## Code Description

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv1D, MaxPooling1D, LSTM, Dropout
```

a) pandas**:** used for reading and processing the dataset.
b) numpy**:** used for handling arrays and numerical operations.
c) train_test_split**:** splits data into training and testing sets.
d) StandardScaler**:** scales features to have mean 0 and unit variance (important for ML).
e) classification_report, confusion_matrix**:** for evaluating model performance.
f) matplotlib & seaborn**:** for plotting graphs and confusion matrix.
g) tensorflow & keras layers**:** for building and training the deep learning model.

```
DATA_PATH = "/content/MH-100K/mh_100k.csv"
df = pd.read_csv(DATA_PATH)
```

a. `DATA_PATH`: points to the CSV file containing MH-100K dataset.
b. `pd.read_csv`: loads the CSV file into a pandas DataFrame.

```
print("Original dataset shape:", df.shape)
print("Columns preview:", df.columns[:15])
print(df.head())
```

Displays dataset shape (rows × columns), first 15 column names, and first 5 rows to understand the data.

```
df['CLASS'] = df['CLASS'].fillna(0)
```

Fills missing values in the `CLASS` column with `0` (assuming missing = benign).

# Code Description

```
y =df['CLASS'].astype(int).values
```

Converts the `CLASS` column to integers (0 = benign, 1 = malware) and stores as NumPy array `y` (target labels).

```
X = df.drop(columns=['Unnamed:0','SHA256','NOME','PACOTE','CLASS'],errors='ignore')
```

Drops unnecessary columns like IDs, names, and the target column `CLASS`.

Remaining columns become the features (`X`).

```
X = X.apply(pd.to_numeric,
```

Converts all feature values to numeric (if some are strings like "0;") and replaces invalid or missing values with 0.

```
print("Features shape:", X.shape, "Labels shape:", y.shape)
```

Prints the shape of `X` (features) and `y` (labels) to confirm preprocessing.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
```

a) Splits the dataset into training (80%) and test (20%) sets.
b) `stratify=y` ensures class balance remains the same in both sets.

```
scaler = StandardScaler(with_mean=False)
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

c) Scales features so they have similar ranges (important for neural networks).
d) `with_mean=False` avoids errors if data is sparse.

```
X_train = X_train.toarray() if hasattr(X_train, "toarray") else X_train
X_test = X_test.toarray() if hasattr(X_test, "toarray") else X_test
```

Converts sparse matrices (if any) to dense NumPy arrays for TensorFlow compatibility.

# Code Description

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

Adds an extra dimension to match CNN/LSTM input shape `(samples, timesteps, features=1)`.

```
print("Reshaped X_train:", X_train.shape)
print("Reshaped X_test:", X_test.shape)
```

Prints the final shape of training and testing data to confirm reshaping.

```
model = Sequential()
```

Creates a sequential neural network model.

```
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(X_train.shape[1], 1)))
```

Adds a **1D convolutional layer** with 64 filters of size 3 to detect patterns in features.

```
model.add(MaxPooling1D(pool_size=2))
```

Reduces feature map size by taking max value in each window (downsampling).

```
model.add(LSTM(64, return_sequences=False))
```

Adds an **LSTM layer** with 64 units to learn sequential dependencies across features.

```
model.add(Dense(128, activation='relu'))
```

Fully connected layer with 128 neurons for feature learning

```
model.add(Dropout(0.3))
```

Randomly drops 30% of neurons during training to prevent overfitting

```
model.add(Dense(1, activation='sigmoid'))
```

Output layer with **sigmoid activation** for binary classification (malware or benign)

# Code Description

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

a. Compiles the model using Adam optimizer and binary cross-entropy loss.
b. Prints model architecture summary.

```
early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
```

Stops training early if validation loss stops improving for 3 epochs and restores best weights.

```
history = model.fit(
    X_train, y_train,
```

a. Trains the model on `X_train`/`y_train` for up to 10 epochs, using 20% of training data for validation.
b. Batch size = 128 samples per update.

```
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print(f"\n✅ Test Accuracy: {acc:.4f}")
```

Evaluates trained model on the test set and prints accuracy

```
y_pred_prob = model.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)
```

Gets predicted probabilities and converts them to class labels (0/1)

```
print("\n🔷 Classification Report:")
print(classification_report(y_test, y_pred, digits=4))
Prints precision, recall, F1-score, and support for each class.
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
        xticklabels=['Benign','Malware'],
        yticklabels=['Benign','Malware'])
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()
```

# Code Description

Computes and plots a confusion matrix heatmap for visualizing model predictions

```
plt.figure(figsize=(8,5))
plt.plot(history.history['accuracy'], label='Train Accuracy', marker='o')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy', marker='o')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```

Plots training vs validation accuracy over epochs to check learning progress

```
plt.figure(figsize=(8,5))
plt.plot(history.history['loss'], label='Train Loss', marker='o')
plt.plot(history.history['val_loss'], label='Validation Loss',
marker='o')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

Plots training vs validation loss over epochs to detect overfitting

```
model.save("mh100k_lstm_cnn_model.h5")
print("✅ Model saved as mh100k_lstm_cnn_model.h5")
```

Saves trained model as an HDF5 file (`.h5`) for future use (loading, inference, fine-tuning)

# Code Description

## Simplified Picture