 - lr: 0.0010
Epoch 2/60
373/373 [==============================] - 4s 11ms/step - loss: 0.7134 - accuracy:
0.9686 - val_loss: 0.6588 - val_accuracy: 0.9695 - lr: 0.0010
Epoch 3/60
373/373 [==============================] - 4s 11ms/step - loss: 0.5663 - accuracy:
0.9761 - val_loss: 0.5516 - val_accuracy: 0.9665 - lr: 0.0010
Epoch 4/60
373/373 [==============================] - 4s 11ms/step - loss: 0.4911 - accuracy:
0.9756 - val_loss: 0.5180 - val_accuracy: 0.9658 - lr: 0.0010
Epoch 5/60
373/373 [==============================] - 4s 11ms/step - loss: 0.4249 - accuracy:
0.9810 - val_loss: 0.4493 - val_accuracy: 0.9668 - lr: 0.0010
Epoch 6/60
373/373 [==============================] - 4s 11ms/step - loss: 0.3884 - accuracy:
0.9800 - val_loss: 0.3956 - val_accuracy: 0.9678 - lr: 0.0010
Epoch 7/60
373/373 [==============================] - 4s 11ms/step - loss: 0.3524 - accuracy:
0.9800 - val_loss: 0.3808 - val_accuracy: 0.9665 - lr: 0.0010
Epoch 8/60
373/373 [==============================] - 4s 11ms/step - loss: 0.3293 - accuracy:
0.9809 - val_loss: 0.3520 - val_accuracy: 0.9698 - lr: 0.0010
Epoch 9/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2873 - accuracy:
0.9826 - val_loss: 0.3260 - val_accuracy: 0.9695 - lr: 0.0010
Epoch 10/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2644 - accuracy:
0.9828 - val_loss: 0.3083 - val_accuracy: 0.9715 - lr: 0.0010
Epoch 11/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2647 - accuracy:
0.9820 - val_loss: 0.2961 - val_accuracy: 0.9712 - lr: 0.0010
Epoch 12/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2435 - accuracy:
0.9826 - val_loss: 0.2904 - val_accuracy: 0.9681 - lr: 0.0010
Epoch 13/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2483 - accuracy:
0.9821 - val_loss: 0.2770 - val_accuracy: 0.9691 - lr: 0.0010
Epoch 14/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2251 - accuracy:
0.9837 - val_loss: 0.2828 - val_accuracy: 0.9685 - lr: 0.0010
Epoch 15/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2401 - accuracy:
0.9829 - val_loss: 0.2770 - val_accuracy: 0.9718 - lr: 0.0010
Epoch 16/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2195 - accuracy:
0.9837 - val_loss: 0.2747 - val_accuracy: 0.9688 - lr: 0.0010
Epoch 17/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2212 - accuracy:
0.9828 - val_loss: 0.2524 - val_accuracy: 0.9712 - lr: 0.0010
Epoch 18/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2075 - accuracy:
0.9832 - val_loss: 0.2526 - val_accuracy: 0.9691 - lr: 0.0010
Epoch 19/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2156 - accuracy:
```

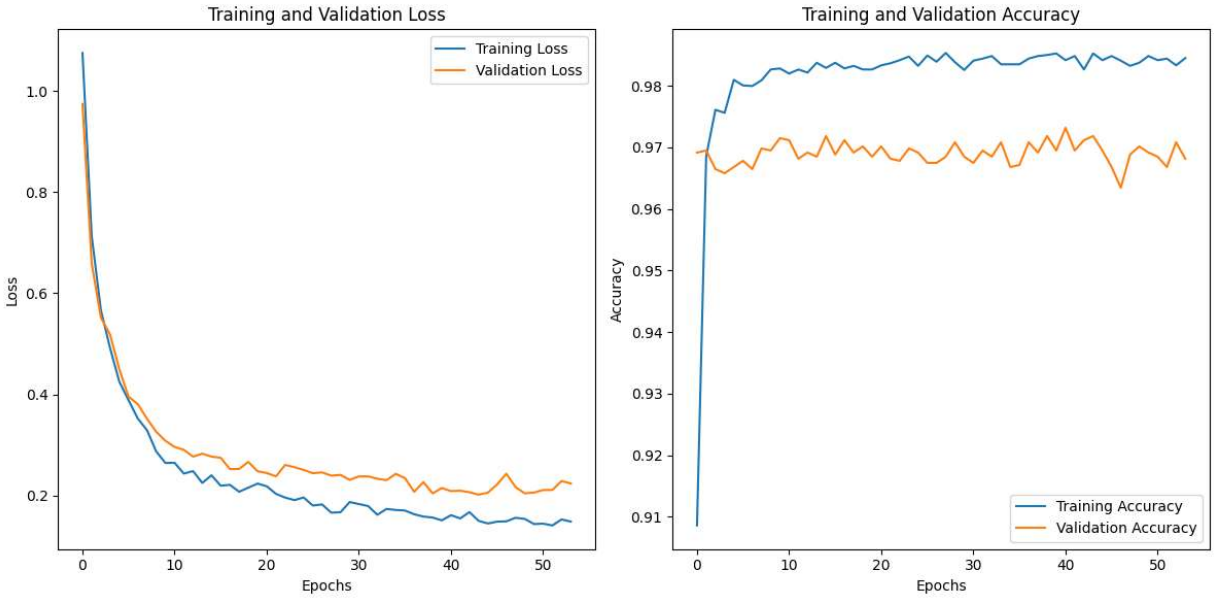
```
0.9826 - val_loss: 0.2665 - val_accuracy: 0.9702 - lr: 0.0010
Epoch 20/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2237 - accuracy:
0.9826 - val_loss: 0.2481 - val_accuracy: 0.9685 - lr: 0.0010
Epoch 21/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2183 - accuracy:
0.9833 - val_loss: 0.2446 - val_accuracy: 0.9702 - lr: 0.0010
Epoch 22/60
373/373 [==============================] - 4s 11ms/step - loss: 0.2033 - accuracy:
0.9836 - val_loss: 0.2380 - val_accuracy: 0.9681 - lr: 0.0010
Epoch 23/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1960 - accuracy:
0.9842 - val_loss: 0.2606 - val_accuracy: 0.9678 - lr: 0.0010
Epoch 24/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1910 - accuracy:
0.9847 - val_loss: 0.2561 - val_accuracy: 0.9698 - lr: 0.0010
Epoch 25/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1962 - accuracy:
0.9832 - val_loss: 0.2508 - val_accuracy: 0.9691 - lr: 0.0010
Epoch 26/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1803 - accuracy:
0.9849 - val_loss: 0.2442 - val_accuracy: 0.9675 - lr: 0.0010
Epoch 27/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1825 - accuracy:
0.9839 - val_loss: 0.2459 - val_accuracy: 0.9675 - lr: 0.0010
Epoch 28/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1662 - accuracy:
0.9853 - val_loss: 0.2393 - val_accuracy: 0.9685 - lr: 0.0010
Epoch 29/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1671 - accuracy:
0.9838 - val_loss: 0.2410 - val_accuracy: 0.9708 - lr: 0.0010
Epoch 30/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1872 - accuracy:
0.9826 - val_loss: 0.2310 - val_accuracy: 0.9685 - lr: 0.0010
Epoch 31/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1832 - accuracy:
0.9841 - val_loss: 0.2379 - val_accuracy: 0.9675 - lr: 0.0010
Epoch 32/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1793 - accuracy:
0.9844 - val_loss: 0.2379 - val_accuracy: 0.9695 - lr: 0.0010
Epoch 33/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1620 - accuracy:
0.9848 - val_loss: 0.2333 - val_accuracy: 0.9685 - lr: 0.0010
Epoch 34/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1736 - accuracy:
0.9835 - val_loss: 0.2305 - val_accuracy: 0.9708 - lr: 0.0010
Epoch 35/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1715 - accuracy:
0.9835 - val_loss: 0.2429 - val_accuracy: 0.9668 - lr: 0.0010
Epoch 36/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1705 - accuracy:
0.9835 - val_loss: 0.2346 - val_accuracy: 0.9671 - lr: 0.0010
Epoch 37/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1630 - accuracy:
0.9844 - val_loss: 0.2074 - val_accuracy: 0.9708 - lr: 0.0010
Epoch 38/60
```

```
373/373 [==============================] - 4s 11ms/step - loss: 0.1585 - accuracy:
0.9848 - val_loss: 0.2266 - val_accuracy: 0.9691 - lr: 0.0010
Epoch 39/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1566 - accuracy:
0.9850 - val_loss: 0.2042 - val_accuracy: 0.9718 - lr: 0.0010
Epoch 40/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1510 - accuracy:
0.9852 - val_loss: 0.2149 - val_accuracy: 0.9695 - lr: 0.0010
Epoch 41/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1612 - accuracy:
0.9842 - val_loss: 0.2088 - val_accuracy: 0.9732 - lr: 0.0010
Epoch 42/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1548 - accuracy:
0.9848 - val_loss: 0.2097 - val_accuracy: 0.9695 - lr: 0.0010
Epoch 43/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1673 - accuracy:
0.9826 - val_loss: 0.2065 - val_accuracy: 0.9712 - lr: 0.0010
Epoch 44/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1500 - accuracy:
0.9852 - val_loss: 0.2018 - val_accuracy: 0.9718 - lr: 0.0010
Epoch 45/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1448 - accuracy:
0.9842 - val_loss: 0.2058 - val_accuracy: 0.9695 - lr: 0.0010
Epoch 46/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1485 - accuracy:
0.9848 - val_loss: 0.2218 - val_accuracy: 0.9668 - lr: 0.0010
Epoch 47/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1490 - accuracy:
0.9841 - val_loss: 0.2433 - val_accuracy: 0.9634 - lr: 0.0010
Epoch 48/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1561 - accuracy:
0.9832 - val_loss: 0.2168 - val_accuracy: 0.9688 - lr: 0.0010
Epoch 49/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1539 - accuracy:
0.9837 - val_loss: 0.2045 - val_accuracy: 0.9702 - lr: 0.0010
Epoch 50/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1437 - accuracy:
0.9848 - val_loss: 0.2059 - val_accuracy: 0.9691 - lr: 0.0010
Epoch 51/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1444 - accuracy:
0.9842 - val_loss: 0.2109 - val_accuracy: 0.9685 - lr: 0.0010
Epoch 52/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1409 - accuracy:
0.9844 - val_loss: 0.2111 - val_accuracy: 0.9668 - lr: 0.0010
Epoch 53/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1528 - accuracy:
0.9833 - val_loss: 0.2287 - val_accuracy: 0.9708 - lr: 0.0010
Epoch 54/60
373/373 [==============================] - 4s 11ms/step - loss: 0.1485 - accuracy:
0.9845 - val_loss: 0.2239 - val_accuracy: 0.9681 - lr: 0.0010
Epoch 54: early stopping
117/117 [==============================] - 0s 2ms/step
              precision    recall  f1-score   support

           0       0.99      0.96      0.98      2209
           1       0.95      0.98      0.97      1518
```

```
              accuracy                           0.97      3727
             macro avg       0.97      0.97      0.97      3727
          weighted avg       0.97      0.97      0.97      3727

Accuracy: 0.9710222699221894
```

Plotting Training vs Validation Loss and Training vs Validation Accuracy

In [ ]:
```
plt.figure(figsize=(12, 6))
plt.subplot(1,2,1)
plt.plot(history.history['loss'], label="Training Loss")
plt.plot(history.history['val_loss'], label="Validation Loss")
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```



Using SVM

In [ ]:
```
# Loading the data
df = pd.read_csv('Phishing_Email.csv', header=None, names=['number', 'content', 'la

# Dropping the data where 'content' is N/A
df.dropna(subset=['content'], inplace=True)
```

```
# Only keeping the data where the label is 'Phishing Email' or 'Safe Email'
df = df[df['label'].isin(['Phishing Email', 'Safe Email'])]
df['content'] = df['content'].str.lower()


# Splitting the content and labels into X and y
X = df['content']
y = df['label']

# Converting text into numerical data
tfidf_vectorizer = TfidfVectorizer()
X = tfidf_vectorizer.fit_transform(X)

# Display the shape of the transformed data
X.shape

# Splitting the data (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

# Display the shapes of the split datasets to confirm the split
X_train.shape, X_test.shape, y_train.shape, y_test.shape

# Creating an SVM model with a linear kernel
SVMmodel = svm.SVC(kernel='linear')

# Training the model with the training data
SVMmodel.fit(X_train, y_train)

# Predicting on the training set
y_train_pred = SVMmodel.predict(X_train)

# Calculating the accuracy on the training set
training_accuracy = accuracy_score(y_train, y_train_pred)
print(f"Training Accuracy: {training_accuracy * 100:.2f}%")

# Classification report for the training set
print("Classification Report for Training Set:")
print(classification_report(y_train, y_train_pred))

# Predicting on the test set
y_pred = SVMmodel.predict(X_test)

# Evaluating the model
test_accuracy = accuracy_score(y_test, y_pred)

# Printing the accuracy
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")

# Classification report for the test set
print("Classification Report for Test Set:")
print(classification_report(y_test, y_pred))
```

```
Training Accuracy: 98.77%
Classification Report for Training Set:
                precision    recall  f1-score   support

Phishing Email       0.97      1.00      0.98      5794
    Safe Email       1.00      0.98      0.99      9113

      accuracy                           0.99     14907
     macro avg       0.98      0.99      0.99     14907
  weighted avg       0.99      0.99      0.99     14907

Test Accuracy: 97.69%
Classification Report for Test Set:
                precision    recall  f1-score   support

Phishing Email       0.95      0.99      0.97      1518
    Safe Email       0.99      0.97      0.98      2209

      accuracy                           0.98      3727
     macro avg       0.97      0.98      0.98      3727
  weighted avg       0.98      0.98      0.98      3727
```

Using KNN

In [ ]:
```
df = pd.read_csv('Phishing_Email.csv', header=0, names=['number', 'content', 'label

# Preprocessing the data:
# Handling the missing values by droping the rows where the email text is missing
df = df.dropna(subset=['content'])

# Text normalizing or lowercasing the content of the emails, helps maintain
# consistency with text and strings, as for example, "A" & "a" are considered
# different strings, so in order to be treated the same
df['content'] = df['content'].str.lower()
```

Vectorization

In [ ]:
```
# Vectorization converts text into numerical data, TF-IDF, which is
# Term Frequency-Inverse Document Frequency, gives a weight to each word in the
# document, highlighting the important words and diminishing, and this is crucial
# for text classification
vectorizer = TfidfVectorizer(max_features=1500, ngram_range=(1, 2))
X = vectorizer.fit_transform(df['content'])
y = df['label']
# Splitting the dataset into test and train data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

Identifying the Best Parameters

The main goal of the grid search is to identify the best combination of parameters that yield best performance for our KNN model. This is quite useful because it helps us save the time

of figuring the best combination of parameters that result in the highest accuracy by hand. GridSearch also uses cross-validation which sllits the data into 5 (`cv=5`) folds and performs training validation on these folds multiple times.

In [ ]:
```
# Note: this may take a lot of time and may require high computational capability
param_grid = {
    'n_neighbors': [3,4,5,6],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'cosine']
}
grid_search = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5, verbose=1, n_j
grid_search.fit(X_train, y_train)

# Best parameters
print(grid_search.best_params_)
```

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits
{'metric': 'cosine', 'n_neighbors': 6, 'weights': 'distance'}
```

Training and accuracy

Finally, we train our model according to the optimal parameters we computed and check its accuracy.

In [ ]:
```
knn = KNeighborsClassifier(n_neighbors=6, weights='distance', metric='cosine')
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print(classification_report(y_test, y_pred))
print("Accuracy:", accuracy_score(y_test, y_pred))
```

```
              precision    recall  f1-score   support

Phishing Email       0.92      0.93      0.92      1518
    Safe Email       0.95      0.94      0.95      2209

      accuracy                           0.94      3727
     macro avg       0.93      0.94      0.93      3727
  weighted avg       0.94      0.94      0.94      3727

Accuracy: 0.9369466058492085
```

We can see that we achieved an accuracy of 93.6%, which is very good, but the reason I haven't achieved a higher accuracy is due to the fact accuracy of k-Nearest Neighbors (kNN) in text classification tasks may sometimes be lower than other algorithms. This is due to a multiple of factors such as but not limited to, high dimensionality of text data, parameter sensitivity, sensitivity to noisy features ... etc.

Using Bayes Model

In [ ]:
```
# Loading the data
df = pd.read_csv('Phishing_Email.csv', header=None, names=['number', 'content', 'la
```

```
# Dropping the data where 'content' is N/A
df.dropna(subset=['content'], inplace=True)

# Only keeping the data where the label is 'Phishing Email' or 'Safe Email'
df = df[df['label'].isin(['Phishing Email', 'Safe Email'])]
df['content'] = df['content'].str.lower()

X = df['content']
y = df['label']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

vectorizer = TfidfVectorizer()
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

# Train the model using Multinomial Naive Bayes
model = MultinomialNB()
model.fit(X_train_tfidf, y_train)

# Evaluate the model on the test set
predictions = model.predict(X_test_tfidf)
print(classification_report(y_test, predictions))
print("Confusion Matrix:")
print(confusion_matrix(y_test, predictions))
```

```
                precision    recall  f1-score   support

Phishing Email       0.97      0.76      0.85      1518
    Safe Email       0.86      0.98      0.92      2209

      accuracy                           0.89      3727
     macro avg       0.91      0.87      0.88      3727
  weighted avg       0.90      0.89      0.89      3727

Confusion Matrix:
[[1154  364]
 [  38 2171]]
```

Using Logistic Regression

In [ ]:

```
# Loading the data
df = pd.read_csv('Phishing_Email.csv', header=None, names=['number', 'content', 'la

# Dropping the data where 'content' is N/A
df.dropna(subset=['content'], inplace=True)

# Only keeping the data where the label is 'Phishing Email' or 'Safe Email'
df = df[df['label'].isin(['Phishing Email', 'Safe Email'])]
df['content'] = df['content'].str.lower()

# Since the mail can be either spam or not, we use binary classification
df['label'] = df['label'].map({'Safe Email': 1, 'Phishing Email': 0}).astype('int32
```

```
# Separating the data
X = df['content']
Y = df['label']

# Splitting the data for training and splitting
X_train, X_test, Y_train, Y_test = train_test_split(X,Y,test_size=0.2,random_state=

# Converting the text email into numerical values so we can get the feature -> feat
feature_extraction = TfidfVectorizer(min_df=1, stop_words='english', lowercase=True
X_train_features = feature_extraction.fit_transform(X_train)
X_test_features = feature_extraction.transform(X_test)

# Creating the model
LRmodel = LogisticRegression()
LRmodel.fit(X_train_features,Y_train)
# Predictions and accuracy for training set
prediction_train = LRmodel.predict(X_train_features)
accuracy_train = accuracy_score(Y_train, prediction_train)
print("Training Accuracy:", accuracy_train)
print("Training Classification Report:")
print(classification_report(Y_train, prediction_train))

# Predictions and accuracy for test set
prediction_test = LRmodel.predict(X_test_features)
accuracy_test = accuracy_score(Y_test, prediction_test)
print("Test Accuracy:", accuracy_test)
print("Test Classification Report:")
print(classification_report(Y_test, prediction_test))
```

```
Training Accuracy: 0.9831622727577648
Training Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98      5811
           1       1.00      0.98      0.99      9096

    accuracy                           0.98     14907
   macro avg       0.98      0.98      0.98     14907
weighted avg       0.98      0.98      0.98     14907

Test Accuracy: 0.9720955191843306
Test Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.97      0.97      1501
           1       0.98      0.97      0.98      2226

    accuracy                           0.97      3727
   macro avg       0.97      0.97      0.97      3727
weighted avg       0.97      0.97      0.97      3727
```