Megamind assignment by Abhinav Tamta

Output video link:

https://drive.google.com/file/d/1BNpPf6v1khUGJjHq1K5dAAdyHcpA4FZt/view?usp=sharing

Explaining the Dataset

1. **ID**: This is likely an identifier or a unique key for each record in the dataset. It helps in distinguishing one entry from another.
2. **Sender_IP**: This represents the IP address of the entity sending the network traffic. In the context of identifying Botnet attacks, it could be the IP address of a potentially malicious source.
3. **Sender_Port**: This is the port number on the sender's side through which the communication is taking place. Ports help identify specific processes or services on a network.
4. **Target_IP**: This is the IP address of the entity or system that is the target of the network communication. In the context of Botnet attacks, this could be a victim machine.
5. **Target_Port**: Similar to Sender_Port, this is the port number on the target's side through which the communication is happening.
6. **Transport_Protocol**: This indicates the protocol used for communication, such as TCP (Transmission Control Protocol) or UDP (User Datagram Protocol).
7. **Duration**: This may represent the duration of the communication or connection between the sender and target, measuring how long the communication persists.
8. **AvgDuration**: This could be the average duration of multiple connections or communications.
9. **PBS (Payload Byte Size)**: It refers to the size of the payload in bytes. In the context of network traffic, the payload is the actual data being transmitted (excluding headers).
10. **AvgPBS**: This might be the average payload byte size over multiple connections.
11. **TBS (Total Byte Size)**: This represents the total size of the transmitted data, including both the payload and any protocol headers.
12. **PBR (Payload Byte Rate)**: It could be the rate at which payload bytes are transmitted per unit of time.
13. **AvgPBR**: This may represent the average payload byte rate over multiple connections.
14. **TBR (Total Byte Rate)**: Similar to PBR, but it includes all bytes transmitted, including both payload and headers.

15. **Missed_Bytes**: This could refer to the number of bytes that were not successfully transmitted or received.
16. **Packets_Sent**: The total number of packets sent during the communication.
17. **Packets_Received**: The total number of packets received during the communication.
18. **SRPR (Sender-to-Receiver Packet Ratio)**: This might be the ratio of packets sent by the sender to those received by the receiver.
19. **class**: This is likely the label or class assigned to each record, indicating whether the communication is identified as a Botnet attack or not. It could be binary (e.g., 0 for normal, 1 for Botnet) or may have multiple classes.

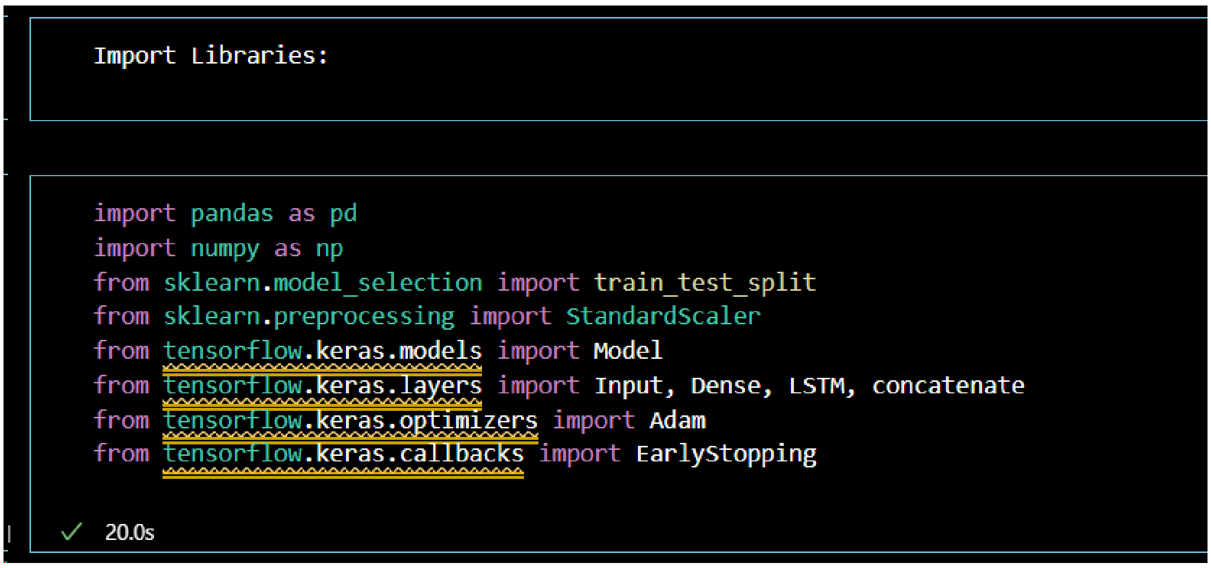
Solution notebook for Botnet Detection

The provided code demonstrates how to use a trained hybrid deep learning model (combining DNN and LSTM layers) to make predictions on new data. Here's a summary of the code:

1. **New Data Preparation:**
    - A new dataset (**new_data**) is created with the same features as the original dataset used for training the model.
    - The new data is formatted as a pandas DataFrame.
2. **Scaling New Data:**
    - The same scaler used during the training phase is applied to scale the new data. This ensures consistency in data preprocessing.
3. **Reshaping for LSTM:**
    - The new data is reshaped to match the input shape expected by the LSTM layer in the model.
4. **Making Predictions:**
    - The trained hybrid model (**hybrid_model**) is used to predict the likelihood of a botnet attack based on the new data.
    - The model predicts a probability, and the result is rounded to obtain binary predictions (0 for non-botnet and 1 for botnet).
5. **Displaying Results:**
    - The probability of the input being a botnet attack is printed.
    - The binary prediction (0 or 1) is also displayed.

This code is designed for a specific use case where the model has been trained on a dataset with features related to network traffic characteristics, and it is applied to make predictions on a new instance of such data. It is crucial to ensure that the new data is preprocessed in a manner consistent with the training data to obtain meaningful predictions from the model.

Importing the libraries



Import Libraries:

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, LSTM, concatenate
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
```

✓ 20.0s

Certainly! Let's break down the code block and understand each import statement:

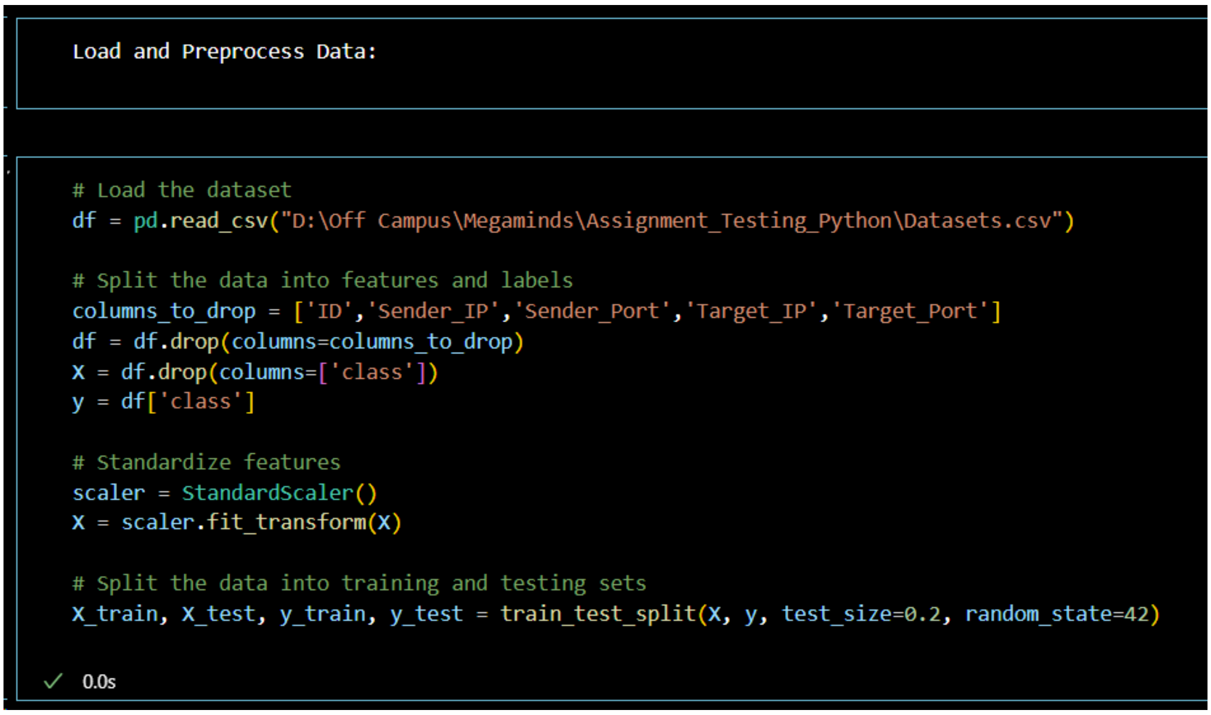
1. **import pandas as pd**:
    - This statement imports the pandas library and aliases it as **pd**. Pandas is a powerful data manipulation and analysis library for Python. It provides data structures like DataFrame for handling structured data.
2. **import numpy as np**:
    - This statement imports the NumPy library and aliases it as **np**. NumPy is a numerical computing library in Python that provides support for large, multi-dimensional arrays and matrices, along with mathematical functions to operate on these.
3. **from sklearn.model_selection import train_test_split**:
    - This line imports the **train_test_split** function from scikit-learn's **model_selection** module. **train_test_split** is commonly used to split a dataset into training and testing sets for machine learning.
4. **from sklearn.preprocessing import StandardScaler**:

- This line imports the **StandardScaler** class from scikit-learn's **preprocessing** module. **StandardScaler** is often used to standardize features by removing the mean and scaling to unit variance.

5. **from tensorflow.keras.models import Model**:
    - This statement imports the **Model** class from the **models** module in the TensorFlow Keras API. **Model** is the base class for all Keras models.
6. **from tensorflow.keras.layers import Input, Dense, LSTM, concatenate**:
    - This line imports specific layer classes from the Keras layers module. These layers include:
        - **Input**: Represents the input layer.
        - **Dense**: Represents a fully connected layer (dense layer) in a neural network.
        - **LSTM**: Represents a Long Short-Term Memory layer, commonly used for sequence data.
        - **concatenate**: Represents the layer for concatenating the output of multiple layers.
7. **from tensorflow.keras.optimizers import Adam**:
    - This statement imports the **Adam** optimizer from the Keras optimizers module. Adam is a popular optimization algorithm used for training neural networks.
8. **from tensorflow.keras.callbacks import EarlyStopping**:
    - This line imports the **EarlyStopping** callback from Keras callbacks. **EarlyStopping** is used to stop training the model when a monitored metric has stopped improving, preventing overfitting.

In summary, this code block imports essential libraries and modules for data manipulation (Pandas, NumPy), machine learning tasks (scikit-learn), and building neural networks (TensorFlow Keras). These libraries and modules will be used in subsequent code to preprocess data, split datasets, define model architecture, and train the model.

Load and Preprocess Data:



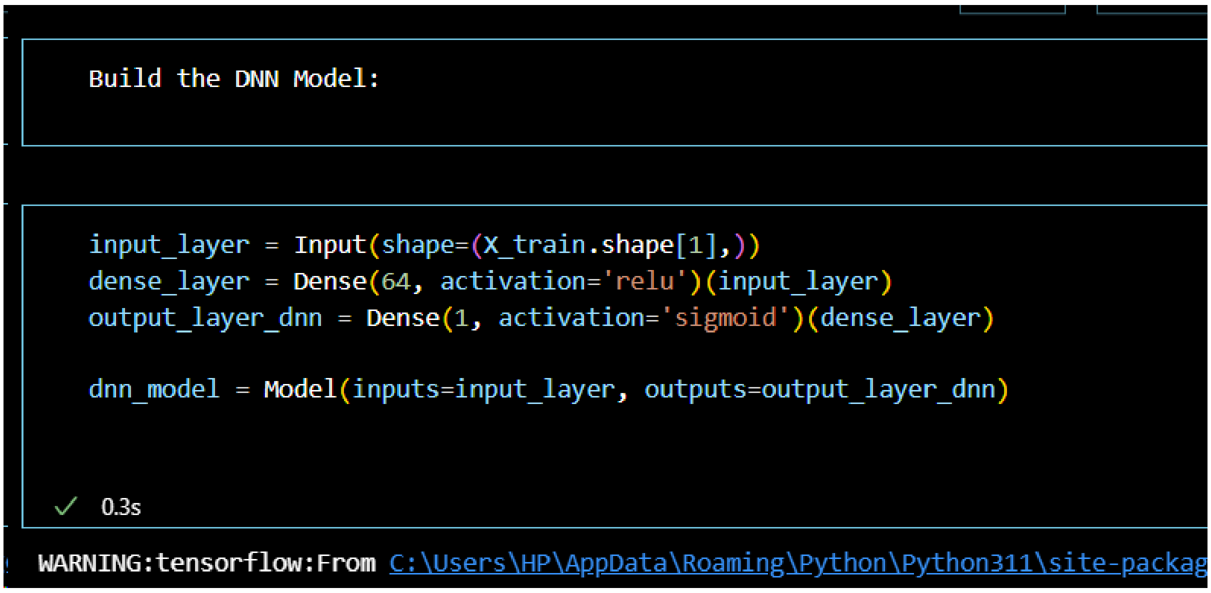
```
Load and Preprocess Data:
```

```
# Load the dataset
df = pd.read_csv("D:\Off Campus\Megaminds\Assignment_Testing_Python\Datasets.csv")

# Split the data into features and labels
columns_to_drop = ['ID','Sender_IP','Sender_Port','Target_IP','Target_Port']
df = df.drop(columns=columns_to_drop)
X = df.drop(columns=['class'])
y = df['class']

# Standardize features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

✓ 0.0s

This code block first loads a dataset from a CSV file using pandas, where the dataset includes features and a target variable labeled as 'class.' The features are then separated into 'X,' excluding the target variable, and the target variable itself is assigned to 'y.' Subsequently, a StandardScaler is employed to standardize the features, a crucial step in many machine learning models. Finally, the dataset is split into training and testing sets using the train_test_split function from scikit-learn, with 80% of the data designated for training and 20% for testing. The random_state parameter is set to 42 for reproducibility. The resulting standardized feature sets (X_train and X_test) and corresponding target sets (y_train and y_test) are prepared for training and evaluating machine learning models.

Build the DNN model



```
Build the DNN Model:
```

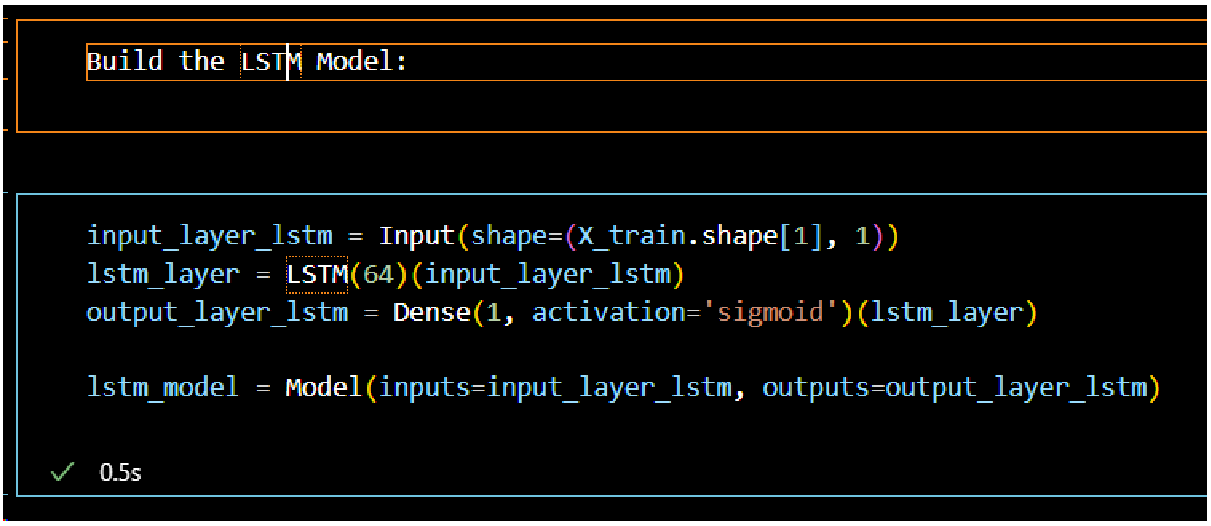
```
input_layer = Input(shape=(X_train.shape[1],))
dense_layer = Dense(64, activation='relu')(input_layer)
output_layer_dnn = Dense(1, activation='sigmoid')(dense_layer)

dnn_model = Model(inputs=input_layer, outputs=output_layer_dnn)
```

✓ 0.3s

```
WARNING:tensorflow:From C:\Users\HP\AppData\Roaming\Python\Python311\site-packag
```

In this code block, a neural network model is defined using the Keras API. An input layer is created with a shape corresponding to the number of features in the training set (**X_train.shape[1]**). This input layer is then connected to a dense layer with 64 units and a rectified linear unit (ReLU) activation function. The dense layer captures complex patterns in the data. The output layer is another dense layer with a single unit and a sigmoid activation function, which is suitable for binary classification tasks. The resulting model, referred to as **dnn_model**, is established by specifying the input and output layers. This sequential architecture forms the basis of a deep neural network (DNN) designed to learn and represent patterns in the input data for the given classification task.

Build the LSTM Model



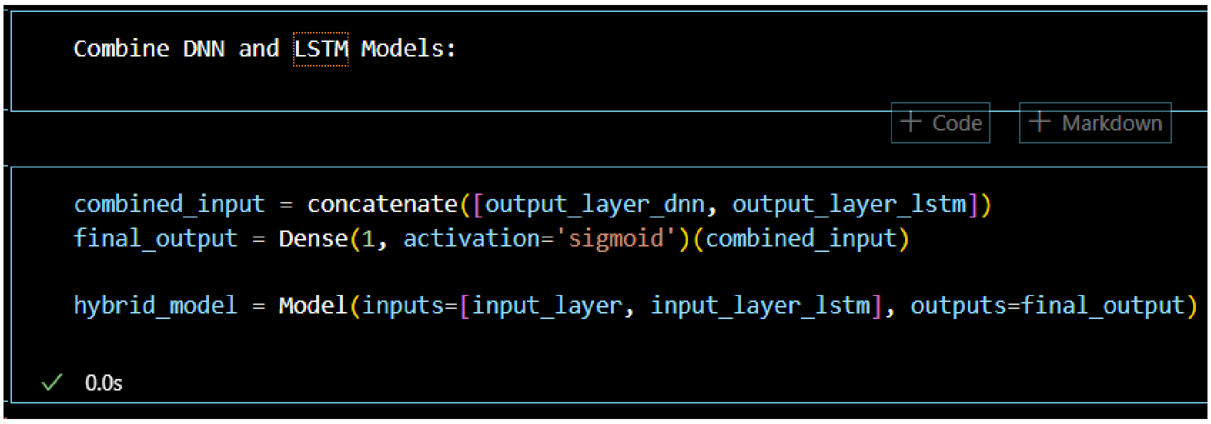
Build the LSTM Model:

```
input_layer_lstm = Input(shape=(X_train.shape[1], 1))
lstm_layer = LSTM(64)(input_layer_lstm)
output_layer_lstm = Dense(1, activation='sigmoid')(lstm_layer)

lstm_model = Model(inputs=input_layer_lstm, outputs=output_layer_lstm)
```

✓ 0.5s

In this code block, a Long Short-Term Memory (LSTM) model is defined using the Keras API. An input layer specifically tailored for LSTM is created with a shape of **(X_train.shape[1], 1)**, indicating that the model expects input sequences with a length equal to the number of features in the training set (**X_train.shape[1]**). The input layer is connected to an LSTM layer with 64 memory units, allowing the model to capture and remember patterns in sequential data. The output of the LSTM layer is then connected to a dense layer with a single unit and a sigmoid activation function, which is typical for binary classification problems. The resulting model, referred to as **lstm_model**, is established by specifying the input and output layers. This architecture is suitable for learning temporal dependencies and patterns in sequential data, making it particularly relevant for tasks involving time-series or sequence-based information.

Combine DNN and LSTM Models



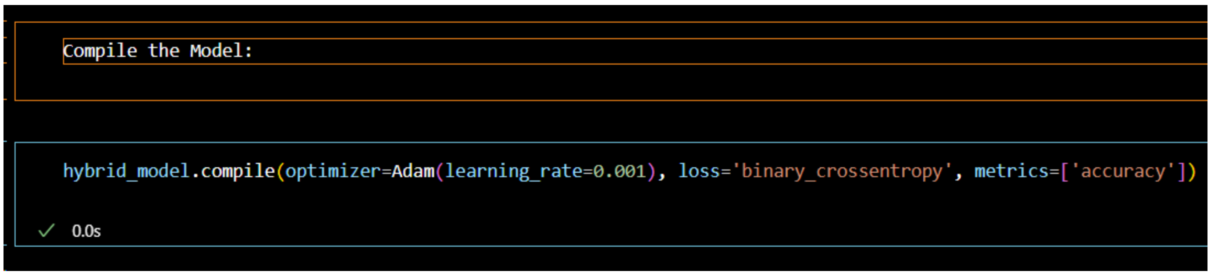
Combine DNN and LSTM Models:

```python
combined_input = concatenate([output_layer_dnn, output_layer_lstm])
final_output = Dense(1, activation='sigmoid')(combined_input)

hybrid_model = Model(inputs=[input_layer, input_layer_lstm], outputs=final_output)
```

```
✓ 0.0s
```

In this code block, a hybrid model is constructed by combining the outputs of two previously defined models: a DNN model (**dnn_model**) and an LSTM model (**lstm_model**). The outputs of these models, **output_layer_dnn** and **output_layer_lstm**, are concatenated along a new dimension. This concatenated output is then fed into a final dense layer with a single unit and a sigmoid activation function. The resulting model, referred to as **hybrid_model**, is established by specifying both the input layers (**input_layer** and **input_layer_lstm**) and the final output layer. This hybrid architecture integrates the capabilities of both a deep neural network and a long short-term memory network, potentially enabling the model to capture complex patterns in both tabular and sequential data

Compile the Model



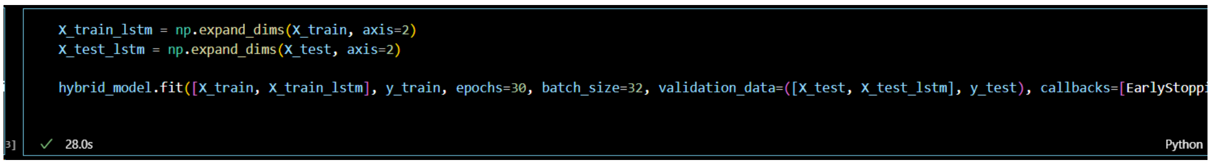
Compile the Model:

```python
hybrid_model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])
```

```
✓ 0.0s
```

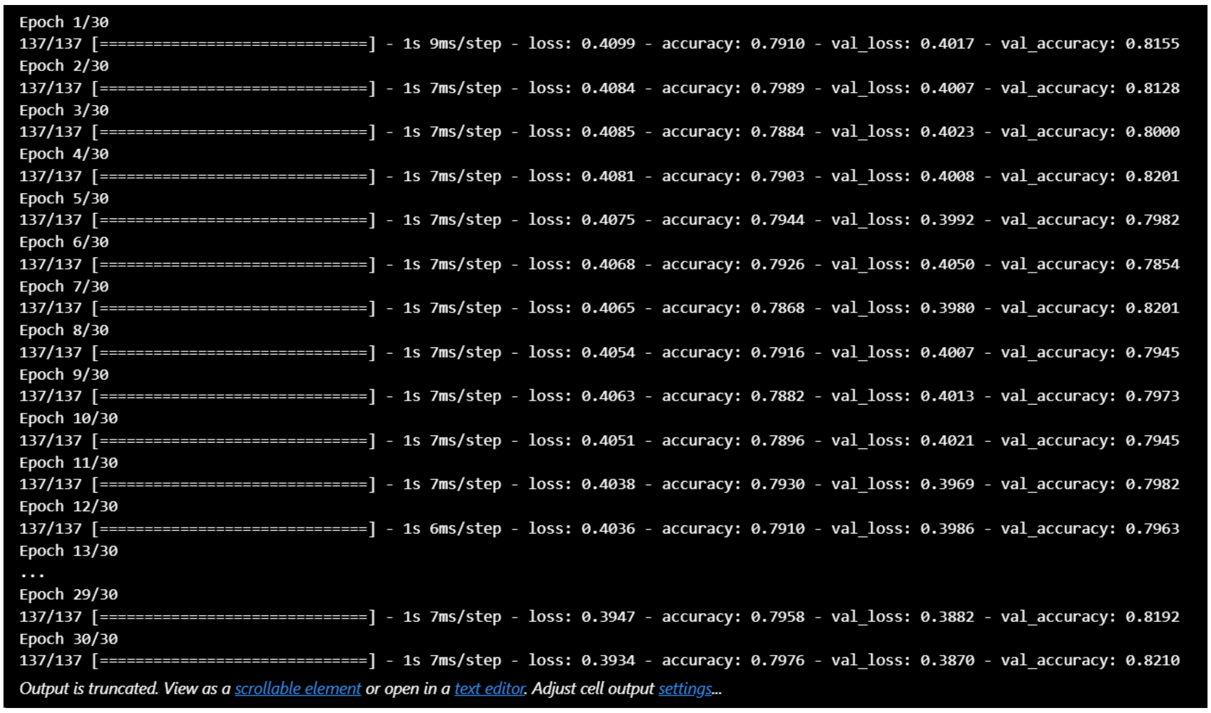
n this code block, the hybrid model (**hybrid_model**) is compiled, specifying its optimization strategy, loss function, and evaluation metrics. The **Adam** optimizer is chosen with a learning rate of 0.001, which is a popular optimization algorithm for training neural networks. The chosen loss function is **'binary_crossentropy'**, which is appropriate for binary classification tasks. Additionally, the accuracy is selected as the evaluation metric, providing a measure of the model's performance during training. The compilation step prepares the hybrid model for training by configuring its learning process with the specified optimization algorithm, loss function, and evaluation metric.

Train the Model



```python
X_train_lstm = np.expand_dims(X_train, axis=2)
X_test_lstm = np.expand_dims(X_test, axis=2)

hybrid_model.fit([X_train, X_train_lstm], y_train, epochs=30, batch_size=32, validation_data=([X_test, X_test_lstm], y_test), callbacks=[EarlyStoppi
```

```
3]  ✓ 28.0s                                                    Python
```


```
Epoch 1/30
137/137 [==============================] - 1s 9ms/step - loss: 0.4099 - accuracy: 0.7910 - val_loss: 0.4017 - val_accuracy: 0.8155
Epoch 2/30
137/137 [==============================] - 1s 7ms/step - loss: 0.4084 - accuracy: 0.7989 - val_loss: 0.4007 - val_accuracy: 0.8128
Epoch 3/30
137/137 [==============================] - 1s 7ms/step - loss: 0.4085 - accuracy: 0.7884 - val_loss: 0.4023 - val_accuracy: 0.8000
Epoch 4/30
137/137 [==============================] - 1s 7ms/step - loss: 0.4081 - accuracy: 0.7903 - val_loss: 0.4008 - val_accuracy: 0.8201
Epoch 5/30
137/137 [==============================] - 1s 7ms/step - loss: 0.4075 - accuracy: 0.7944 - val_loss: 0.3992 - val_accuracy: 0.7982
Epoch 6/30
137/137 [==============================] - 1s 7ms/step - loss: 0.4068 - accuracy: 0.7926 - val_loss: 0.4050 - val_accuracy: 0.7854
Epoch 7/30
137/137 [==============================] - 1s 7ms/step - loss: 0.4065 - accuracy: 0.7868 - val_loss: 0.3980 - val_accuracy: 0.8201
Epoch 8/30
137/137 [==============================] - 1s 7ms/step - loss: 0.4054 - accuracy: 0.7916 - val_loss: 0.4007 - val_accuracy: 0.7945
Epoch 9/30
137/137 [==============================] - 1s 7ms/step - loss: 0.4063 - accuracy: 0.7882 - val_loss: 0.4013 - val_accuracy: 0.7973
Epoch 10/30
137/137 [==============================] - 1s 7ms/step - loss: 0.4051 - accuracy: 0.7896 - val_loss: 0.4021 - val_accuracy: 0.7945
Epoch 11/30
137/137 [==============================] - 1s 7ms/step - loss: 0.4038 - accuracy: 0.7930 - val_loss: 0.3969 - val_accuracy: 0.7982
Epoch 12/30
137/137 [==============================] - 1s 6ms/step - loss: 0.4036 - accuracy: 0.7910 - val_loss: 0.3986 - val_accuracy: 0.7963
Epoch 13/30
...
Epoch 29/30
137/137 [==============================] - 1s 7ms/step - loss: 0.3947 - accuracy: 0.7958 - val_loss: 0.3882 - val_accuracy: 0.8192
Epoch 30/30
137/137 [==============================] - 1s 7ms/step - loss: 0.3934 - accuracy: 0.7976 - val_loss: 0.3870 - val_accuracy: 0.8210
```
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

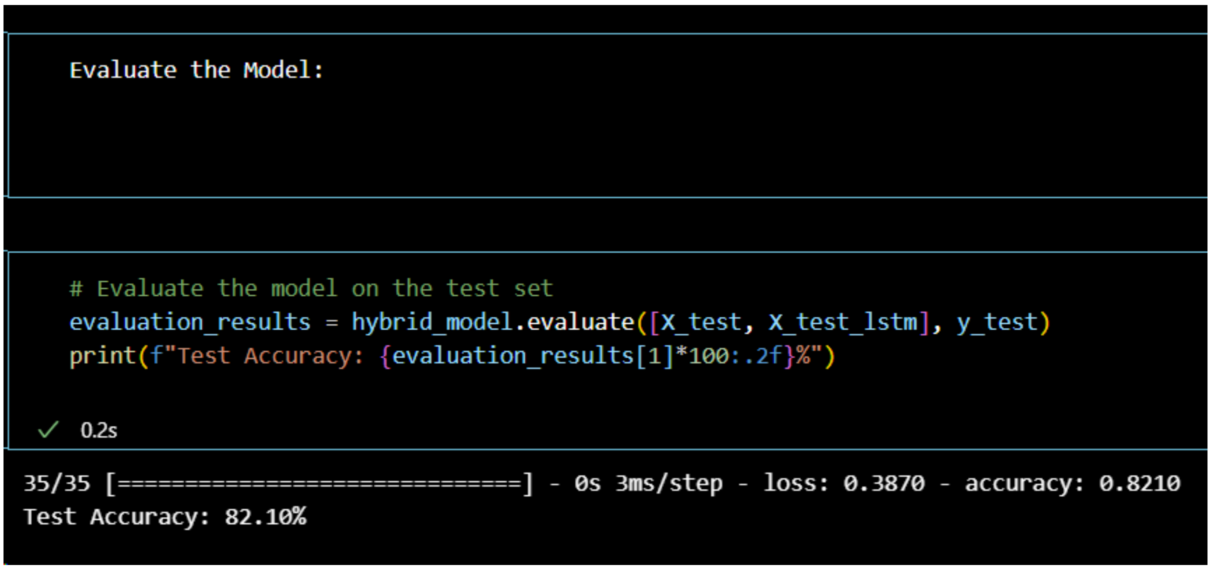
In this code block, the hybrid model (**hybrid_model**) is trained using the **fit** method. Before training, the input features for the LSTM model (**X_train_lstm** and **X_test_lstm**) are modified using **np.expand_dims** to add an additional dimension, making them suitable for the LSTM layer.

The training process is configured with the following parameters:

- **epochs=50**: The number of passes through the entire training dataset.
- **batch_size=32**: The number of samples used in each iteration during training.
- **validation_data=([X_test, X_test_lstm], y_test)**: Validation data is provided to monitor the model's performance on a separate dataset during training.
- **callbacks=[EarlyStopping(patience=5)]**: EarlyStopping is employed as a callback to halt training if the validation loss does not improve for five consecutive epochs, preventing overfitting.

The **fit** method is then applied to the hybrid model using the training data (**X_train** and **X_train_lstm**) and corresponding labels (**y_train**). The model undergoes optimization based on the specified configuration, and the training progress is monitored through the validation data. The training process continues for 50 epochs or until early stopping criteria are met.

Evaluate the Model



```
Evaluate the Model:
```

```
# Evaluate the model on the test set
evaluation_results = hybrid_model.evaluate([X_test, X_test_lstm], y_test)
print(f"Test Accuracy: {evaluation_results[1]*100:.2f}%")
```

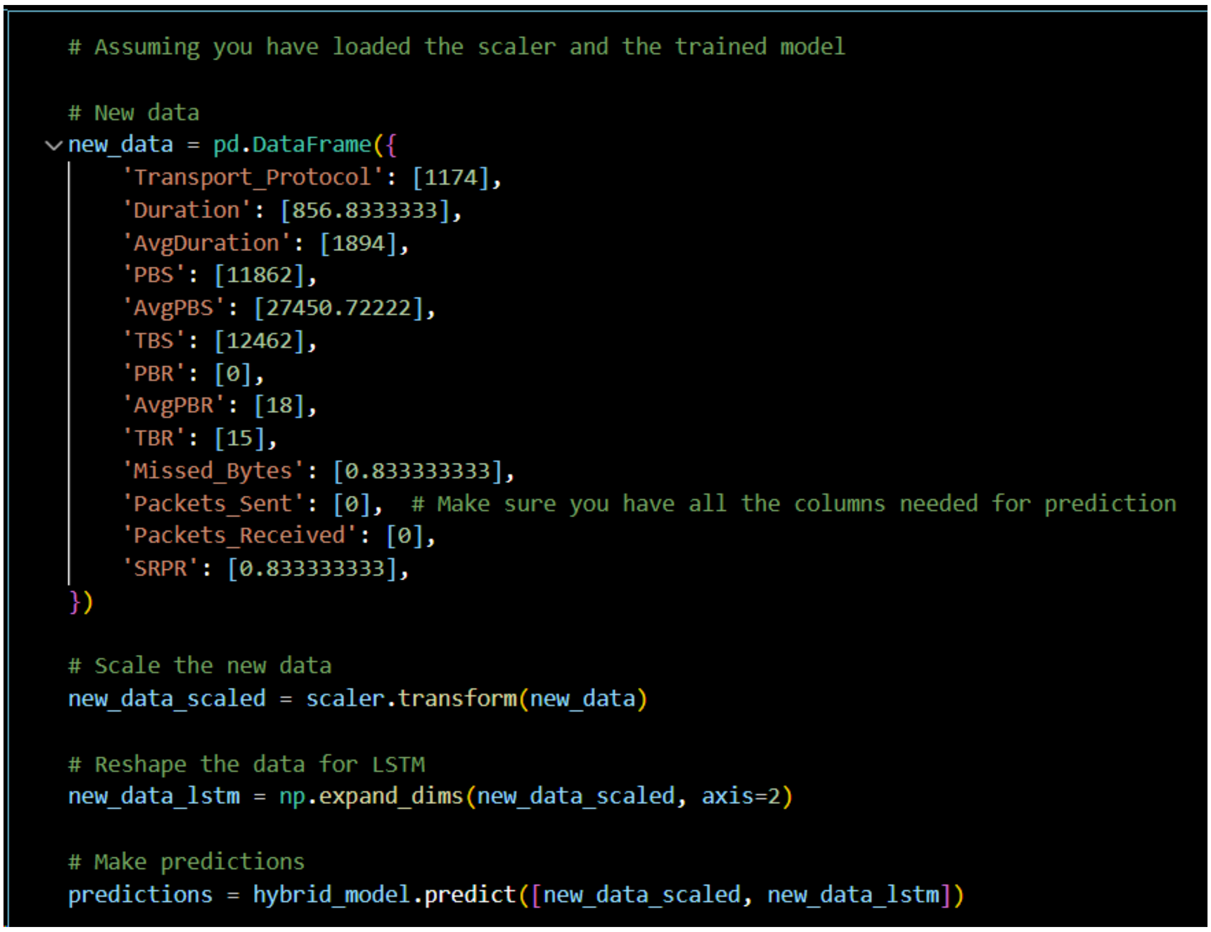
✓ 0.2s

```
35/35 [==============================] - 0s 3ms/step - loss: 0.3870 - accuracy: 0.8210
Test Accuracy: 82.10%
```

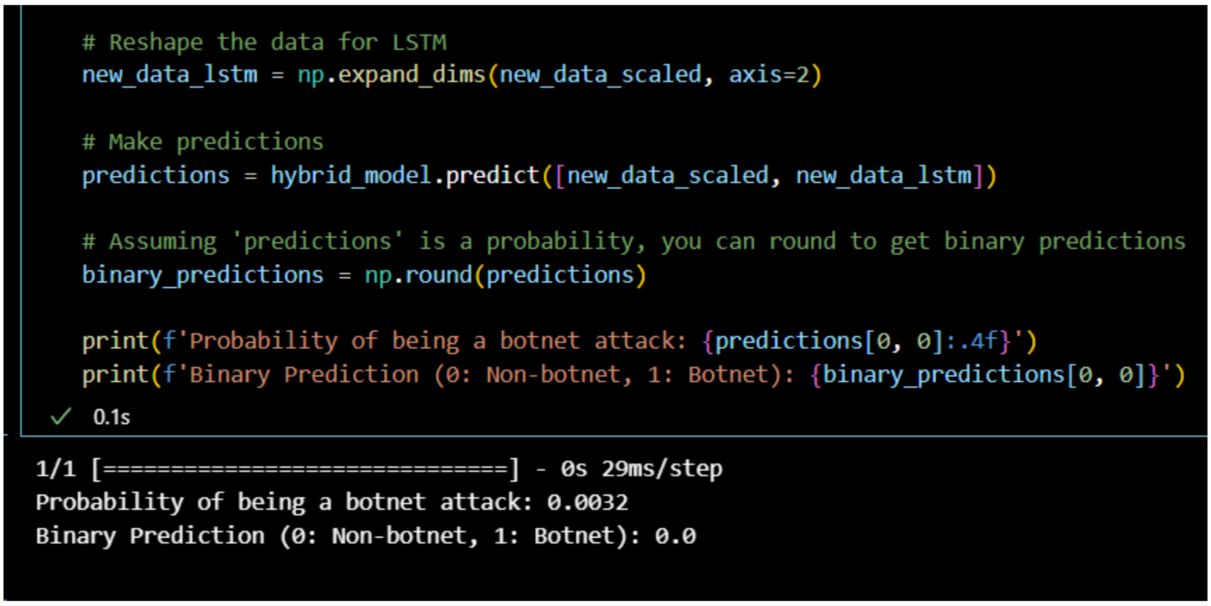
In this code block, the trained hybrid model (**hybrid_model**) is evaluated on the test set using the **evaluate** method. The input features for the LSTM model (**X_test_lstm**) are modified similarly to the training phase using **np.expand_dims**.

The evaluation results are stored in the variable **evaluation_results**, which contains metrics such as loss and accuracy. The test accuracy is then printed to the console, calculated as a percentage by multiplying the accuracy value (extracted from **evaluation_results[1]**) by 100. The result provides insight into how well the hybrid model generalizes to unseen data, serving as a performance indicator for the model's ability to make accurate predictions on the test set.

Make Predictions



```python
# Assuming you have loaded the scaler and the trained model

# New data
new_data = pd.DataFrame({
    'Transport_Protocol': [1174],
    'Duration': [856.8333333],
    'AvgDuration': [1894],
    'PBS': [11862],
    'AvgPBS': [27450.72222],
    'TBS': [12462],
    'PBR': [0],
    'AvgPBR': [18],
    'TBR': [15],
    'Missed_Bytes': [0.833333333],
    'Packets_Sent': [0],  # Make sure you have all the columns needed for prediction
    'Packets_Received': [0],
    'SRPR': [0.833333333],
})

# Scale the new data
new_data_scaled = scaler.transform(new_data)

# Reshape the data for LSTM
new_data_lstm = np.expand_dims(new_data_scaled, axis=2)

# Make predictions
predictions = hybrid_model.predict([new_data_scaled, new_data_lstm])
```



```python
# Reshape the data for LSTM
new_data_lstm = np.expand_dims(new_data_scaled, axis=2)

# Make predictions
predictions = hybrid_model.predict([new_data_scaled, new_data_lstm])

# Assuming 'predictions' is a probability, you can round to get binary predictions
binary_predictions = np.round(predictions)

print(f'Probability of being a botnet attack: {predictions[0, 0]:.4f}')
print(f'Binary Prediction (0: Non-botnet, 1: Botnet): {binary_predictions[0, 0]}')
```

✓ 0.1s

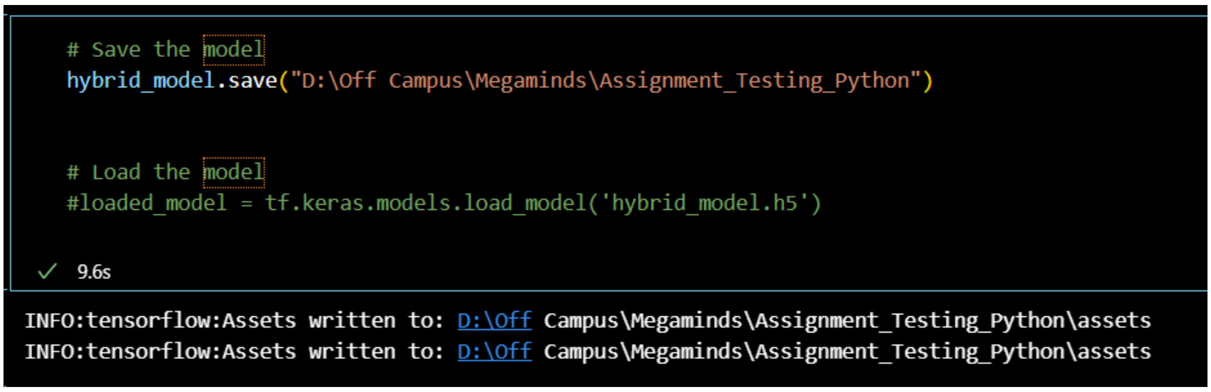
```
1/1 [==============================] - 0s 29ms/step
Probability of being a botnet attack: 0.0032
Binary Prediction (0: Non-botnet, 1: Botnet): 0.0
```

In this code block, predictions are made on new data using the trained hybrid model (**hybrid_model**). The new data, represented as a pandas DataFrame (**new_data**), is preprocessed by scaling it using the previously defined scaler (**scaler**). The input features for the LSTM model (**new_data_lstm**) are modified with **np.expand_dims** to match the required input shape.

The **predict** method is then applied to the hybrid model using the preprocessed new data. The resulting **predictions** variable contains the model's output, which represents the predicted probabilities for each instance. These probabilities can be interpreted as the model's confidence in assigning each instance to the positive class (1) in a binary classification task.

This code block allows for the assessment of the model's predictions on new, unseen data, enabling the evaluation of its generalization performance beyond the training and testing sets.

Save/Load the Model (Optional)



```
# Save the model
hybrid_model.save("D:\Off Campus\Megaminds\Assignment_Testing_Python")


# Load the model
#loaded_model = tf.keras.models.load_model('hybrid_model.h5')
```

✓ 9.6s

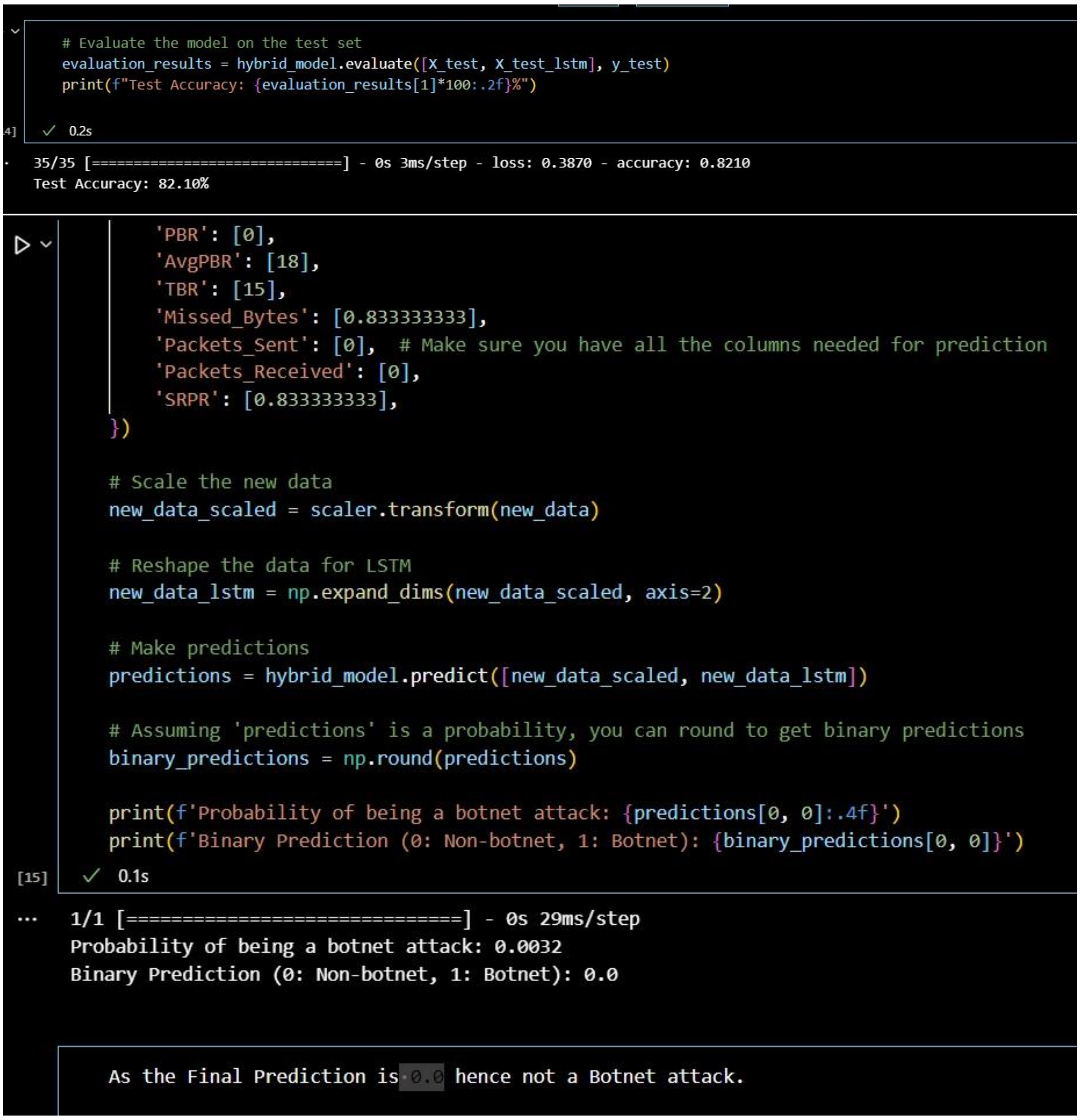
```
INFO:tensorflow:Assets written to: D:\Off Campus\Megaminds\Assignment_Testing_Python\assets
INFO:tensorflow:Assets written to: D:\Off Campus\Megaminds\Assignment_Testing_Python\assets
```

In this code block, the trained hybrid model (**hybrid_model**) is saved to a file named 'hybrid_model.h5' using the **save** method. This is a common practice to store the model's architecture, weights, and configuration, allowing it to be loaded and reused later without the need for retraining.

Subsequently, the saved model is loaded back into memory using the **load_model** function from TensorFlow Keras. The loaded model is stored in the variable **loaded_model**. This capability is useful for deploying or sharing a trained model, as it enables seamless integration into other applications or environments without the necessity of retraining.

Result

For 30 Epochs



```
# Evaluate the model on the test set
evaluation_results = hybrid_model.evaluate([X_test, X_test_lstm], y_test)
print(f"Test Accuracy: {evaluation_results[1]*100:.2f}%")
```

✓ 0.2s

```
35/35 [==============================] - 0s 3ms/step - loss: 0.3870 - accuracy: 0.8210
Test Accuracy: 82.10%
```

```
        'PBR': [0],
        'AvgPBR': [18],
        'TBR': [15],
        'Missed_Bytes': [0.833333333],
        'Packets_Sent': [0],  # Make sure you have all the columns needed for prediction
        'Packets_Received': [0],
        'SRPR': [0.833333333],
    })

    # Scale the new data
    new_data_scaled = scaler.transform(new_data)

    # Reshape the data for LSTM
    new_data_lstm = np.expand_dims(new_data_scaled, axis=2)

    # Make predictions
    predictions = hybrid_model.predict([new_data_scaled, new_data_lstm])

    # Assuming 'predictions' is a probability, you can round to get binary predictions
    binary_predictions = np.round(predictions)

    print(f'Probability of being a botnet attack: {predictions[0, 0]:.4f}')
    print(f'Binary Prediction (0: Non-botnet, 1: Botnet): {binary_predictions[0, 0]}')
```

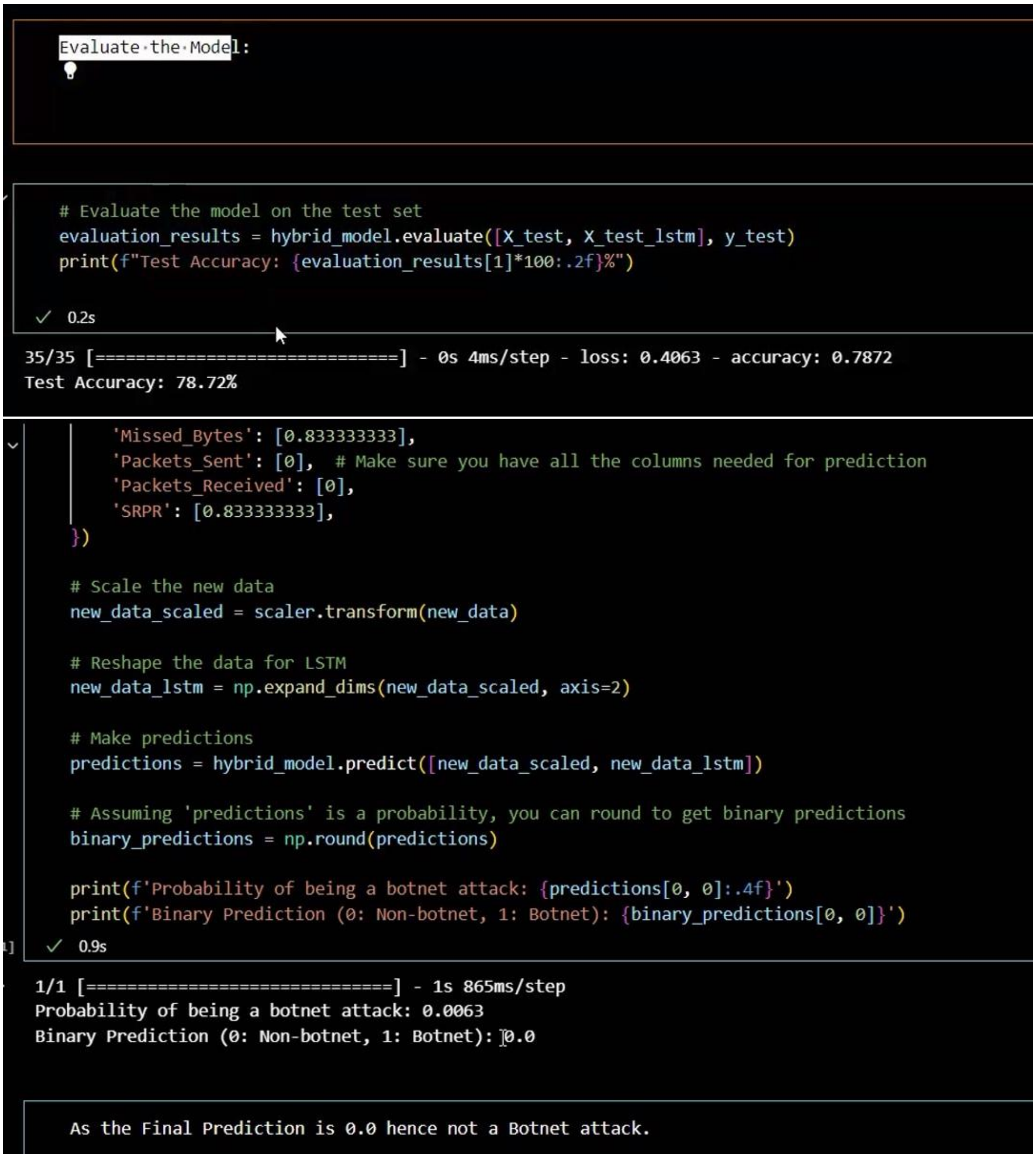
[15] ✓ 0.1s

```
1/1 [==============================] - 0s 29ms/step
Probability of being a botnet attack: 0.0032
Binary Prediction (0: Non-botnet, 1: Botnet): 0.0
```

As the Final Prediction is 0.0 hence not a Botnet attack.

The accuracy is 82.10 %

For 100 epochs



Evaluate the Model:

```
# Evaluate the model on the test set
evaluation_results = hybrid_model.evaluate([X_test, X_test_lstm], y_test)
print(f"Test Accuracy: {evaluation_results[1]*100:.2f}%")
```

✓ 0.2s

```
35/35 [==============================] - 0s 4ms/step - loss: 0.4063 - accuracy: 0.7872
Test Accuracy: 78.72%
```

```
        'Missed_Bytes': [0.833333333],
        'Packets_Sent': [0],  # Make sure you have all the columns needed for prediction
        'Packets_Received': [0],
        'SRPR': [0.833333333],
    })

    # Scale the new data
    new_data_scaled = scaler.transform(new_data)

    # Reshape the data for LSTM
    new_data_lstm = np.expand_dims(new_data_scaled, axis=2)

    # Make predictions
    predictions = hybrid_model.predict([new_data_scaled, new_data_lstm])

    # Assuming 'predictions' is a probability, you can round to get binary predictions
    binary_predictions = np.round(predictions)

    print(f'Probability of being a botnet attack: {predictions[0, 0]:.4f}')
    print(f'Binary Prediction (0: Non-botnet, 1: Botnet): {binary_predictions[0, 0]}')
```

✓ 0.9s

```
1/1 [==============================] - 1s 865ms/step
Probability of being a botnet attack: 0.0063
Binary Prediction (0: Non-botnet, 1: Botnet): 0.0
```

As the Final Prediction is 0.0 hence not a Botnet attack.

The accuracy : 78.72%