Configuration Manual for Phishing

1 System Requirements

This whole project takes into the account three important steps,

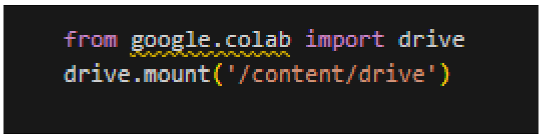
RAM: 16GB DDR2

OS: Windows 11 pro

Processor: i7 9th generation

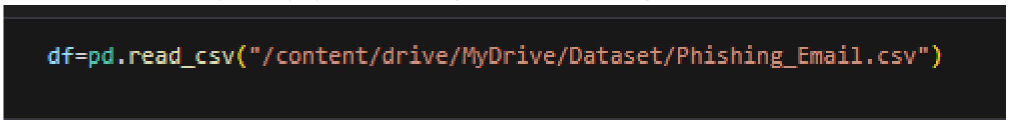
Technology required: Python, Anaconda, Spyder, Streamlit

2 Code execution



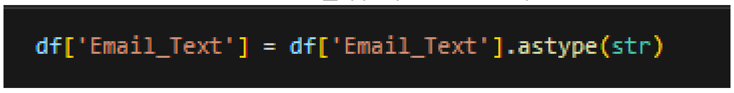
```
from google.colab import drive
drive.mount('/content/drive')
```

Figure1. This code logs into your Google Account and demonstrates how to mount your Google Drive into Colab; subsequently, you can drag files from Google Drive into the Colab notebook.



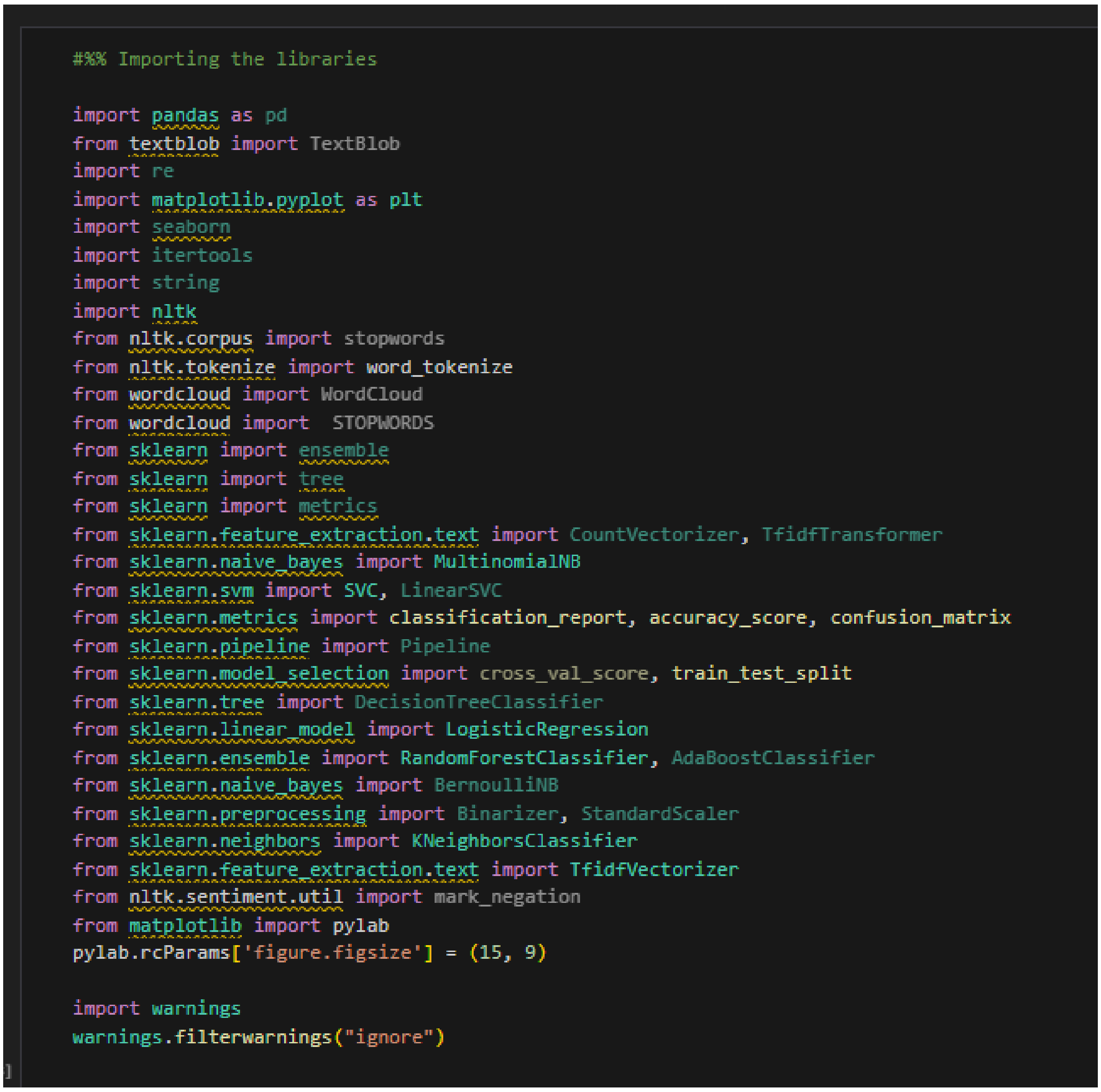
```
df=pd.read_csv("/content/drive/MyDrive/Dataset/Phishing_Email.csv")
```

Figure2. This code reads a CSV file that is named Phishing_Email. It then loads `data|filename:data. it moves the file named `(file_type|extension|date). csv` from Google Drive to a DataFrame called `df`.



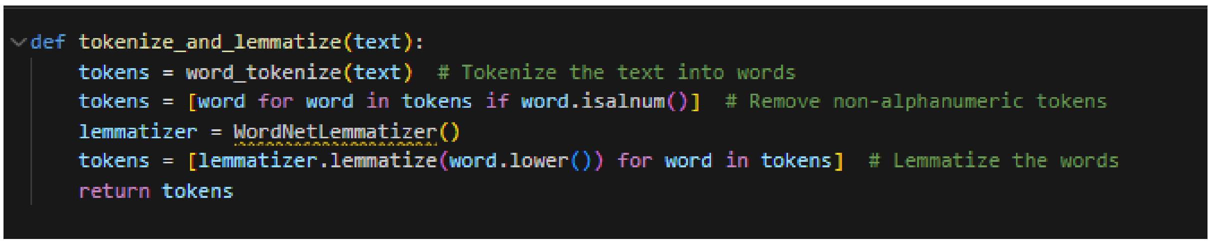
```
df['Email_Text'] = df['Email_Text'].astype(str)
```

Figure3. This code converts the 'Email_Text' as the column of the DataFrame 'df' as string data type.



```
#%% Importing the libraries

import pandas as pd
from textblob import TextBlob
import re
import matplotlib.pyplot as plt
import seaborn
import itertools
import string
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from wordcloud import WordCloud
from wordcloud import  STOPWORDS
from sklearn import ensemble
from sklearn import tree
from sklearn import metrics
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import SVC, LinearSVC
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import BernoulliNB
from sklearn.preprocessing import Binarizer, StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.sentiment.util import mark_negation
from matplotlib import pylab
pylab.rcParams['figure.figsize'] = (15, 9)

import warnings
warnings.filterwarnings("ignore")
```

Figure4. These are the statements that import several required libraries and modules that will be used in data manipulation, text preprocessing, machine learning, and data visualization. The imported libraries are pandas for data manipulation, textblob and nltk for text manipulation, sklearn for machine learning solution, and matplotlib, seaborn, and wordcloud for data visualization. Further, it sets the plot sizes by using the `matplotlib` and mutes all the warnings thrown by `matplotlib` to clean the output.



```
def tokenize_and_lemmatize(text):
    tokens = word_tokenize(text)  # Tokenize the text into words
    tokens = [word for word in tokens if word.isalnum()]  # Remove non-alphanumeric tokens
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word.lower()) for word in tokens]  # Lemmatize the words
    return tokens
```

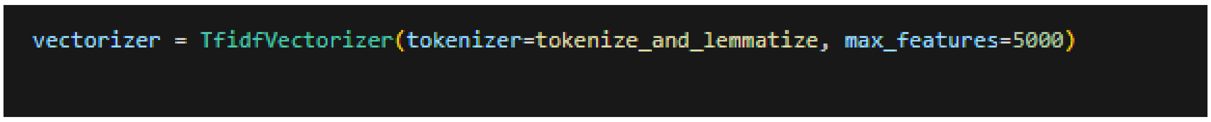
Figure5. The `tokenize_and_lemmatize` function processes a given text by:The `tokenize_and_lemmatize` function processes a given text by:

1. Resulting in tokenization of the text, which can be further defined as splitting the text into individual words.

2. Removing any non-alphanumeric tokens.

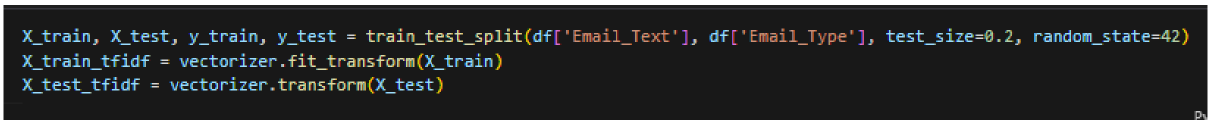
3. Lowercasing the remaining tokens and lemmatizing them which in other words means reducing more complex words to their base form.

The function returns words after preprocessing them. Otherwise, the function will return `None`, it means that the regular expression does not match.



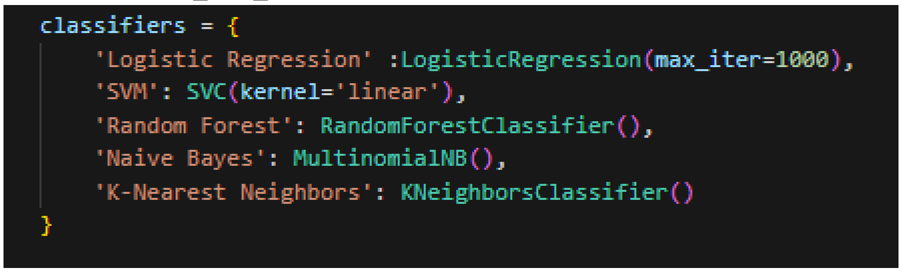
```
vectorizer = TfidfVectorizer(tokenizer=tokenize_and_lemmatize, max_features=5000)
```

Figure6. Here this code is separated by the custom tokenizer function and the maximum number of total features is defined as 5000 terms. The `TfidfVectorizer` fits our text data and then transforms it into matrix of TF-IDF values, relative to each word to document and to the entire text.



```
X_train, X_test, y_train, y_test = train_test_split(df['Email_Text'], df['Email_Type'], test_size=0.2, random_state=42)
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)
```

Figure7. This code includes the data split where the data splits are about 80% to 20% where the 80% is trained and the 20% is tested. It then fits the `TfidfVectorizer’ on the training data to transform the data into a TF-IDF matrix which is referred to as `X_train_tfidf’ and uses the same to transform the test data into `X_test_tfidf’.



```
classifiers = {
    'Logistic Regression' :LogisticRegression(max_iter=1000),
    'SVM': SVC(kernel='linear'),
    'Random Forest': RandomForestClassifier(),
    'Naive Bayes': MultinomialNB(),
    'K-Nearest Neighbors': KNeighborsClassifier()
}
```

Figure8. This code initializes a dictionary with various machine learning classifiers, each associated with its model:This code initializes a dictionary with various machine learning classifiers, each associated with its model:

- Logistic Regression: Set at a maximum of 1,000 iterations but; can be adjusted depending on the difficulty in solving a particular problem.

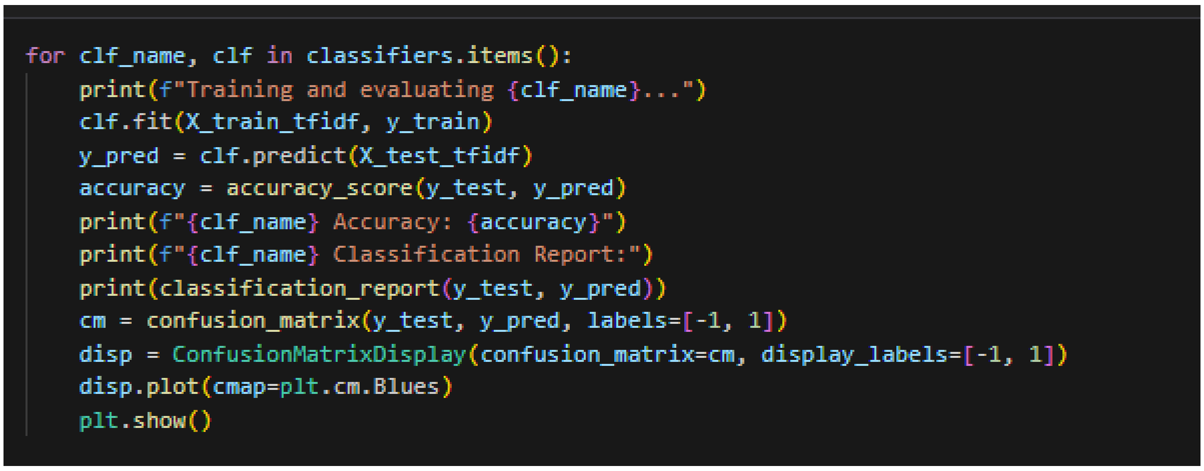
- SVM (Support Vector Machine): Falling from the linear kernel.

- Random Forest: Refers to a configuration which is comprehensible in any context by practically all technical fields.

- Naive Bayes: Defining the model of communication and using the Multinomial Naive Bayes approach.

- K-Nearest Neighbors: Standard process, that can be used with modifications.

In this dictionary, the key to the classifier's name contains the value that points to the model instance of the classifier.



```
for clf_name, clf in classifiers.items():
    print(f"Training and evaluating {clf_name}...")
    clf.fit(X_train_tfidf, y_train)
    y_pred = clf.predict(X_test_tfidf)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"{clf_name} Accuracy: {accuracy}")
    print(f"{clf_name} Classification Report:")
    print(classification_report(y_test, y_pred))
    cm = confusion_matrix(y_test, y_pred, labels=[-1, 1])
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[-1, 1])
    disp.plot(cmap=plt.cm.Blues)
    plt.show()
```

Figure10. This code trains and evaluates each classifier listed in the `classifiers` dictionary:This code trains and evaluates each classifier listed in the `classifiers` dictionary:

1. Training and Evaluation:

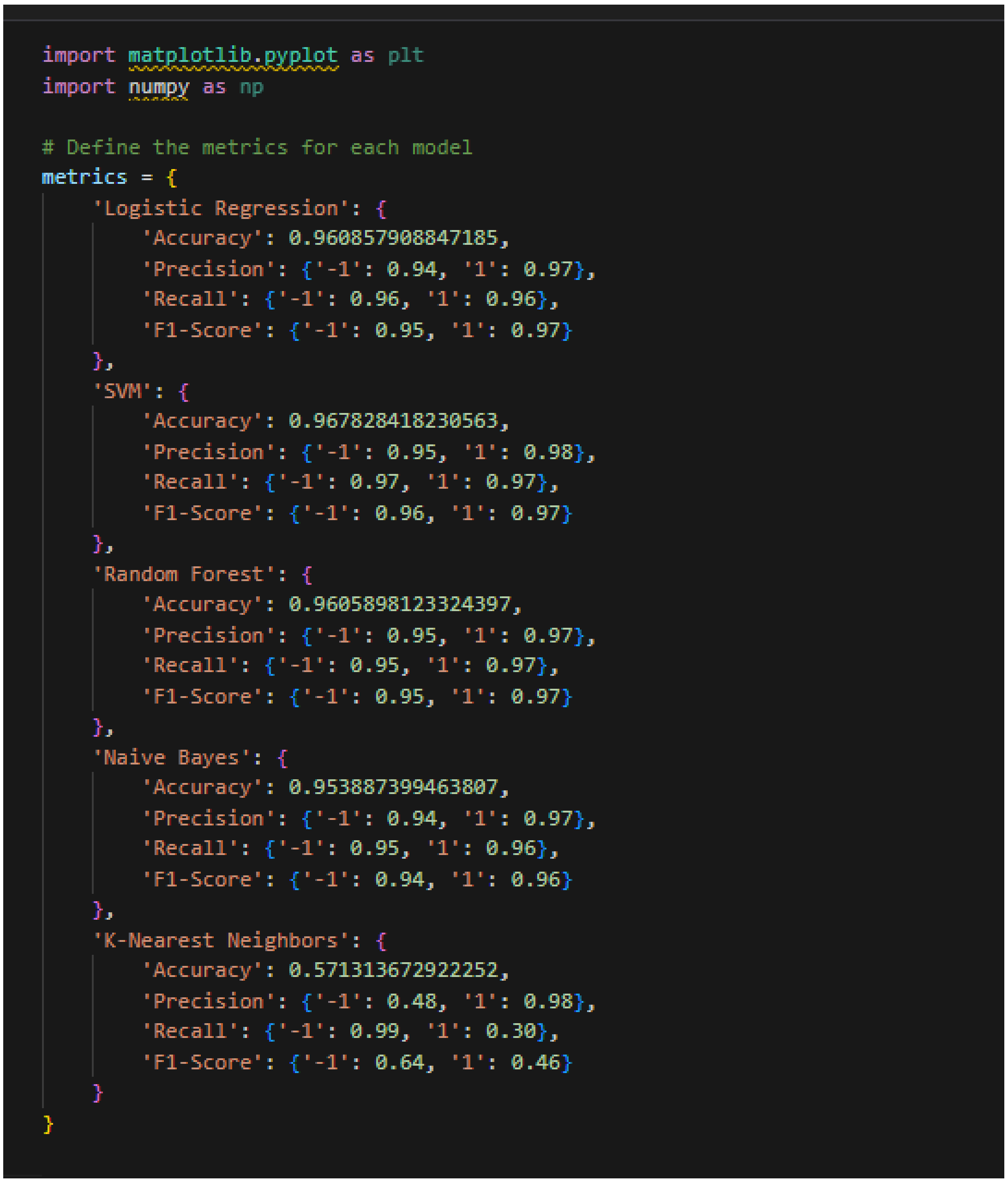
- Training: Superposition was made of the data used for the training of each model, namely the transition matrix of the TF-IDF vectors `X_train_tfidf` and the vector of the corresponding target values `y_train`.

- Prediction: The model predicts outcomes on the test data and shown in the above portion where `X_test_tfidf` represents the vectors for the test data.

- Accuracy: Calculates and prints the accuracy score between the predicted and the actual labels.

- Classification Report: Automates the production of a report that gives the precision, recall, and F1 score of each classifier.

- Confusion Matrix: Conduct metrics true positive and true negative with the help of the confusion matrix that is presented in the form of heatmap.



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

# Define the metrics for each model
metrics = {
    'Logistic Regression': {
        'Accuracy': 0.960857908847185,
        'Precision': {'-1': 0.94, '1': 0.97},
        'Recall': {'-1': 0.96, '1': 0.96},
        'F1-Score': {'-1': 0.95, '1': 0.97}
    },
    'SVM': {
        'Accuracy': 0.967828418230563,
        'Precision': {'-1': 0.95, '1': 0.98},
        'Recall': {'-1': 0.97, '1': 0.97},
        'F1-Score': {'-1': 0.96, '1': 0.97}
    },
    'Random Forest': {
        'Accuracy': 0.9605898123324397,
        'Precision': {'-1': 0.95, '1': 0.97},
        'Recall': {'-1': 0.95, '1': 0.97},
        'F1-Score': {'-1': 0.95, '1': 0.97}
    },
    'Naive Bayes': {
        'Accuracy': 0.953887399463807,
        'Precision': {'-1': 0.94, '1': 0.97},
        'Recall': {'-1': 0.95, '1': 0.96},
        'F1-Score': {'-1': 0.94, '1': 0.96}
    },
    'K-Nearest Neighbors': {
        'Accuracy': 0.571313672922252,
        'Precision': {'-1': 0.48, '1': 0.98},
        'Recall': {'-1': 0.99, '1': 0.30},
        'F1-Score': {'-1': 0.64, '1': 0.46}
    }
}
```

Figure11. This particular algorithm defines a dictionary named `metrics` which holds the performances of different machine learning models. For each model, the metrics include:For each model, the metrics include:

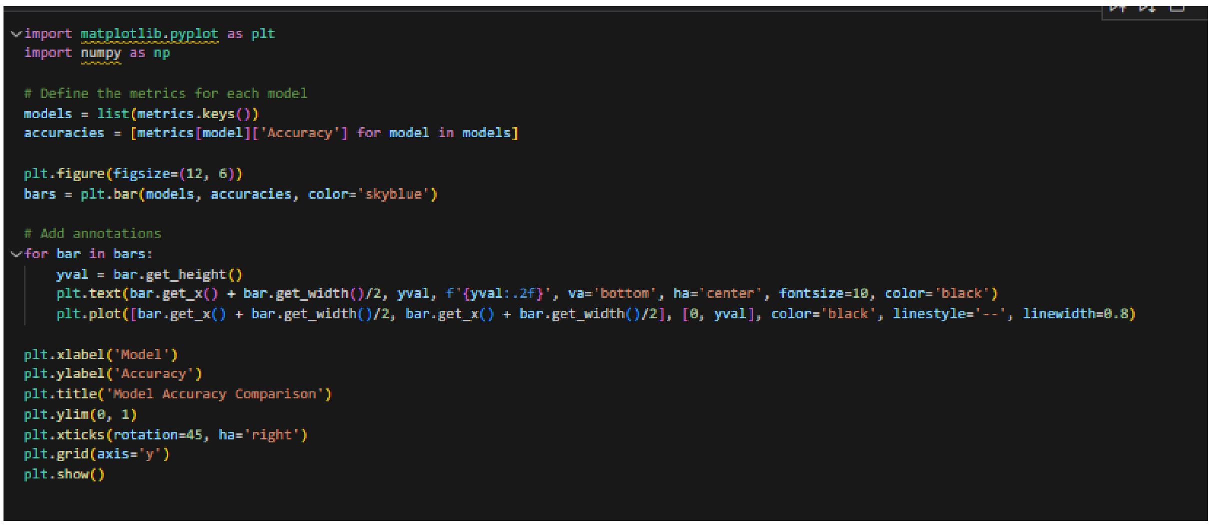
- Accuracy: The measure of accuracy of the totals of the model.

- Precision: This would provide the precision scores for each of the class '-1' and '1'.

- Recall: Remember the scores of each of the classes.

- F1-Score: Table of F1 scores for each of the classes.

These metrics help provide a clear and all rounded evaluation of each of the models.



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

# Define the metrics for each model
models = list(metrics.keys())
accuracies = [metrics[model]['Accuracy'] for model in models]

plt.figure(figsize=(12, 6))
bars = plt.bar(models, accuracies, color='skyblue')

# Add annotations
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    plt.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Model Accuracy Comparison')
plt.ylim(0, 1)
plt.xticks(rotation=45, ha='right')
plt.grid(axis='y')
plt.show()
```

Figure12. This code generates a bar chart to visualize the accuracy of various machine learning models:This code generates a bar chart to visualize the accuracy of various machine learning models:

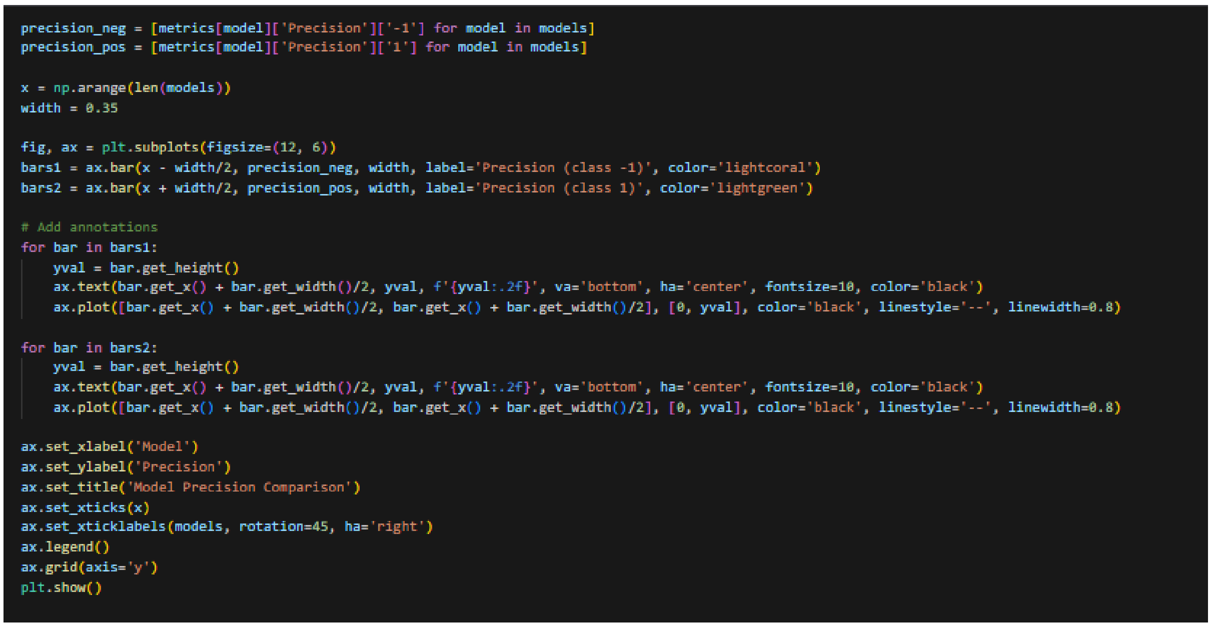
1. Data Preparation: Calls the `extract_model_accuracy()` function and passes the `metrics` dictionary as an argument; the function extracts only model names and their scores.

2. Plotting: Produces a bar chart with labels on the x-axis and the models on the y-axis, while the accuracy scores are on the y-axis.

3. Annotations: Plots accuracy values over each bar and makes dashed lines from top of the bar to the X-axis.

4. Formatting: This code sets up the labels, title, y axis limits and rotates the x axis labels by 90 for better reading.

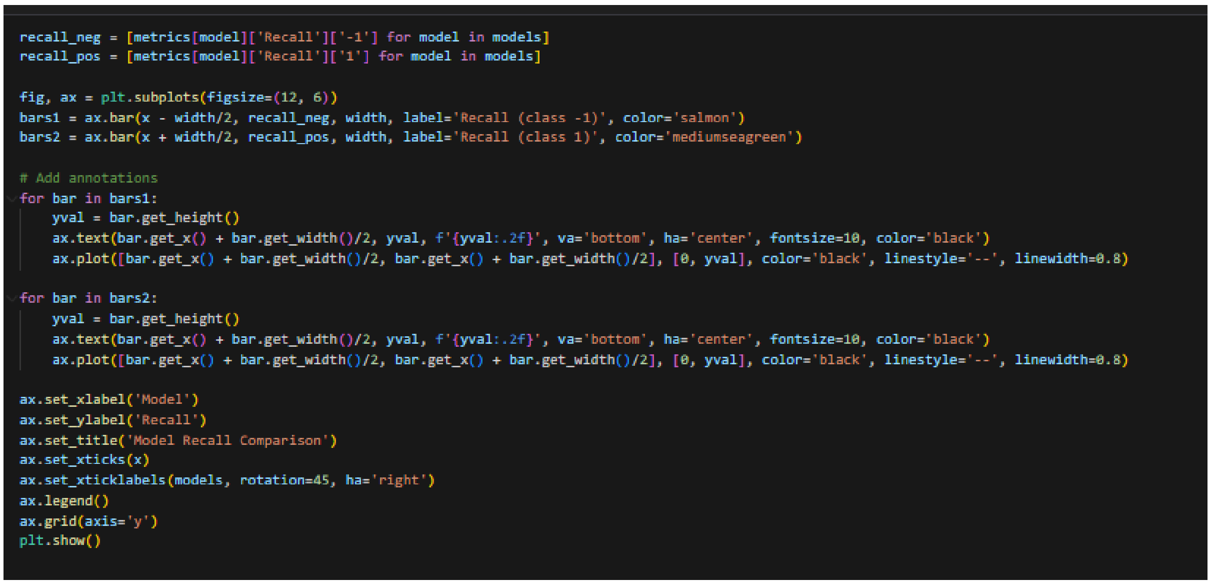
5. Display: Presents the bar chart with the possibility of comparing the corresponding elements and knowing which model is best among them.



```python
precision_neg = [metrics[model]['Precision']['-1'] for model in models]
precision_pos = [metrics[model]['Precision']['1'] for model in models]

x = np.arange(len(models))
width = 0.35

fig, ax = plt.subplots(figsize=(12, 6))
bars1 = ax.bar(x - width/2, precision_neg, width, label='Precision (class -1)', color='lightcoral')
bars2 = ax.bar(x + width/2, precision_pos, width, label='Precision (class 1)', color='lightgreen')

# Add annotations
for bar in bars1:
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    ax.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

for bar in bars2:
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    ax.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

ax.set_xlabel('Model')
ax.set_ylabel('Precision')
ax.set_title('Model Precision Comparison')
ax.set_xticks(x)
ax.set_xticklabels(models, rotation=45, ha='right')
ax.legend()
ax.grid(axis='y')
plt.show()
```

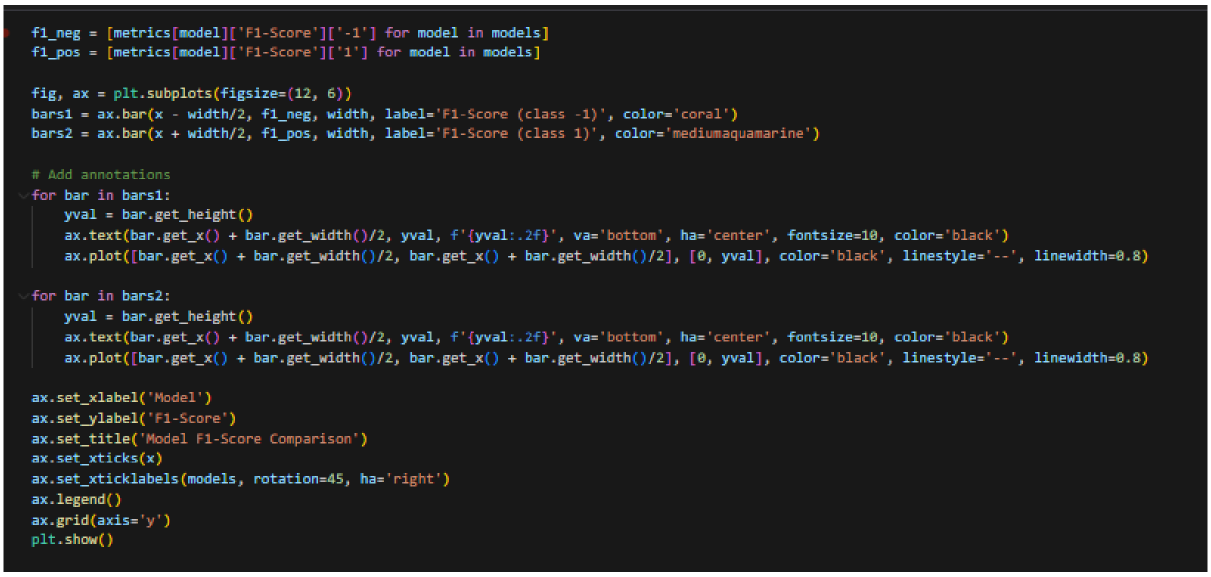
Figure13. The code implementation generates a bar chart that quantifies the precision of a number of models of two classes (-1 and 1). It compares the precision-curve ratings of each model side by

side while putting additional notes to qualify the curves. This chart proves useful in performance evaluation and comparing the models according to their precision of the two classes in the set.



```python
recall_neg = [metrics[model]['Recall']['-1'] for model in models]
recall_pos = [metrics[model]['Recall']['1'] for model in models]

fig, ax = plt.subplots(figsize=(12, 6))
bars1 = ax.bar(x - width/2, recall_neg, width, label='Recall (class -1)', color='salmon')
bars2 = ax.bar(x + width/2, recall_pos, width, label='Recall (class 1)', color='mediumseagreen')

# Add annotations
for bar in bars1:
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    ax.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

for bar in bars2:
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    ax.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

ax.set_xlabel('Model')
ax.set_ylabel('Recall')
ax.set_title('Model Recall Comparison')
ax.set_xticks(x)
ax.set_xticklabels(models, rotation=45, ha='right')
ax.legend()
ax.grid(axis='y')
plt.show()
```

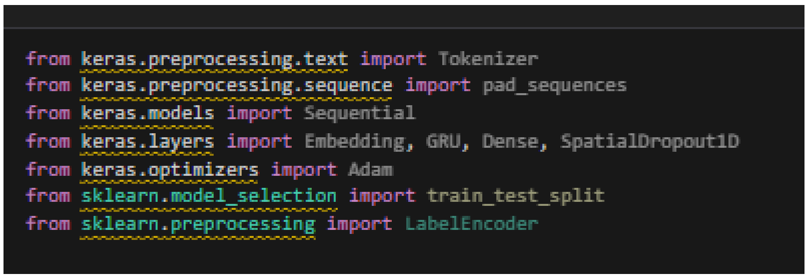
Figure14. The code produces a bar chart that depicts the recall of different models with regard to the two classes, namely, -1 and 1. It shows the recall scores of all the models and on the right axis to help understand every bar we have. This chart is ideal in presenting the extent to which the models can preserve examples of the two classes.



```python
f1_neg = [metrics[model]['F1-Score']['-1'] for model in models]
f1_pos = [metrics[model]['F1-Score']['1'] for model in models]

fig, ax = plt.subplots(figsize=(12, 6))
bars1 = ax.bar(x - width/2, f1_neg, width, label='F1-Score (class -1)', color='coral')
bars2 = ax.bar(x + width/2, f1_pos, width, label='F1-Score (class 1)', color='mediumaquamarine')

# Add annotations
for bar in bars1:
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    ax.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

for bar in bars2:
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    ax.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

ax.set_xlabel('Model')
ax.set_ylabel('F1-Score')
ax.set_title('Model F1-Score Comparison')
ax.set_xticks(x)
ax.set_xticklabels(models, rotation=45, ha='right')
ax.legend()
ax.grid(axis='y')
plt.show()
```

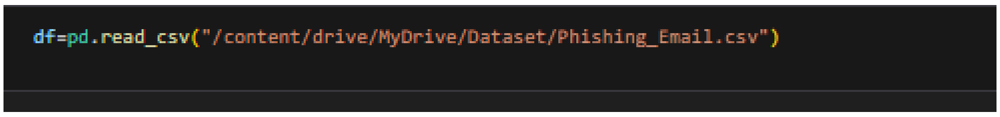
Figure15. The code then generates a bar chart that plots the F1 scores between two classes, namely -1 and 1, for the different models. It provides F1 scores of each model and annotations to properly link them with the corresponding models. It also assists in the evaluation of the region of interest of the Precision and Recall for each model of the two classes.

Phishing



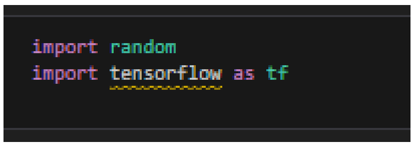
```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Embedding, GRU, Dense, SpatialDropout1D
from keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
```

Figure16. The code consists of data pre-processing and creation of a neural network through Keras. It involves identifying text data for reading the model, specifying a sequential model with the embedding layer, a GRU layer, and the dataset splitting, training, and optimizing processes.



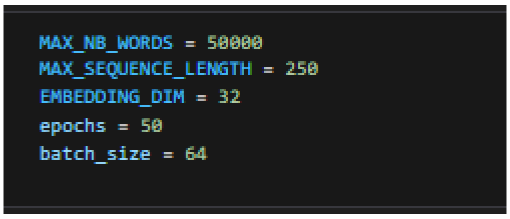
```
df=pd.read_csv("/content/drive/MyDrive/Dataset/Phishing_Email.csv")
```

Figure17. The code reads a text file with a name of Phishing_Email and an extension of csv. the ‘csv’ file from a given path on Google Drive and loads it into a pandas DataFrame called ‘df’.



```
import random
import tensorflow as tf
```

Figure18. The code starts with importing a random environmental library for the random numbers and importing tensorflow as ‘tf’ which is widely used for various machine learning and deep learning operations.



```
MAX_NB_WORDS = 50000
MAX_SEQUENCE_LENGTH = 250
EMBEDDING_DIM = 32
epochs = 50
batch_size = 64
```

Figure19. These hyperparameters are crucial for configuring a deep learning model:These hyperparameters are crucial for configuring a deep learning model:

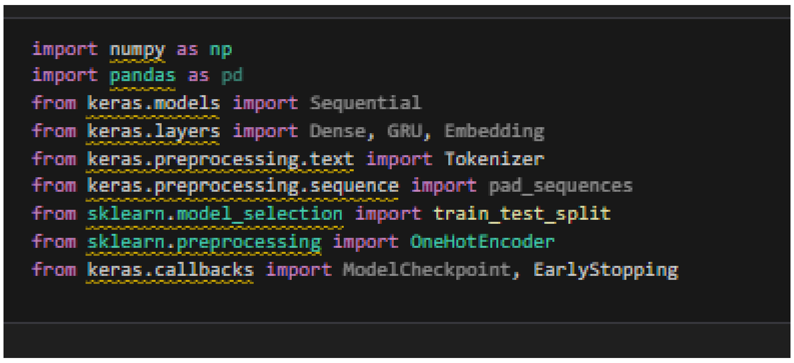
- `MAX_NB_WORDS`: This is the largest number of distinctive words that can be included when the tokens are being generated.

- `MAX_SEQUENCE_LENGTH`: The maximum size of input sequences including the padding.

- `EMBEDDING_DIM`: The size of the word vectors in the embedding layer with the three most common architectures of deep learning models.

- `epochs`: The number of epochs that was run through the full set of the data set in the training faze of the model.

- `batch_size`: The number of samples that go through the network before the weights of the model are adjusted.



```python
import numpy as np
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense, GRU, Embedding
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from keras.callbacks import ModelCheckpoint, EarlyStopping
```

Figure20. The code sets up a deep learning pipeline for text classification, including:The code sets up a deep learning pipeline for text classification, including:

1. Preprocessing:

- Tokenizer: Maps text into list of integers.

- pad_sequences: It is used to pad sequences to match the length of longest sequence in all sequences or truncate those that are longer than the longest.

2. Model Configuration:

- Sequential Model: Piles them up in a sequential manner.

- Embedding Layer: Maps integer sequences into rarely sparse, high-dimensional space.

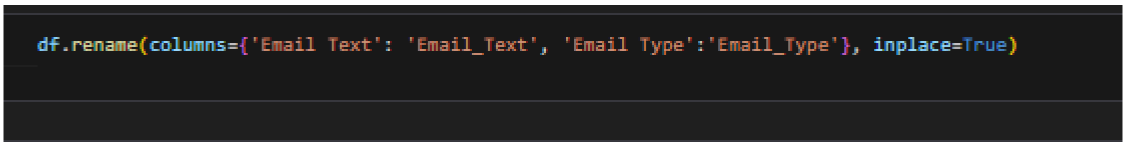
- GRU Layer: Responsible for processing sequential data with a combination of the GRU unit.

- Dense Layer: Fully connected layer final layer for classification into the appropriate class.

3. Training Setup:

- Callbacks: There shall be a use of early stopping and also model saving measures.

- Train-Test Split: Splits data into training and validation set.

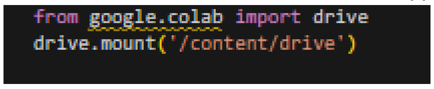


```python
df.rename(columns={'Email Text': 'Email_Text', 'Email Type':'Email_Type'}, inplace=True)
```

Figure21. The code updates column names in the DataFrame `df` to ensure consistency:The code updates column names in the DataFrame `df` to ensure consistency:

- Field name `Email Text` is changed to a new name which is `Email_Text`.

- Thus, the column name `Email Type` is modified to `Email_Type`.



```python
from google.colab import drive
drive.mount('/content/drive')
```

Figure22. This code enables one to access files which are stored in the Google Drive within the Colab environment. It mounts Drive, so the contents of Drive become accessible in this directory:

/content/drive.

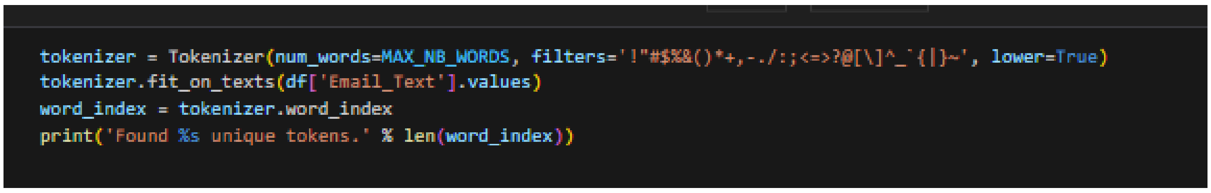

```
tokenizer = Tokenizer(num_words=MAX_NB_WORDS, filters='!"#$%&()*+,-./:;<=>?@[\]^_`{|}~', lower=True)
tokenizer.fit_on_texts(df['Email_Text'].values)
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

Figure23. The following code first fit a keras tokenizer to the `Email_Text` column of the DataFrame, and then use this tokenizer to transform this feature into a sequence of tokens. It sets up the tokenizer defining parameters such as `MAX_NB_WORDS` to reduce the number of words, and characters that should be omitted, plus transforming all the text to lower case. Following the fitting of the tokenizer, it then prints out the total numeric count of unique tokens that has been identified to exist within the `Email_Text` column.

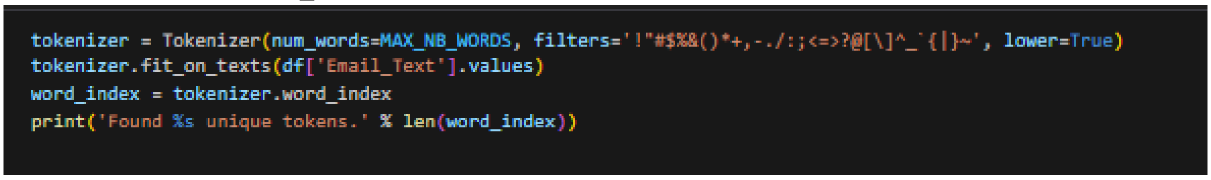

```
tokenizer = Tokenizer(num_words=MAX_NB_WORDS, filters='!"#$%&()*+,-./:;<=>?@[\]^_`{|}~', lower=True)
tokenizer.fit_on_texts(df['Email_Text'].values)
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

Figure24. This code sets up and uses a Keras `Tokenizer` for text data processing:This code sets up and uses a Keras `Tokenizer` for text data processing:

1. Initialize Tokenizer: In the building of a `Tokenizer` object, there are some parameters predetermined such as `MAX_NB_WORDS` where the number of words is to be restricted in use, eliminate characters (e. g. , ‘,’ and ‘. ’) and lower casing of the text.

2. Fit Tokenizer: The tokenizer is fitted on the column of DataFrame ‘df’ denoted as ‘Email_Text’, where the word index is created based on the texts.

3. Print Unique Tokens: It prints the total number of different tokens (words) which is discovered by the help of tokenizer.

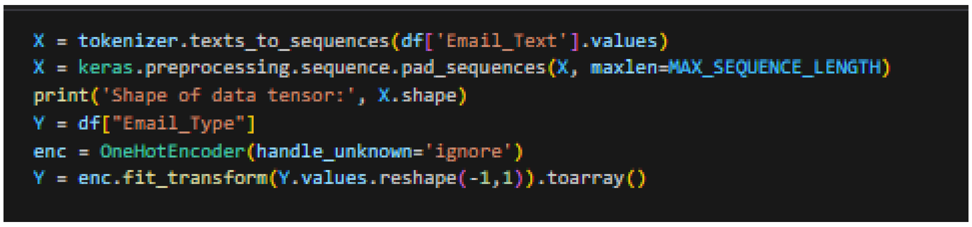

```
X = tokenizer.texts_to_sequences(df['Email_Text'].values)
X = keras.preprocessing.sequence.pad_sequences(X, maxlen=MAX_SEQUENCE_LENGTH)
print('Shape of data tensor:', X.shape)
Y = df["Email_Type"]
enc = OneHotEncoder(handle_unknown='ignore')
Y = enc.fit_transform(Y.values.reshape(-1,1)).toarray()
```

Figure25. This code prepares the data for training a neural network model:This code prepares the data for training a neural network model:

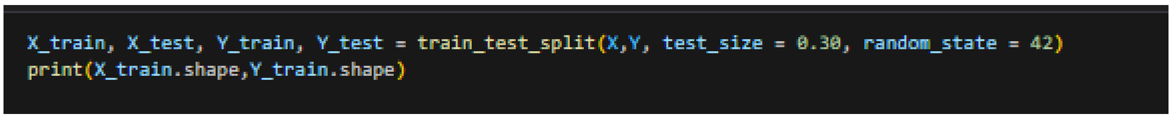
1. Tokenize and Sequence Texts: Converts the text from the `Email_Text` column into sequences of integers using the `tokenizer` whereby every word to a unique integer.

2. Pad Sequences: To make the input of all these integer sequences uniform for the model the current integer sequences are padded to a length of `MAX_SEQUENCE_LENGTH` using zeros.

3. Print Shape: Shows the shape of the resulting data tensor, it is clear from the figure below that the input features are of 7200 rows and 6 columns.

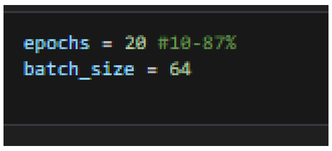
4. Prepare Labels: Proceeds to transform the `Email_Type ` to feature and use `OneHotEncoder ` to convert the labels into a binary matrix form. The process of applying this transformation is to

convert a categorical label into a format that can be understood and learned by a model with much improvements it will transform into a binary matrix.



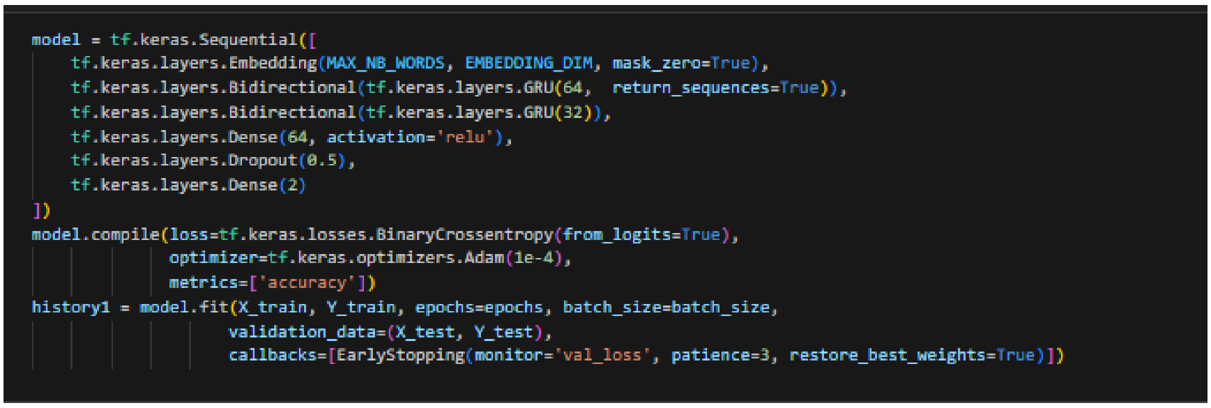
```
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.30, random_state = 42)
print(X_train.shape,Y_train.shape)
```

Figure26. Dataset Splitting: Below is the code that split the entire dataset into the training and testing set, the testing set contain 30% of the overall records. It then follows by printing the shapes of the training feature and label arrays that are produced.



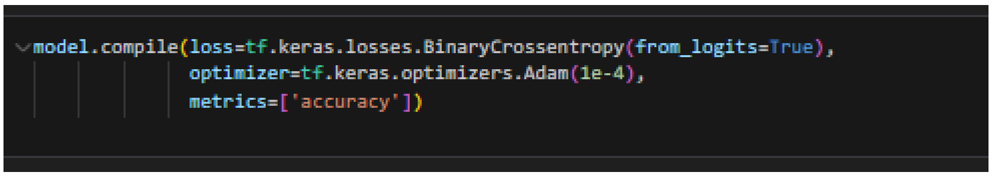
```
epochs = 20 #10-87%
batch_size = 64
```

Figure27. Training Configuration: The following code determines the training process settings for the considered model, with the number of epochs set to 20 and the batch size to 64.



```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(MAX_NB_WORDS, EMBEDDING_DIM, mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.GRU(64,  return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.GRU(32)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(2)
])
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(1e-4),
              metrics=['accuracy'])
history1 = model.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size,
                     validation_data=(X_test, Y_test),
                     callbacks=[EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)])
```

Figure28. Model Definition and Training: First of all, this code compiles and trains a bidirectional GRU neural network for the involved text classify. There are an embedding layer, bidirectional GRU layer, dense layer, dropout layer and a Final dense output layer. Binary cross-entropy is sued as the loss function with Adam optimizer to optimize the model; to curb over training, early stopping is applied such that the training stops after 20 epochs.



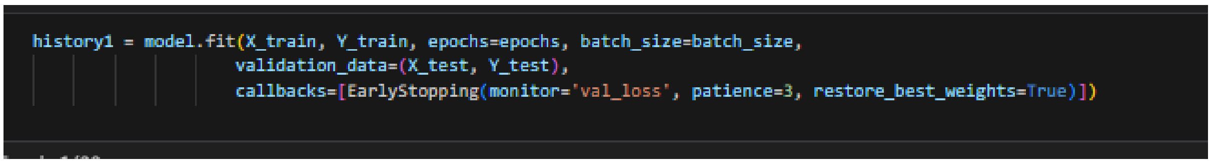
```
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(1e-4),
              metrics=['accuracy'])
```

Figure29. Model Compilation: The following is the Keras model defined by this code:

- Loss Function: `BinaryCrossentropy(from_logits=True)` – Suitable for binary classification problems and deals with the logits directly output from the model.

- Optimizer: `Adam(1e-4)` – The name of an optimization which uses 1. e-4 as its learning rate. 0001 – describes the current learning rates and performs changes in the process of training.

- Metrics: `['accuracy'] – Measures the accuracy of the model when the model is trained abd tested.



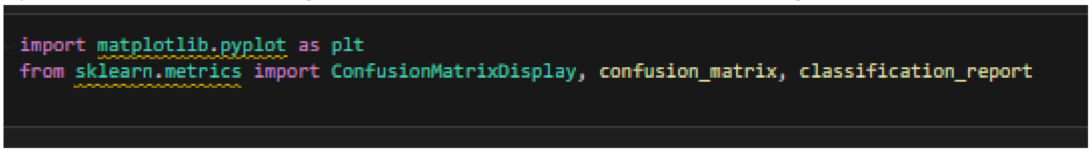
```
history1 = model.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size,
                     validation_data=(X_test, Y_test),
                     callbacks=[EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)])
```

Figure30. Model Training: The following parameters for Keras model training is set in this code:

- Epochs: Number of iterations passes over the entire training data set (as identified earlier).

- Batch Size: Silver count before the model is updated, given in the section above.

- Early Stopping Callback: It can stop the training if the validation loss does not augment for three epochs, and save the weights of the best model to avoid overfitting.



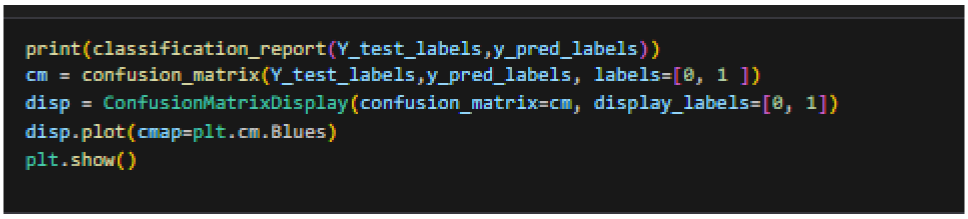
```
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix, classification_report
```

Figure31. Library Imports for Model Evaluation: The code also brings in libraries that will be used to assess the performance of the model:

- `matplotlib. pyplot`: It is used in developing plot and visualization.

- `sklearn. metrics. ConfusionMatrixDisplay` and `confusion_matrix`: Regarding their use: for computing and displaying confusion matrices.

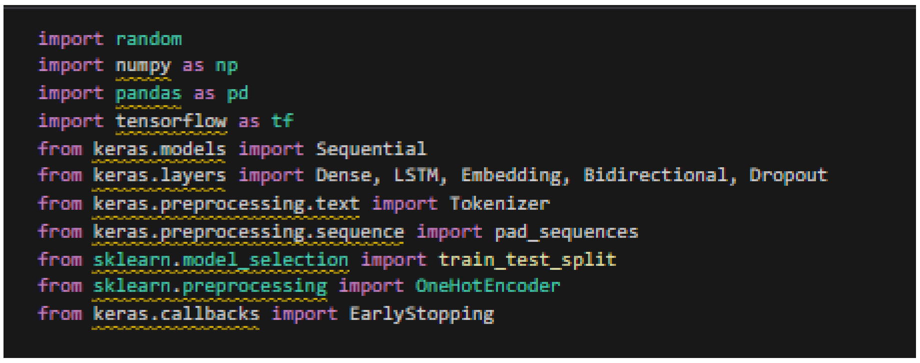
- `classification_report`: Helps to generate a report that will contain the values of precision, recall, and F1-score of the model.



```
print(classification_report(Y_test_labels,y_pred_labels))
cm = confusion_matrix(Y_test_labels,y_pred_labels, labels=[0, 1 ])
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

Figure32. Model Evaluation: On completing the code, the system creates a classification report that provides precision, the recall and F1-score. They also give a heatmap of the confusion matrix to present the actual class against the predicted one in the class distribution.

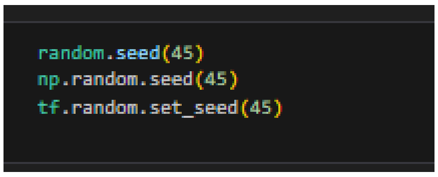
LSTM



```
import random
import numpy as np
import pandas as pd
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, LSTM, Embedding, Bidirectional, Dropout
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from keras.callbacks import EarlyStopping
```

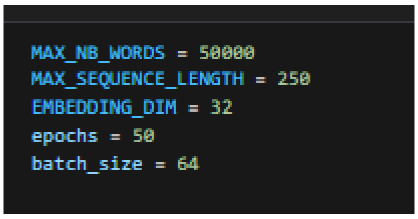
Figure33. Library Imports: The code is used to import the core packages that are essential for deep learning text classification such as, TensorFlow & Keras for the model development and training,

Sklearn for data pre-processing and splitting the data set & several Keras layers, useful utilities for text tokenization & padding of the LSTM based neural network.



```
random.seed(45)
np.random.seed(45)
tf.random.set_seed(45)
```

Figure34. The code initializes random seeds for the Python's built-in random module and libraries, numpy, and Tensorflow to avoid replicability. This has the consequence that, when these seeds are used to initialize the random number generator, every time the same piece of code is executed, the same results are obtained, which is very useful for the debugging phase and for comparisons.



```
MAX_NB_WORDS = 50000
MAX_SEQUENCE_LENGTH = 250
EMBEDDING_DIM = 32
epochs = 50
batch_size = 64
```

Figure35. The code defines key parameters for text processing and model training:The code defines key parameters for text processing and model training:

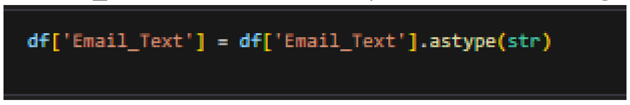
- MAX_NB_WORDS: Restricts the number of different words in the text to fifty thousand.

- MAX_SEQUENCE_LENGTH: Cuts of sequences' length to the 250 tokens and applies padding to the sequences shorter than it.

- EMBEDDING_DIM: Uses 32-dimensional vectors for the word embeddings.

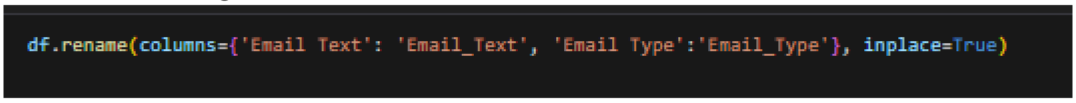
- epochs: That is why to train the model for 50 iterations over the entire dataset is an optimal solution.

- batch_size: Trains on 64 samples at a time during the training session.



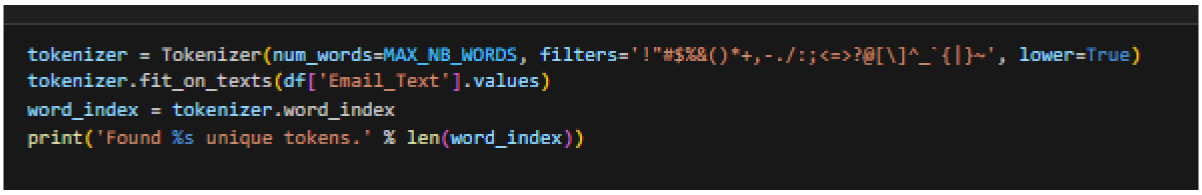
```
df['Email_Text'] = df['Email_Text'].astype(str)
```

Figure36. The code typecasts the 'Email_Text' column of the DataFrame `df` to string data type so that all the entries in this column are treated as string variables. This is very useful in preprocessing of text and training of the models.



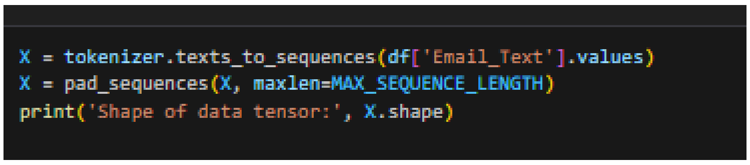
```
df.rename(columns={'Email Text': 'Email_Text', 'Email Type':'Email_Type'}, inplace=True)
```

Figure37. The code renames columns in the DataFrame `df` to avoid naming conflicts: In noun compound, prefix or suffix is added with the separator underscore (_) in between forming a new word respectively 'Email Text' is replaced by 'Email_Text' and 'Email Type' replaced with 'Email_Type'.



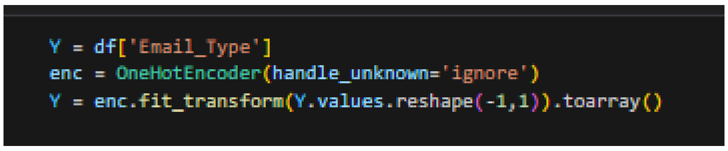
```python
tokenizer = Tokenizer(num_words=MAX_NB_WORDS, filters='!"#$%&()*+,-./:;<=>?@[\]^_`{|}~', lower=True)
tokenizer.fit_on_texts(df['Email_Text'].values)
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

Figure38. The code specifies a `Tokenizer` to work with text data by identifying the maximum number of words to be used `MAX_NB_WORDS` and filtering out symbols. It then applies the tokenizer on the email text data and then prints the number of tokens in the data set.



```python
X = tokenizer.texts_to_sequences(df['Email_Text'].values)
X = pad_sequences(X, maxlen=MAX_SEQUENCE_LENGTH)
print('Shape of data tensor:', X.shape)
```

Figure39. The code converts text data into sequences of integers using a tokenizer, pads these sequences to a fixed length defined by `MAX_SEQUENCE_LENGTH`, and then prints the shape of the resulting data tensor.

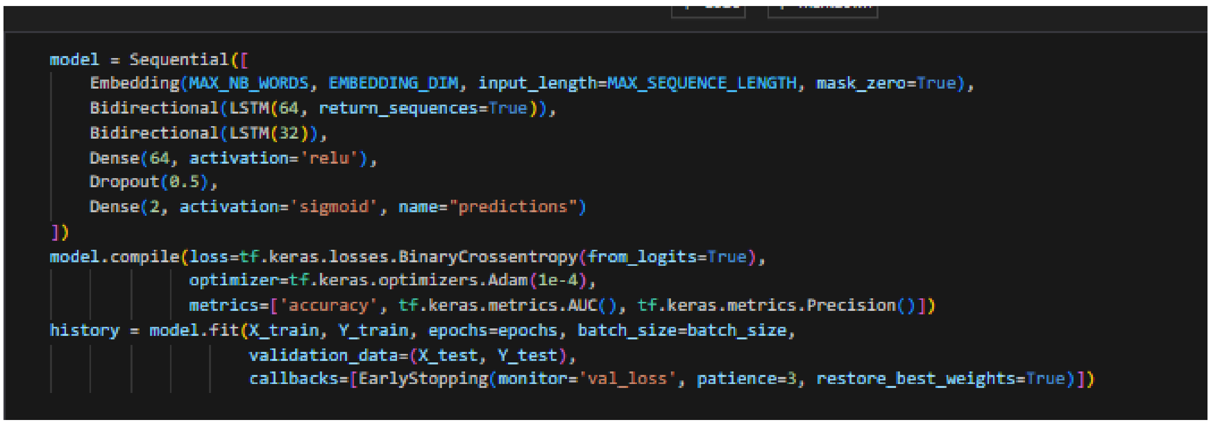


```python
Y = df['Email_Type']
enc = OneHotEncoder(handle_unknown='ignore')
Y = enc.fit_transform(Y.values.reshape(-1,1)).toarray()
```

Figure40. The code performs these actions:The code performs these actions:

1. Extract Target Variable: It sets `Y` equal to the Dframe's column called 'Email_Type', which has the email labels.

2. One-Hot Encoding: For the target variable, which is the species of iris, it further encodes the categorical nature of `Y` as a binary vector using `OneHotEncoder`. This change is used for the classification tasks which involved reformatting the labels and encoding them in numbers.



```python
model = Sequential([
    Embedding(MAX_NB_WORDS, EMBEDDING_DIM, input_length=MAX_SEQUENCE_LENGTH, mask_zero=True),
    Bidirectional(LSTM(64, return_sequences=True)),
    Bidirectional(LSTM(32)),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(2, activation='sigmoid', name="predictions")
])
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(1e-4),
              metrics=['accuracy', tf.keras.metrics.AUC(), tf.keras.metrics.Precision()])
history = model.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size,
                    validation_data=(X_test, Y_test),
                    callbacks=[EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)])
```

Figure41. The code sets up and trains a text classification neural network using a Bidirectional LSTM model with the following components:The code sets up and trains a text classification neural network using a Bidirectional LSTM model with the following components:

- Embedding Layer: Translates the words into high-dimensional vectors, that puts the text in the numerical form for its analyzing.

- Bidirectional LSTM Layers: Computes sequence forward and backward making this model also aware of contexts after and before any location in a sequence.

- Dense Layer: A Dense layer connected layer that takes features generated by the LSTM layers and provides the prediction.

- Dropout Layer: Guards against over fitting by eliminating a randomly selected unit at training with an aim of making the model to learn to perform on any given set of units.

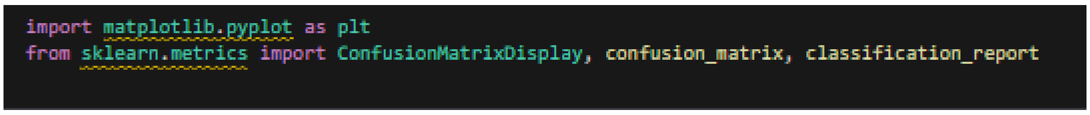
- Output Layer: Gives out probabilities of the results in a binary classification.

The model is compiled with:The model is compiled with:

- Loss Function: Binary Crossentropy, for binary classification problem.

- Optimizer: Adam which changes the learning rate in the process of training.

Training is carried out to a maximum of 50 iterations using dropout regulation based on the validation set loss. Accuracy rate AUC and precision are the common model performance measuring standards.



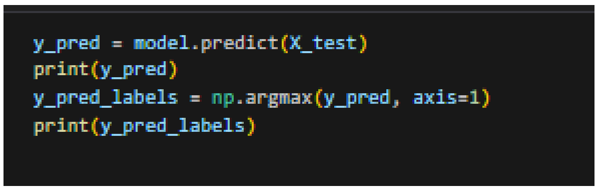
```python
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix, classification_report
```

Figure42. The code imports libraries for visualizing the performance of machine learning models:The code imports libraries for visualizing the performance of machine learning models:

- `matplotlib. pyplot`: It is used for the development of a range of plots and graphics such as line plots, scatter plots, distributes and many more.

- `sklearn. metrics. ConfusionMatrixDisplay` and `confusion_matrix`: For measuring and showing the confusion matrices.

- `classification_report`: Outputs an elaborate result report which contains precision, recall, and a F1-score to compare the efficiency of the model..

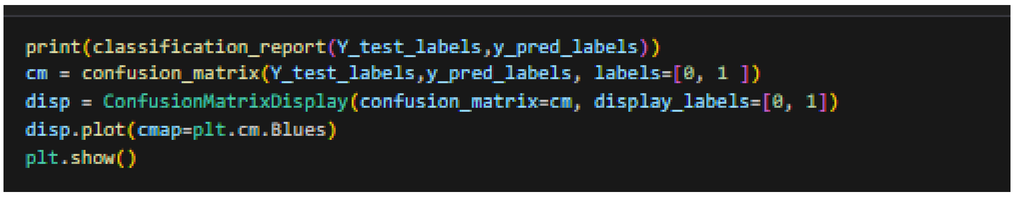


```python
y_pred = model.predict(X_test)
print(y_pred)
y_pred_labels = np.argmax(y_pred, axis=1)
print(y_pred_labels)
```

Figure43. The code evaluates the model on the test set by:The code evaluates the model on the test set by:

1. Estimating Class Probabilities: Applying the model to predict probabilities for each class on the test data.

2. Converting Probabilities to Class Labels: Deciding on the predicted class by choosing the index of the maximum probability for each instance, thus, converting the probabilistic results obtained into distinct classes..



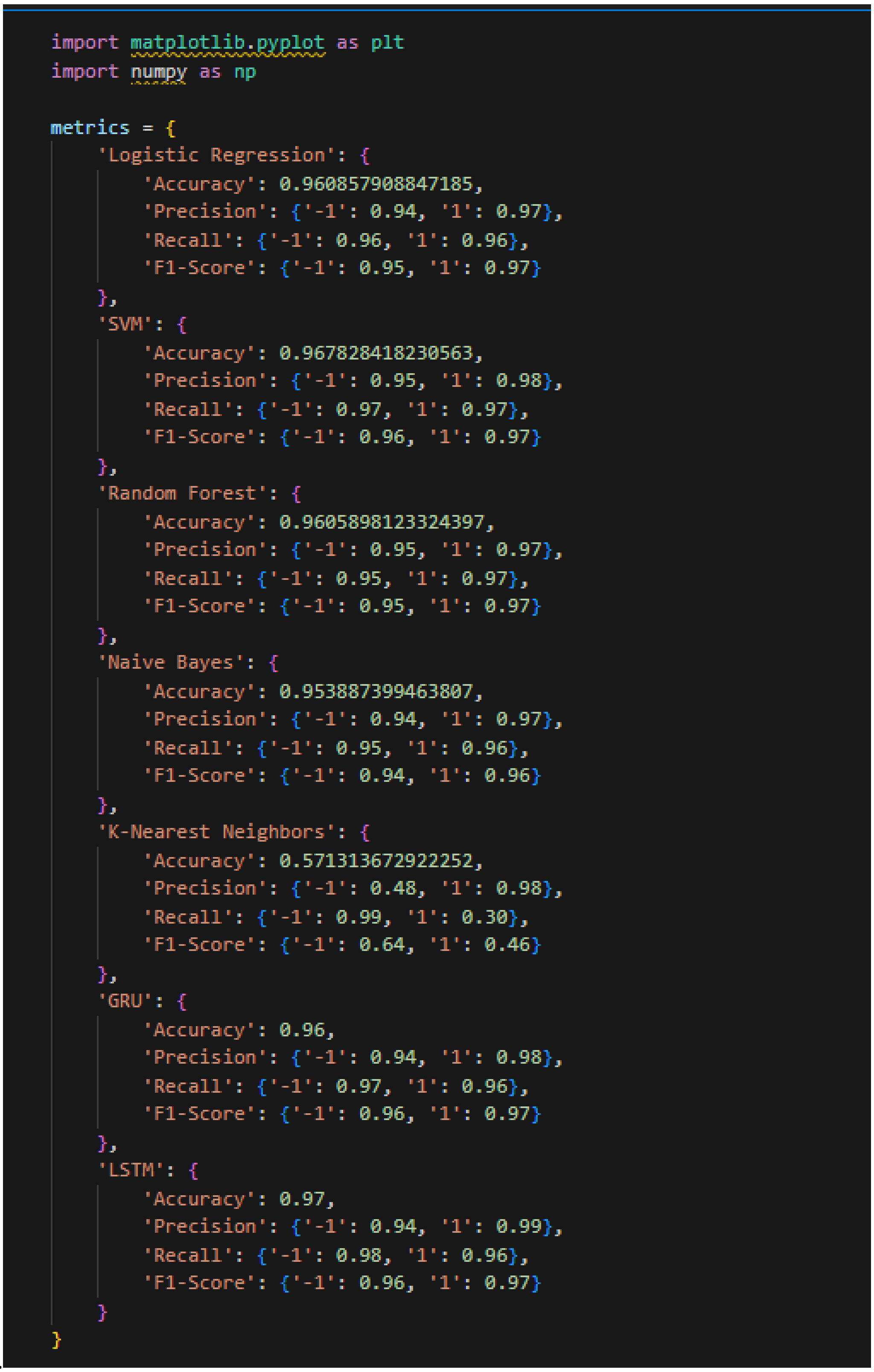
```
print(classification_report(Y_test_labels,y_pred_labels))
cm = confusion_matrix(Y_test_labels,y_pred_labels, labels=[0, 1 ])
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

Figure44. The code evaluates the model's performance by:The code evaluates the model's performance by:

1. Printing the Classification Report: It includes precision, recall, and F1-score measures for each class to describe the model's performance in detail.

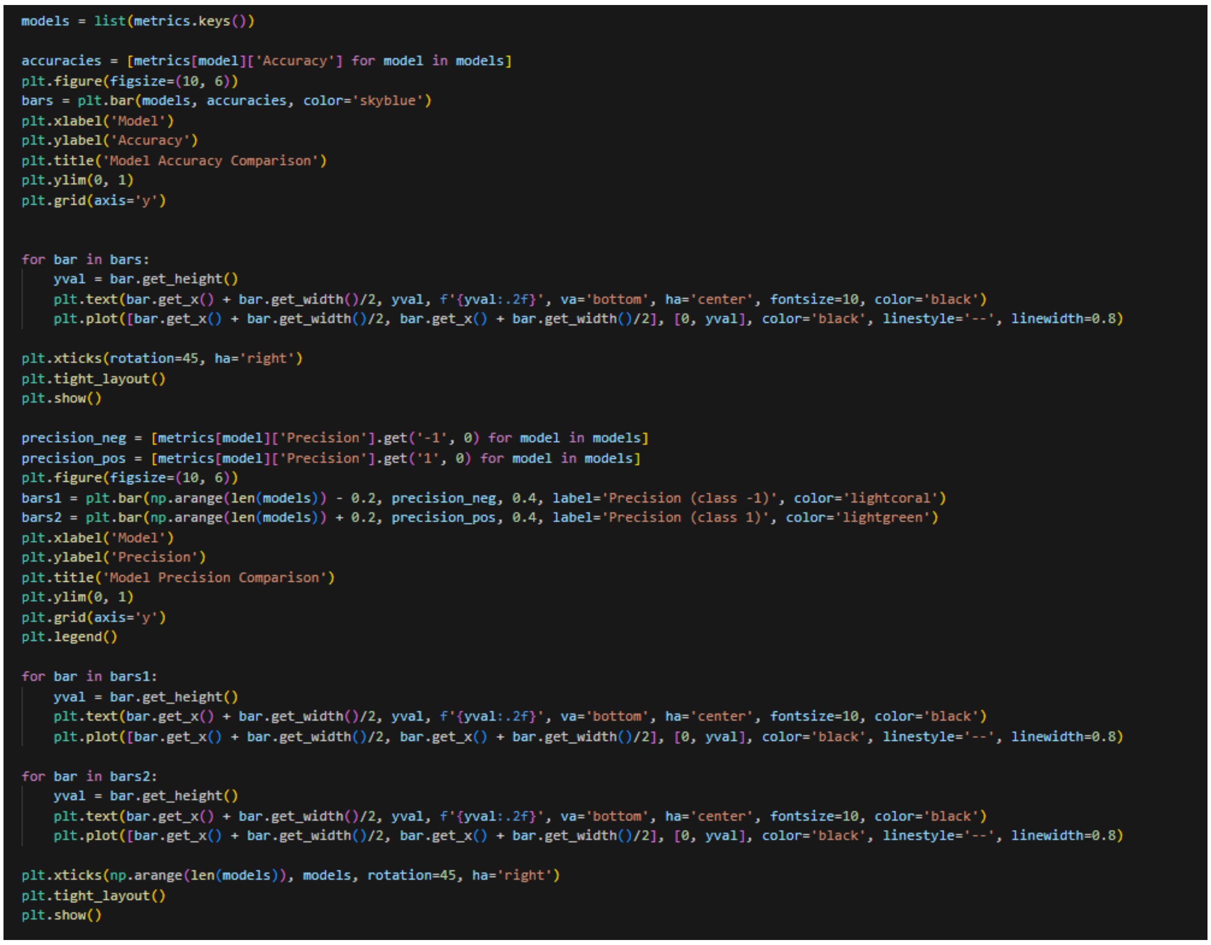
2. Computing the Confusion Matrix: Computes for the matrix consisting of true positive values, true negatives, false positives and false negatives.

3. Visualizing the Confusion Matrix: For the visualization of the confusion matrix, it plots the heatmap to enhance the performance interpretation of the model.

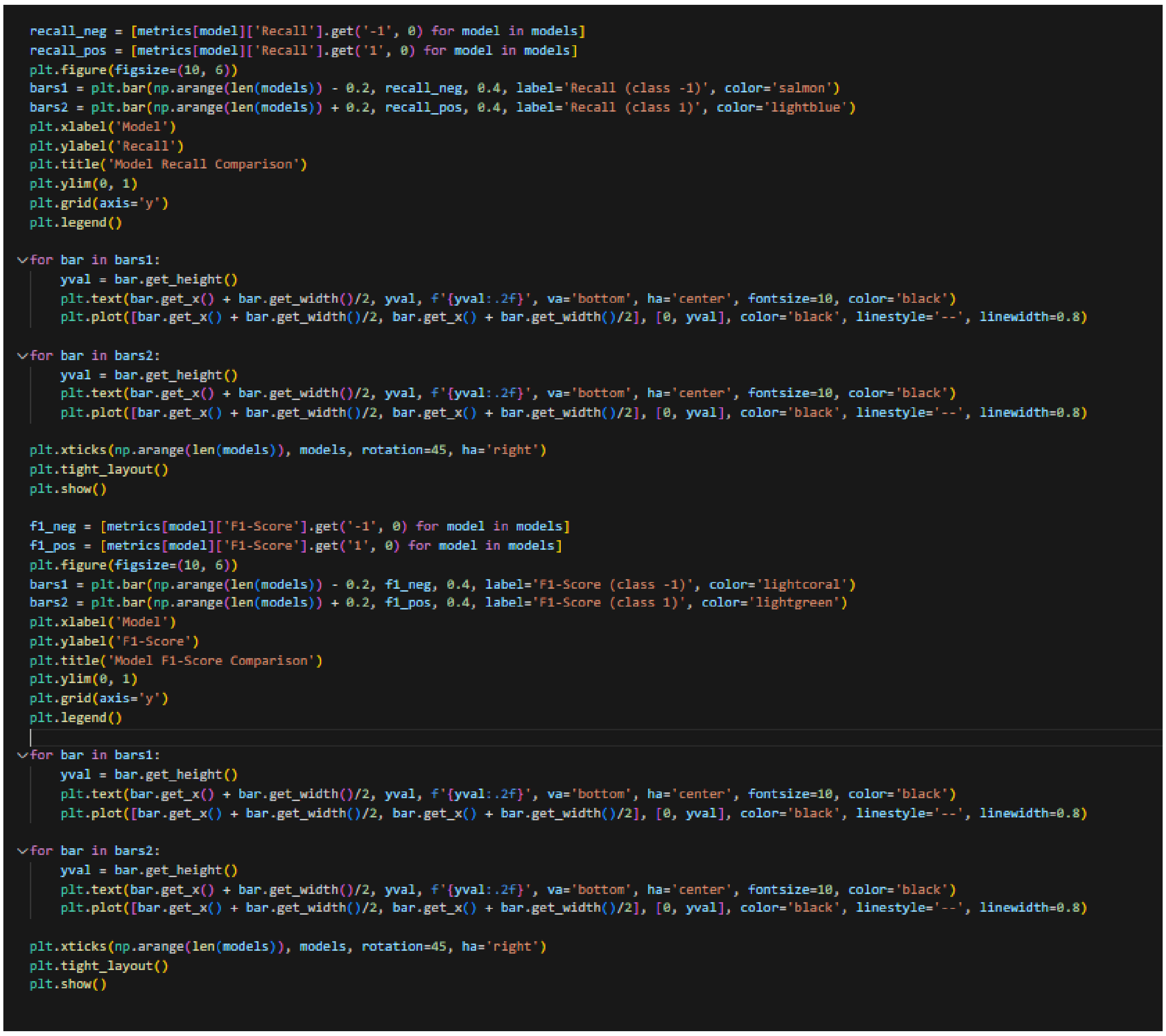


```python
import matplotlib.pyplot as plt
import numpy as np

metrics = {
    'Logistic Regression': {
        'Accuracy': 0.960857908847185,
        'Precision': {'-1': 0.94, '1': 0.97},
        'Recall': {'-1': 0.96, '1': 0.96},
        'F1-Score': {'-1': 0.95, '1': 0.97}
    },
    'SVM': {
        'Accuracy': 0.967828418230563,
        'Precision': {'-1': 0.95, '1': 0.98},
        'Recall': {'-1': 0.97, '1': 0.97},
        'F1-Score': {'-1': 0.96, '1': 0.97}
    },
    'Random Forest': {
        'Accuracy': 0.9605898123324397,
        'Precision': {'-1': 0.95, '1': 0.97},
        'Recall': {'-1': 0.95, '1': 0.97},
        'F1-Score': {'-1': 0.95, '1': 0.97}
    },
    'Naive Bayes': {
        'Accuracy': 0.953887399463807,
        'Precision': {'-1': 0.94, '1': 0.97},
        'Recall': {'-1': 0.95, '1': 0.96},
        'F1-Score': {'-1': 0.94, '1': 0.96}
    },
    'K-Nearest Neighbors': {
        'Accuracy': 0.571313672922252,
        'Precision': {'-1': 0.48, '1': 0.98},
        'Recall': {'-1': 0.99, '1': 0.30},
        'F1-Score': {'-1': 0.64, '1': 0.46}
    },
    'GRU': {
        'Accuracy': 0.96,
        'Precision': {'-1': 0.94, '1': 0.98},
        'Recall': {'-1': 0.97, '1': 0.96},
        'F1-Score': {'-1': 0.96, '1': 0.97}
    },
    'LSTM': {
        'Accuracy': 0.97,
        'Precision': {'-1': 0.94, '1': 0.99},
        'Recall': {'-1': 0.98, '1': 0.96},
        'F1-Score': {'-1': 0.96, '1': 0.97}
    }
}
```

3..



```python
models = list(metrics.keys())

accuracies = [metrics[model]['Accuracy'] for model in models]
plt.figure(figsize=(10, 6))
bars = plt.bar(models, accuracies, color='skyblue')
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Model Accuracy Comparison')
plt.ylim(0, 1)
plt.grid(axis='y')


for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    plt.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

precision_neg = [metrics[model]['Precision'].get('-1', 0) for model in models]
precision_pos = [metrics[model]['Precision'].get('1', 0) for model in models]
plt.figure(figsize=(10, 6))
bars1 = plt.bar(np.arange(len(models)) - 0.2, precision_neg, 0.4, label='Precision (class -1)', color='lightcoral')
bars2 = plt.bar(np.arange(len(models)) + 0.2, precision_pos, 0.4, label='Precision (class 1)', color='lightgreen')
plt.xlabel('Model')
plt.ylabel('Precision')
plt.title('Model Precision Comparison')
plt.ylim(0, 1)
plt.grid(axis='y')
plt.legend()

for bar in bars1:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    plt.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

for bar in bars2:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    plt.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

plt.xticks(np.arange(len(models)), models, rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



```python
recall_neg = [metrics[model]['Recall'].get('-1', 0) for model in models]
recall_pos = [metrics[model]['Recall'].get('1', 0) for model in models]
plt.figure(figsize=(10, 6))
bars1 = plt.bar(np.arange(len(models)) - 0.2, recall_neg, 0.4, label='Recall (class -1)', color='salmon')
bars2 = plt.bar(np.arange(len(models)) + 0.2, recall_pos, 0.4, label='Recall (class 1)', color='lightblue')
plt.xlabel('Model')
plt.ylabel('Recall')
plt.title('Model Recall Comparison')
plt.ylim(0, 1)
plt.grid(axis='y')
plt.legend()

for bar in bars1:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    plt.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

for bar in bars2:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    plt.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

plt.xticks(np.arange(len(models)), models, rotation=45, ha='right')
plt.tight_layout()
plt.show()

f1_neg = [metrics[model]['F1-Score'].get('-1', 0) for model in models]
f1_pos = [metrics[model]['F1-Score'].get('1', 0) for model in models]
plt.figure(figsize=(10, 6))
bars1 = plt.bar(np.arange(len(models)) - 0.2, f1_neg, 0.4, label='F1-Score (class -1)', color='lightcoral')
bars2 = plt.bar(np.arange(len(models)) + 0.2, f1_pos, 0.4, label='F1-Score (class 1)', color='lightgreen')
plt.xlabel('Model')
plt.ylabel('F1-Score')
plt.title('Model F1-Score Comparison')
plt.ylim(0, 1)
plt.grid(axis='y')
plt.legend()

for bar in bars1:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    plt.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

for bar in bars2:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}', va='bottom', ha='center', fontsize=10, color='black')
    plt.plot([bar.get_x() + bar.get_width()/2, bar.get_x() + bar.get_width()/2], [0, yval], color='black', linestyle='--', linewidth=0.8)

plt.xticks(np.arange(len(models)), models, rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

Figure45. Here's a summary of what each section of the code does for plotting model performance metrics:Here's a summary of what each section of the code does for plotting model performance metrics:

1. Accuracy Comparison:

- Data Preparation: Pulling out the accuracy values from the `metrics` dictionary for each model.

- Plotting: Produces a bar chart of the models' performances; the y-axis represents the accuracy of the model and the labels add clarity.

2. Precision Comparison:

- Data Preparation: Pulls the precision values for both classes (-1 and 1) from the `metrics` dictionary.

- Plotting: Builds the grouped bar chart to compare the precision of each model for both classes with annotations on the values.

3. Recall Comparison:

- Data Preparation: Pulls out the recall values for classes -1 and 1 from the `metrics` dictionary.

- Plotting: Decision: presents a grouped bar chart with the recall scores for each model and class, labeling the axes and the chart with recall values.

4. F1-Score Comparison:

- Data Preparation: Gets the F1-Score averages of every class (-1 and 1) from the 'metrics' dictionary.

- Plotting: Generates a grouped bar to give F1-Scores of each model on different classes with additional commentaries for F1-Score.

All presented charts use grouped bars for different classes, have some value annotations, and have a legend so it would be easier to compare models by various metrics.

3 Steps to Run and execute the codes

Step 1: Initially the user has to go to the google drive which contain the codes and data.

Step 2: Run the colab file.

Step 3: Access authentication for the drive must be made.

Step 4: Now record the application by start running it in the Anaconda Prompt.

Step 5: Run the code python -m streamlit run app. py