Keystroke Behavioral Biometrics for User Authentication

In recent years, user authentication methods have seen a shift towards biometric recognition and behavioral analysis, largely due to the limitations and methodological issues associated with traditional approaches. The rise of neural networks has made these biometric methods more accessible and practical than ever before. Often, these new authentication procedures complement or even replace conventional password-based systems.

Biometric recognition encompasses a range of distinctive physical characteristics unique to individual users, such as iris patterns, facial features, fingerprints, and voice signatures. Additionally, behavioral patterns and habits can serve as reliable identifiers. Among these biometric behaviors, keystroke typing (also known as keystroke dynamics) has emerged as a promising method for user identification, much like handwriting analysis.

One of the pioneering implementations of keystroke dynamics for user authentication was Coursera's Signature Track technology. This system was developed to verify the identities of students participating in online course assessments, ensuring the validity of their statements of accomplishment upon course completion.

The Signature Track technology, as described in a paper published in the Ubiquity Symposium, aimed to solve the challenge of assigning verifiable credentials to each student. This process links a student's coursework to their real-world identity, resulting in a verified certificate issued by Coursera and the partnering university. These certificates include a unique verification code, allowing third parties such as potential employers to confirm the authenticity of the course completion.

Signature Track's distinctive features extend beyond mere authentication and identity verification. The system is designed to handle the massive scale of Coursera's user base, with typical courses enrolling between 40,000 and 60,000 students. This necessitates highly efficient verification procedures that can operate without manual intervention from instructors or staff.

Unlike other web services, Coursera faced an additional challenge: the temptation for users to share their login credentials with others who could complete assignments on their behalf. To address this, Coursera implemented a dual authentication system combining facial recognition and keystroke dynamics. In addition, during the enrollment phase, the student is asked to write a short sentence on their keyboard so that their own biometric keystroke profile can be recognized. This is done using keystroke dynamics.

Keystroke dynamics analyzes the unique cadence and rhythm of a user's typing pattern. However, raw keystroke data can be influenced by various external factors such as interruptions, error corrections, or the use of special keys. To create a reliable dataset, this raw data must be transformed into a set of features that accurately represent the user's typing dynamics while filtering out random disturbances.

One of the earliest scientific studies on using keystroke dynamics for anomaly detection was conducted by Kevin S. Killourhy and Roy A. Maxion. Their research involved collecting keystroke data from 51 subjects typing 400 passwords each. This data was then analyzed using 14 different algorithms to evaluate user detection performance. The primary goal was to identify impostors attempting to use stolen passwords by recognizing differences in typing patterns.

Some of the key features used to determine keystroke dynamics include:

- Keydown-keydown: The time between consecutive key presses
- Keyup-keydown: The time between releasing one key and pressing the next
- Hold: The duration of each key press

These timing features, extracted from raw data, form the basis for user detection algorithms, enabling more secure and reliable authentication systems based on individual typing behaviors.

<u>**Code for Implementing Keystroke Behavioral Analysis**</u>

The following is an example of the implementation of keystroke dynamics based on the dataset described in the study *Comparing anomaly-detection for keystroke dynamics* mentioned earlier. The dataset is also available for download in .csv format at https://github.com/avnishnaithani/pythonforcybersecurity/blob/main/Capstone%20Projects/DSL-StrongPasswordData.csv

The dataset consists of 51 subjects, each typing 400 passwords. Also, among the measures collected are these hold times (represented in the dataset with the label H):

Keydown-keydown time (labeled as DD)

Keyup-keydown time (labeled as UD)

The **DSL-StrongPasswordData.csv** dataset captures timing-related features of typing dynamics, which are useful in biometric identification (i.e., identifying individuals based on their typing patterns). This dataset is often used for studying **keystroke dynamics**, particularly when typing passwords. Here's a breakdown of how the features relate to typing patterns and how they help in this program.

Key Features in the Dataset

In the **DSL-StrongPasswordData.csv** dataset, the following feature types are typically included:

1. **Subject**:

- This column identifies the person or the subject typing the password.
- The goal is to classify different subjects based on how they type the password.

2. **H (Hold Time)**:

- These features are labeled like H.period, H.t, etc., where each feature represents the **hold time** of a specific key.
- **Hold time** (or dwell time) refers to the duration a key is pressed down (i.e., time between pressing and releasing the key).
- For example, H.period refers to the time the subject holds the . key, H.t refers to how long the t key is pressed, and so on.

3. **DD (Key Down to Key Down)**:

- These features are labeled like DD.t.h, DD.h.e, etc.
- **DD** represents the time between pressing two consecutive keys (i.e., the difference between the time the first key was pressed down and the time the next key is pressed).
- For example, DD.t.h is the time between pressing the t key and the h key. This helps capture the subject's typing rhythm or speed between key presses.

4. **UD (Key Up to Key Down)**:

- These features are labeled like UD.t.h, UD.h.e, etc.
- **UD** is the time between releasing one key and pressing the next key.
- For example, UD.t.h is the time between releasing the t key and pressing the h key.

Mapping Keystrokes to Features

When a subject types a password, the following events are recorded:

- **Key press events**: The moment a key is pressed.
- **Key release events**: The moment a key is released.

From these events, three key timing features can be extracted:

- **Hold Time (H)**: Time between pressing and releasing a key.
- **Down-Down (DD)**: Time between pressing two consecutive keys.
- **Up-Down (UD)**: Time between releasing one key and pressing the next key.

These features, which are computed for all relevant keys, make up the columns in the dataset, except for the subject column.

How the Dataset Helps in the Program

1. **Feature Extraction and Grouping**:

   In the program, the columns that begin with DD are extracted and grouped by subject (plot.groupby('subject').mean()), representing the **average keystroke latency** for each subject. This helps to visualize typing dynamics across different users.

2. **Model Training and Testing**:

- The dataset is split into training and testing sets (train_test_split). The features used for training (e.g., H, DD, UD values) are the time-based metrics, and the label (target variable) is the subject.

- Various classifiers (K-Nearest Neighbors, SVM, Multi-Layer Perceptron) are trained using the features to classify different users based on their keystroke dynamics. The goal is to correctly identify which subject typed the password based on the provided features.

3. **Model Evaluation**:

- The **accuracy** of each classifier is calculated by comparing the predicted subject labels (e.g., using metrics.accuracy_score) with the actual labels in the test set.
- The **confusion matrix** provides insight into which subjects are easily confused with others by the model, indicating the overall effectiveness of the classifier.

This dataset helps to train machine learning models to distinguish between different users based on their keystroke patterns, providing a biometric-based identification approach. By learning how each subject types the password, models can generalize and identify users even with unseen typing data.

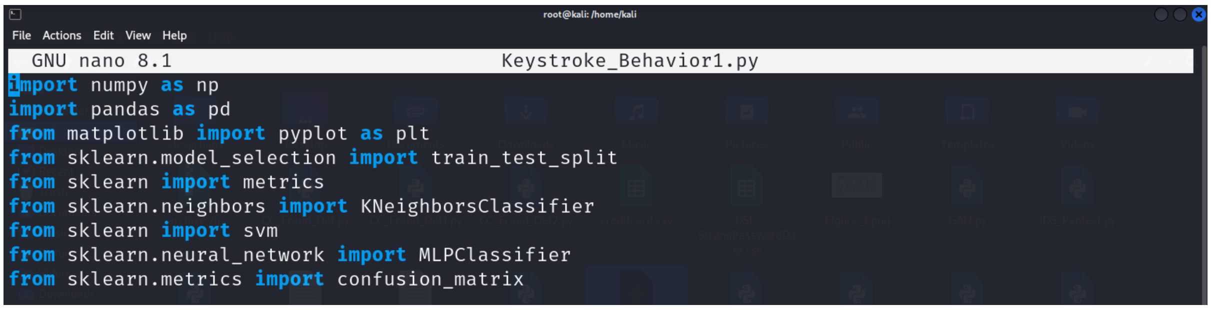
Open the nano editor to write the code using:

```
$ nano Keystroke_Behaviour1.py
```

```
┌──(root㉿kali)-[/home/kali]
└─# nano Keystroke_Behavior1.py
```

The code is available at https://github.com/avnishnaithani/pythonforcybersecurity/blob/main/Capstone%20Projects/Keystroke_Behavior1.py

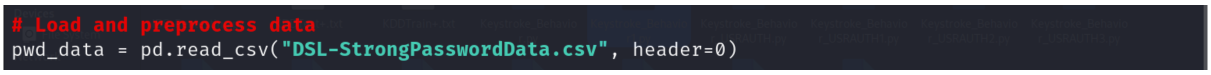
Step 1 - Importing libraries



```
root@kali: /home/kali
File  Actions  Edit  View  Help
  GNU nano 8.1                          Keystroke_Behavior1.py
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.neighbors import KNeighborsClassifier
from sklearn import svm
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import confusion_matrix
```

This section imports all the necessary libraries. NumPy and Pandas are for data manipulation, Matplotlib for plotting, and the rest are from Scikit-learn for machine learning tasks.

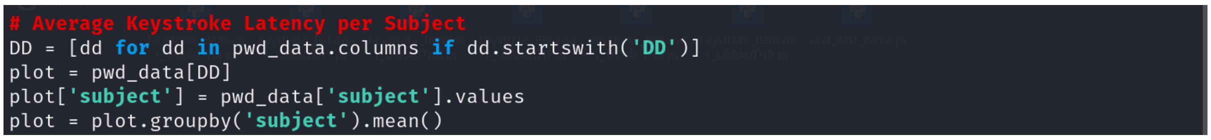
Step 2 - Loading the data



```
# Load and preprocess data
pwd_data = pd.read_csv("DSL-StrongPasswordData.csv", header=0)
```

This line reads a CSV file named "DSL-StrongPasswordData.csv" into a Pandas DataFrame called pwd_data. Each row in this dataset corresponds to a subject's typing pattern, which consists of various latency measurements and an associated subject identifier. The header=0 argument tells Pandas to use the first row as column names.

Step 3 - Preparing data for plotting



```
# Average Keystroke Latency per Subject
DD = [dd for dd in pwd_data.columns if dd.startswith('DD')]
plot = pwd_data[DD]
plot['subject'] = pwd_data['subject'].values
plot = plot.groupby('subject').mean()
```

This creates a new DataFrame plot with only the columns that start with 'DD' (representing keystroke data - Time between pressing two consecutive keys). It then adds a 'subject' column and calculates the mean values for each subject.

Step 4 - Plotting average keystroke latency

```
plot.iloc[:6].T.plot(figsize=(8, 6), title='Average Keystroke Latency per Subject')
plt.show()
```

This creates a line plot of the average keystroke latency for the first 6 subjects.

Step 5 - Splitting the data

```
# Split data
data_train, data_test = train_test_split(pwd_data, test_size=0.2, random_state=0)
X_train = data_train[pwd_data.columns[2:]]
y_train = data_train['subject']
X_test = data_test[pwd_data.columns[2:]]
y_test = data_test['subject']
```

This splits the data into training (80%) and testing (20%) sets. It then separates the features (X) and the target variable (y) for both sets.

X_train and X_test: These are the feature sets, which exclude the first two columns (e.g., subject and another feature). It assumes the keystroke latencies are the remaining columns.

y_train and y_test: These are the labels (subjects).

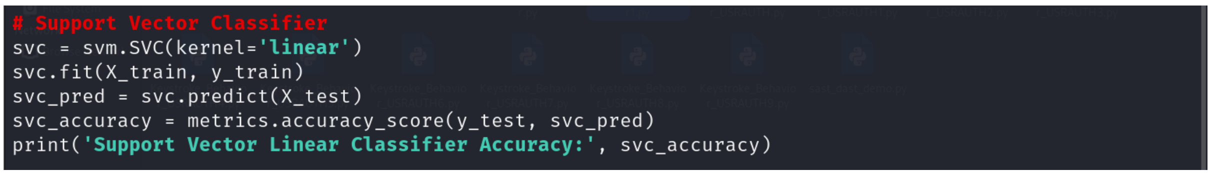
Step 6 – Training and Testing using different Machine Learning Models

K-Nearest Neighbour Classifier

```
# K-Nearest Neighbor Classifier
knc = KNeighborsClassifier()
knc.fit(X_train, y_train)
knc_pred = knc.predict(X_test)
knc_accuracy = metrics.accuracy_score(y_test, knc_pred)
print('K-Nearest Neighbor Classifier Accuracy:', knc_accuracy)
```

This creates a K-Nearest Neighbour model, trains it on the training data, makes predictions on the test data, and prints the accuracy.

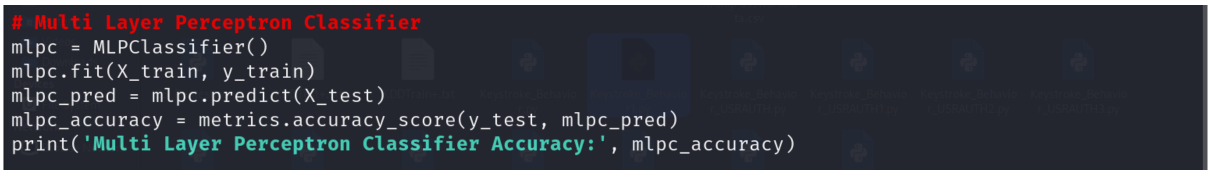
Support Vector Classifier



```
# Support Vector Classifier
svc = svm.SVC(kernel='linear')
svc.fit(X_train, y_train)
svc_pred = svc.predict(X_test)
svc_accuracy = metrics.accuracy_score(y_test, svc_pred)
print('Support Vector Linear Classifier Accuracy:', svc_accuracy)
```

Same steps as KNN classifier done here.

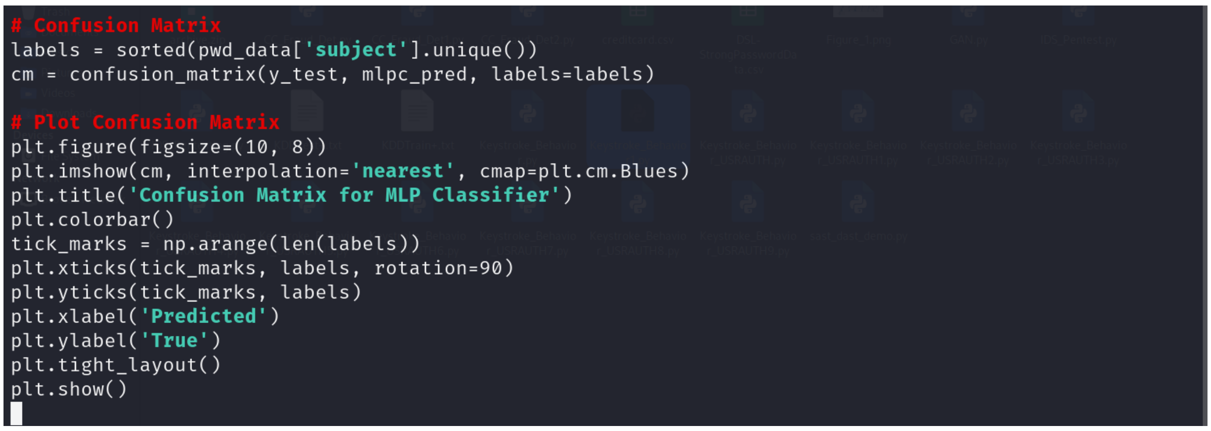
Multi Layer Perceptron Classifier



```
# Multi Layer Perceptron Classifier
mlpc = MLPClassifier()
mlpc.fit(X_train, y_train)
mlpc_pred = mlpc.predict(X_test)
mlpc_accuracy = metrics.accuracy_score(y_test, mlpc_pred)
print('Multi Layer Perceptron Classifier Accuracy:', mlpc_accuracy)
```

Same steps as KNN classifier done here.

Step 7 – Preparing and Plotting Confusion Matrix



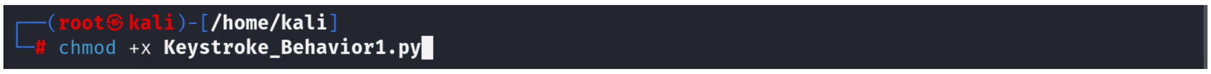
```
# Confusion Matrix
labels = sorted(pwd_data['subject'].unique())
cm = confusion_matrix(y_test, mlpc_pred, labels=labels)

# Plot Confusion Matrix
plt.figure(figsize=(10, 8))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix for MLP Classifier')
plt.colorbar()
tick_marks = np.arange(len(labels))
plt.xticks(tick_marks, labels, rotation=90)
plt.yticks(tick_marks, labels)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.tight_layout()
plt.show()
```

This gets the unique subject labels and creates a confusion matrix for the MLP classifier predictions and creates a visual representation of the confusion matrix, showing how well the MLP classifier performed in predicting each subject.

Save the file and close it. (CTRL+x -> y -> Enter)

Convert the file to an executable one using:

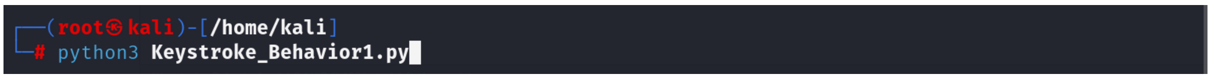
$ **`chmod +x Keystroke_Behaviour1.py`**



```
┌──(root㉿kali)-[/home/kali]
└─# chmod +x Keystroke_Behavior1.py
```

Finally run the program using the command:

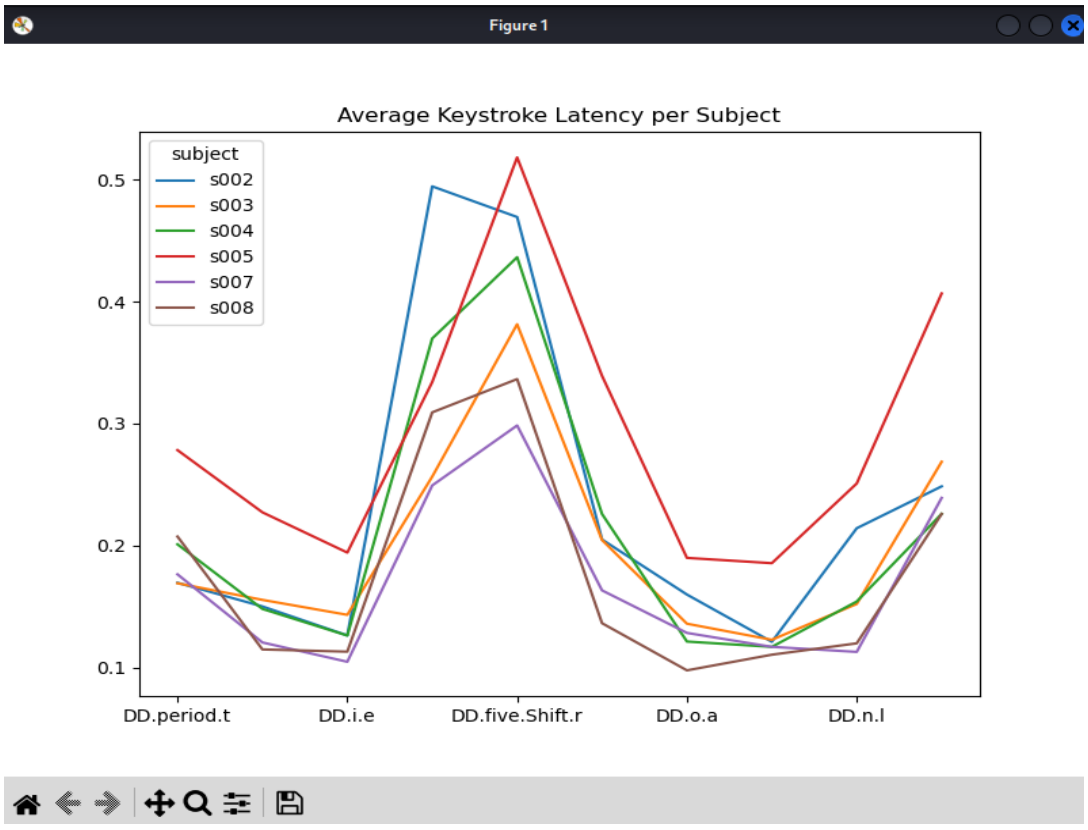
$ **`python3 Keystroke_Behaviour1.py`**



```
┌──(root㉿kali)-[/home/kali]
└─# python3 Keystroke_Behavior1.py
```

Output

The output obtained is shown below:



This graph shows the average keystroke latency for six different subjects (s002, s003, s004, s005, s007, and s008) across five different keystroke measures:

1. DD.period.t
2. DD.i.e
3. DD.five.Shift.r
4. DD.o.a
5. DD.n.l

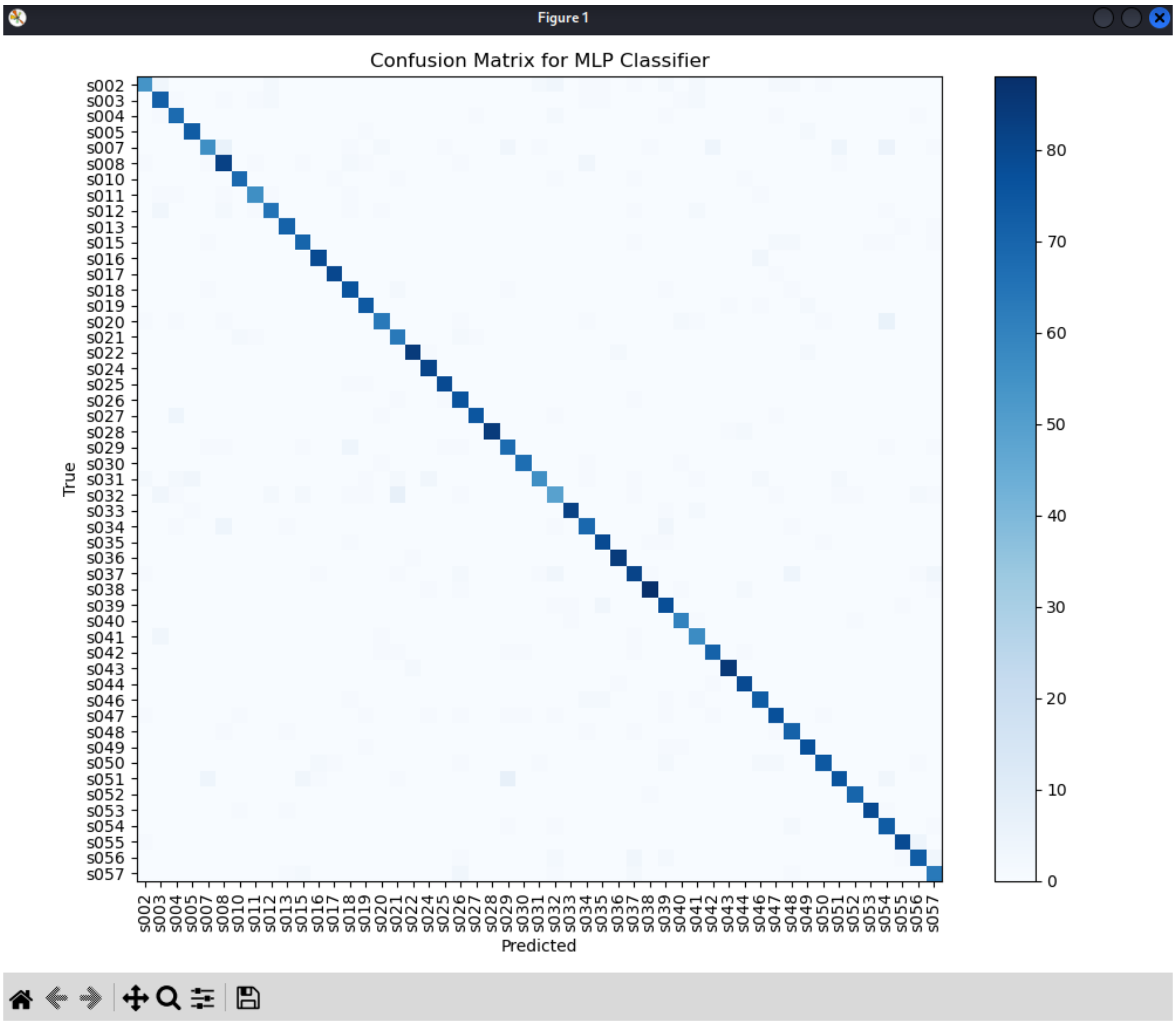
The y-axis represents the latency time (likely in seconds), while the x-axis shows the different keystroke measures.

Key observations:

- Each subject has a unique pattern of keystroke latencies.
- The "DD.five.Shift.r" measure shows the highest latency for most subjects, possibly indicating the time taken to press the Shift key and then 'r'.
- Subject s005 (red line) shows notably different patterns from others, particularly for "DD.five.Shift.r" and "DD.n.l".
- The latencies vary significantly between different keystroke combinations for each subject.

This graph demonstrates how keystroke dynamics differ between individuals, which is the basis for using this method for user authentication.

Confusion Matrix for MLP Classifier



This is a confusion matrix showing the performance of the Multi-Layer Perceptron (MLP) classifier in predicting the subjects based on their keystroke dynamics.

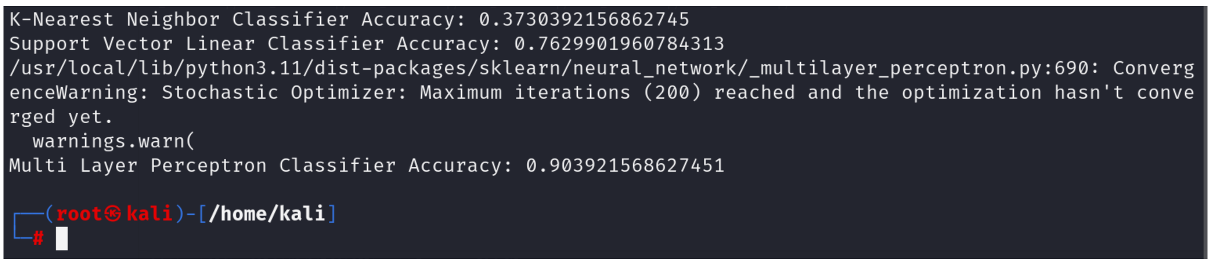
- The y-axis shows the "True" labels (actual subjects).
- The x-axis shows the "Predicted" labels (classifier's predictions).
- The color intensity represents the number of predictions: darker blue indicates more predictions in that cell.

Key observations:

- The strong diagonal line of dark blue squares indicates that the classifier is performing well, correctly identifying most subjects.

- There are very few off-diagonal colored squares, which would represent misclassifications.
- The matrix includes all 51 subjects (s002 to s057), showing the classifier's performance across the entire dataset.
- The overall dark diagonal and light off-diagonal areas suggest high accuracy in subject identification.

Machine Learning Classifier Accuracies



```
K-Nearest Neighbor Classifier Accuracy: 0.3730392156862745
Support Vector Linear Classifier Accuracy: 0.7629901960784313
/usr/local/lib/python3.11/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:690: Converg
enceWarning: Stochastic Optimizer: Maximum iterations (200) reached and the optimization hasn't conve
rged yet.
  warnings.warn(
Multi Layer Perceptron Classifier Accuracy: 0.903921568627451

┌──(root㉿kali)-[/home/kali]
└─# 
```

This image shows the accuracy scores for three different classifiers:

1. K-Nearest Neighbor Classifier Accuracy: 0.3730392156862745 (37.30%)
2. Support Vector Linear Classifier Accuracy: 0.7629901960784313 (76.30%)
3. Multi Layer Perceptron Classifier Accuracy: 0.9039215686274511 (90.39%)

Key observations:

- The Multi Layer Perceptron (MLP) classifier performed the best with about 90.39% accuracy.
- The Support Vector Machine (SVM) with a linear kernel came second with 76.30% accuracy.
- The K-Nearest Neighbor (KNN) classifier performed poorest with 37.30% accuracy.
- There's a warning message indicating that the MLP classifier reached the maximum number of iterations (200) without fully

converging, suggesting that with more iterations or tuning, its performance might improve further.

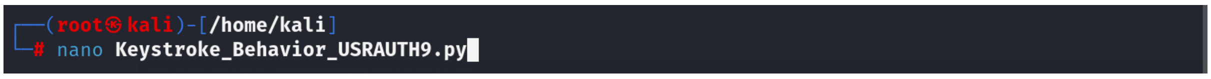
These results demonstrate that keystroke dynamics can indeed be used effectively for user authentication, with the MLP classifier showing particularly strong performance in distinguishing between different users based on their typing patterns.

Implementation Code

We have seen in the previous section that the performance of MLP classifier is the best for the purpose of user keystroke behavior analysis. So, we will implement this project using the MLP classifier.

Open the nano editor to write the code using the nano commnad:

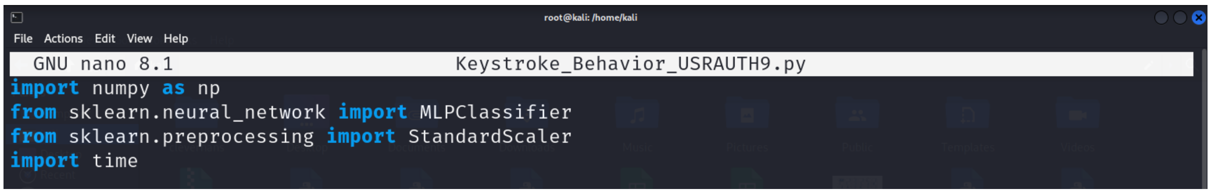
$ nano Keystroke_Behaviour_USRAUTH9.py



```
┌──(root㉿kali)-[/home/kali]
└─# nano Keystroke_Behavior_USRAUTH9.py
```

Start writing the code here. This can be obtained from https://github.com/avnishnaithani/pythonforcybersecurity/blob/main/Capstone%20Projects/Keystroke_Behavior_USRAUTH9.py

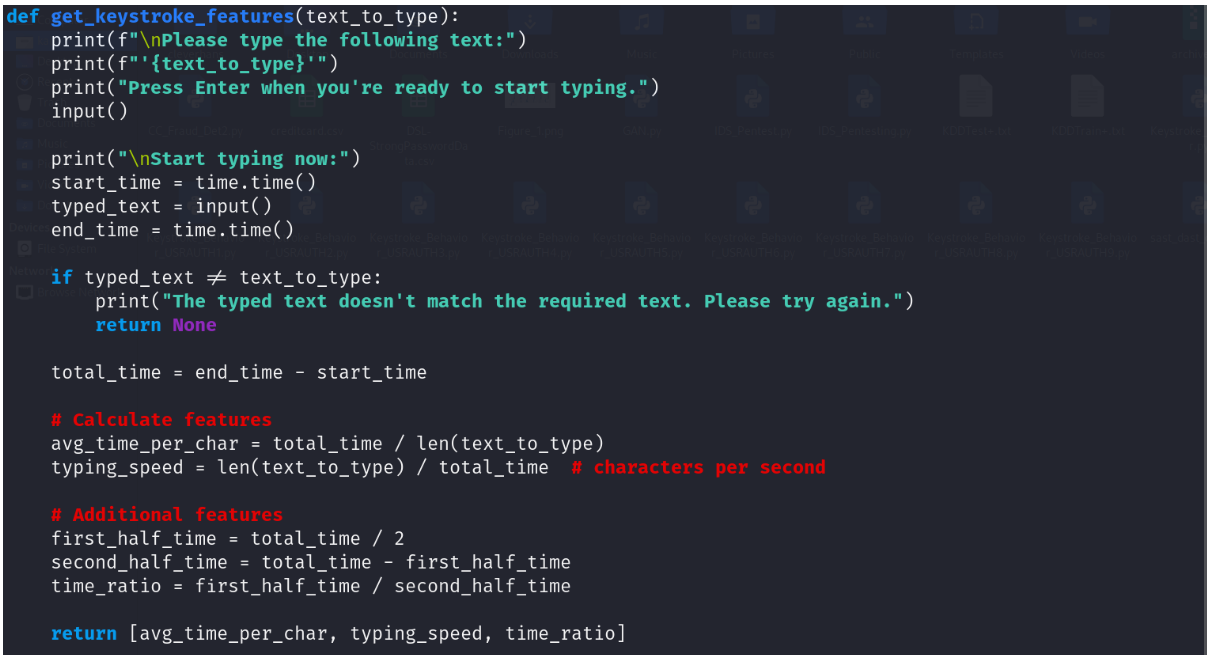
Step 1 – Import necessary libraries



```
  GNU nano 8.1                        Keystroke_Behavior_USRAUTH9.py
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
import time
```

numpy for numerical operations, MLPClassifier for neural network-based classification, StandardScaler for feature scaling, and time for measuring typing duration.

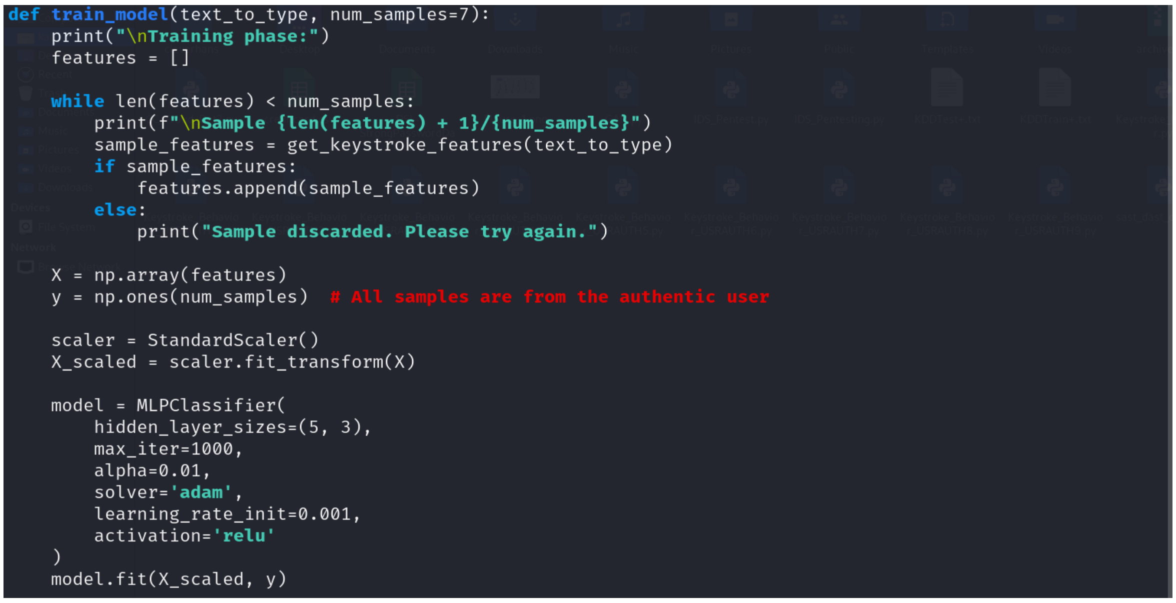
Step 2 – Getting keystroke features



```
def get_keystroke_features(text_to_type):
    print(f"\nPlease type the following text:")
    print(f"'{text_to_type}'")
    print("Press Enter when you're ready to start typing.")
    input()

    print("\nStart typing now:")
    start_time = time.time()
    typed_text = input()
    end_time = time.time()

    if typed_text ≠ text_to_type:
        print("The typed text doesn't match the required text. Please try again.")
        return None

    total_time = end_time - start_time

    # Calculate features
    avg_time_per_char = total_time / len(text_to_type)
    typing_speed = len(text_to_type) / total_time  # characters per second

    # Additional features
    first_half_time = total_time / 2
    second_half_time = total_time - first_half_time
    time_ratio = first_half_time / second_half_time

    return [avg_time_per_char, typing_speed, time_ratio]
```

- Prompts the user to type a specific text.
- Measures the time taken to type the text.
- Calculates features: a) Average time per character b) Typing speed (characters per second) c) Time ratio (first half of typing time / second half)
- Returns these features as a list.

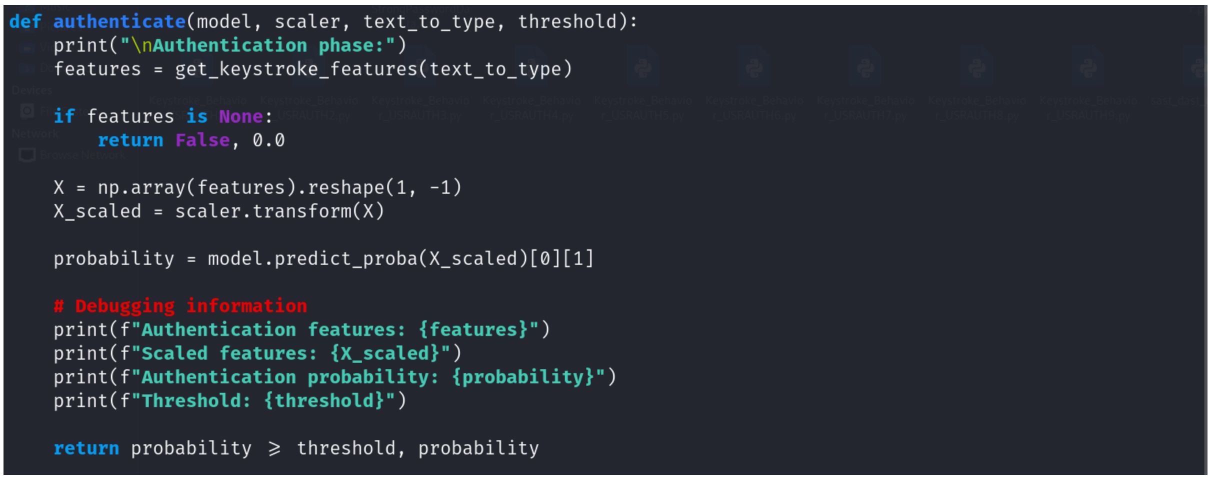
Step 3 – Training the model



```python
def train_model(text_to_type, num_samples=7):
    print("\nTraining phase:")
    features = []

    while len(features) < num_samples:
        print(f"\nSample {len(features) + 1}/{num_samples}")
        sample_features = get_keystroke_features(text_to_type)
        if sample_features:
            features.append(sample_features)
        else:
            print("Sample discarded. Please try again.")

    X = np.array(features)
    y = np.ones(num_samples)  # All samples are from the authentic user

    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    model = MLPClassifier(
        hidden_layer_sizes=(5, 3),
        max_iter=1000,
        alpha=0.01,
        solver='adam',
        learning_rate_init=0.001,
        activation='relu'
    )
    model.fit(X_scaled, y)
```

- Collects multiple samples (default 7) of the user typing the same text.
- Creates a feature matrix (X) and label vector (y).
- Scales the features using StandardScaler.
- Initializes and trains an MLPClassifier (neural network) on this data.
- Returns the trained model and scaler.

Step 4 – Authentication of keystroke behavior

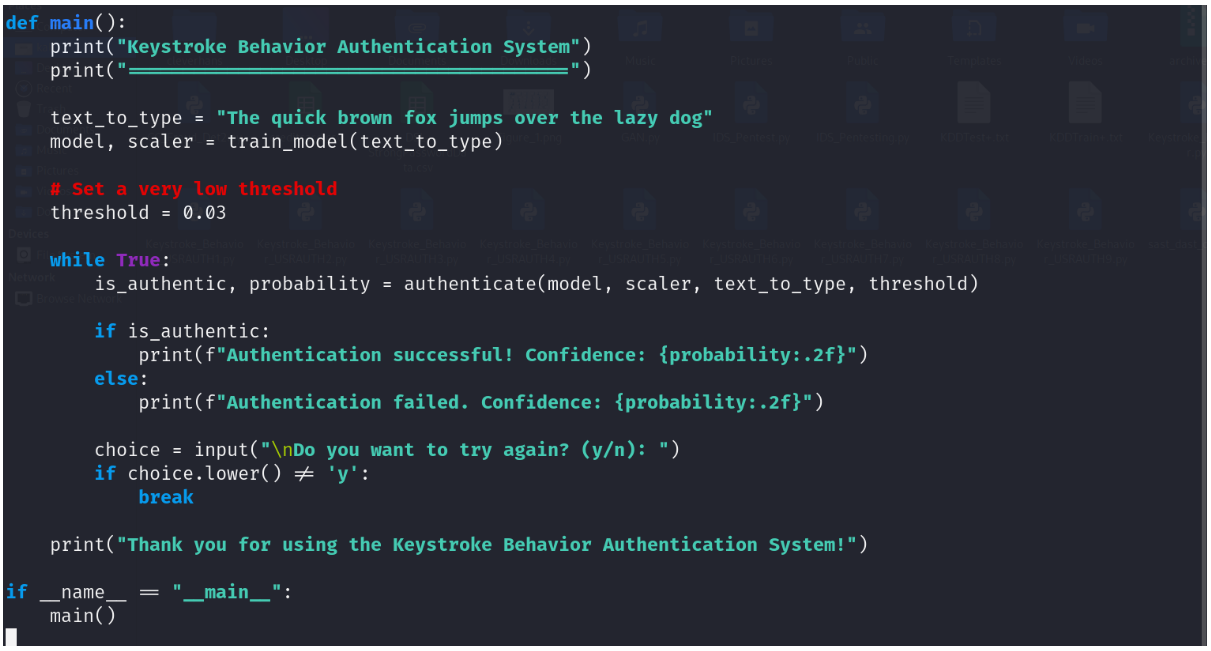


```python
def authenticate(model, scaler, text_to_type, threshold):
    print("\nAuthentication phase:")
    features = get_keystroke_features(text_to_type)

    if features is None:
        return False, 0.0

    X = np.array(features).reshape(1, -1)
    X_scaled = scaler.transform(X)

    probability = model.predict_proba(X_scaled)[0][1]

    # Debugging information
    print(f"Authentication features: {features}")
    print(f"Scaled features: {X_scaled}")
    print(f"Authentication probability: {probability}")
    print(f"Threshold: {threshold}")

    return probability ≥ threshold, probability
```

- Collects a single sample of typing for authentication.

- Scales the features using the same scaler from training.
- Uses the trained model to predict the probability of the sample being from the authentic user.
- Compares this probability to a threshold to determine authentication success.
- Returns the authentication result and probability.

Step 5 - Main Function



```python
def main():
    print("Keystroke Behavior Authentication System")
    print("========================================")

    text_to_type = "The quick brown fox jumps over the lazy dog"
    model, scaler = train_model(text_to_type)

    # Set a very low threshold
    threshold = 0.03

    while True:
        is_authentic, probability = authenticate(model, scaler, text_to_type, threshold)

        if is_authentic:
            print(f"Authentication successful! Confidence: {probability:.2f}")
        else:
            print(f"Authentication failed. Confidence: {probability:.2f}")

        choice = input("\nDo you want to try again? (y/n): ")
        if choice.lower() ≠ 'y':
            break

    print("Thank you for using the Keystroke Behavior Authentication System!")

if __name__ = "__main__":
    main()
```

- Sets up the text to be typed: "The quick brown fox jumps over the lazy dog".
- Trains the model by calling train_model.
- Sets a very low authentication threshold (0.03).
- Enters a loop where it: a) Attempts to authenticate the user. b) Displays the result and confidence. c) Asks if the user wants to try again.

Key Points:

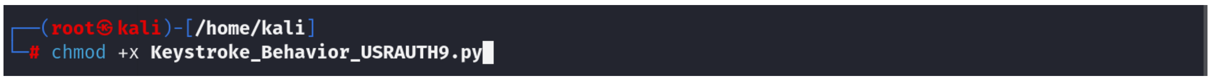
- The system uses a neural network (MLPClassifier) to learn the user's typing patterns.

- It considers timing-based features (average time per character, typing speed, and timing distribution) to characterize the typing behavior.
- The authentication is probabilistic, comparing the model's confidence to a threshold.
- The very low threshold (0.03) suggests high sensitivity, potentially leading to more false positives but fewer false negatives.

Save the file and close it. (CTRL+x -> y -> Enter)

Convert the file to an executable one using:

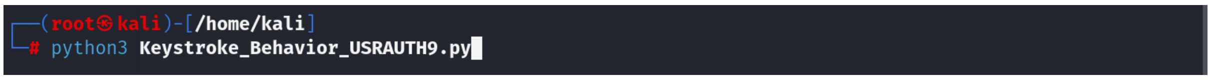
```
$ chmod +x Keystroke_Behaviour_USRAUTH9.py
```



```
┌──(root㉿kali)-[/home/kali]
└─# chmod +x Keystroke_Behavior_USRAUTH9.py
```

Finally run the program using the command:

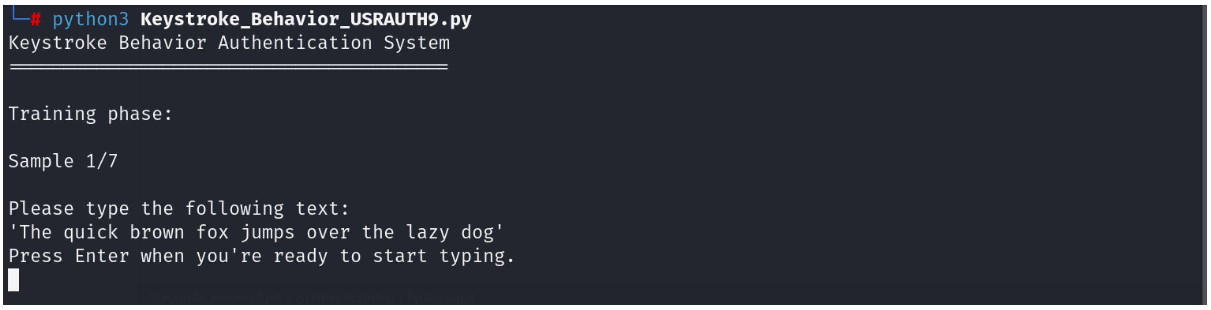
```
$ python3 Keystroke_Behaviour_USRAUTH9.py
```



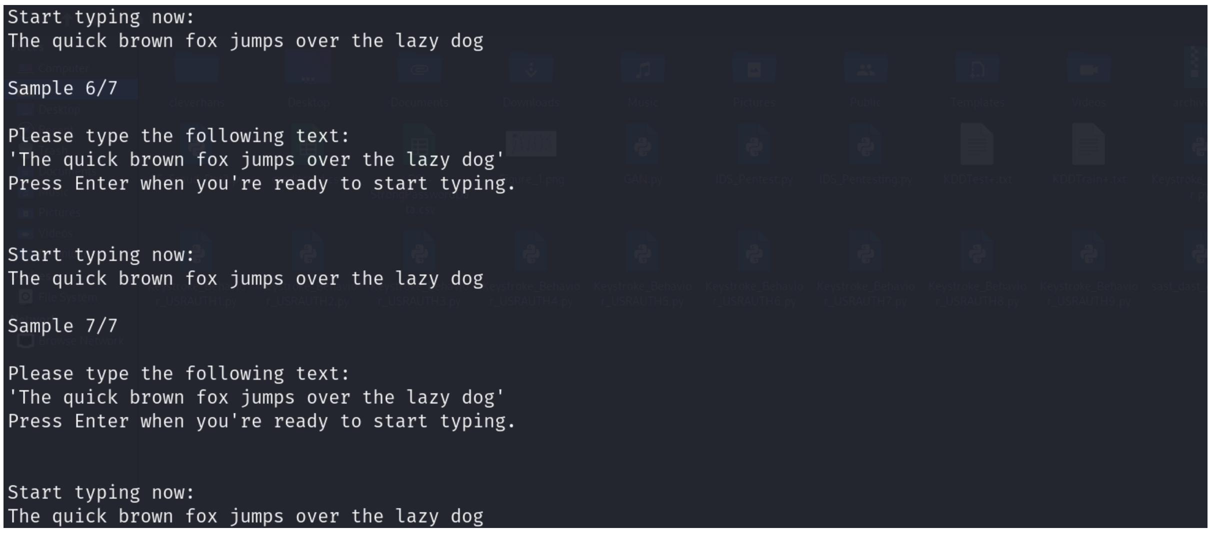
```
┌──(root㉿kali)-[/home/kali]
└─# python3 Keystroke_Behavior_USRAUTH9.py
```

<u>Output</u>

- The script starts by training the model with multiple samples from the user. It will ask to enter the phrase ‘The quick brown fox jumps over the lazy dog’ 7 times

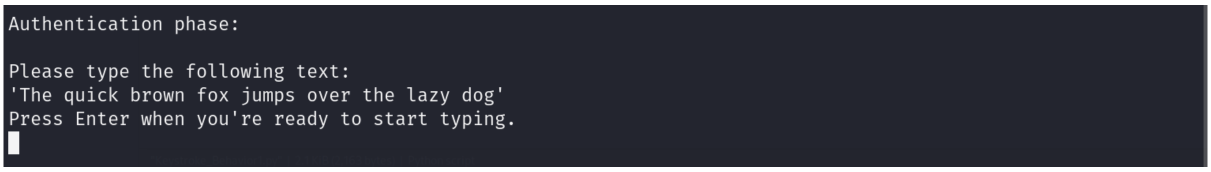


```
└─# python3 Keystroke_Behavior_USRAUTH9.py
Keystroke Behavior Authentication System
════════════════════════════════════════

Training phase:

Sample 1/7

Please type the following text:
'The quick brown fox jumps over the lazy dog'
Press Enter when you're ready to start typing.
```



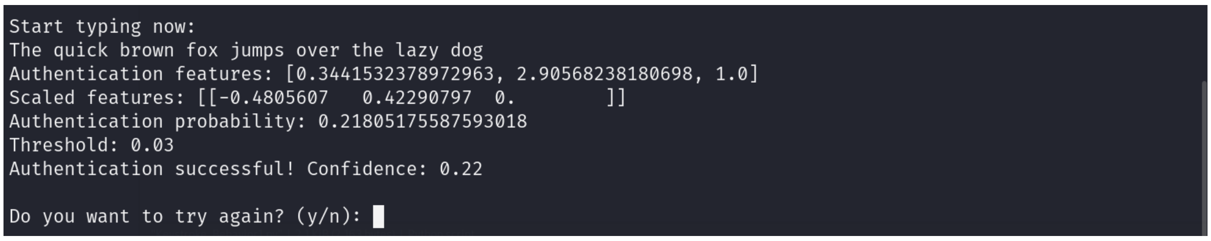
```
Start typing now:
The quick brown fox jumps over the lazy dog

Sample 6/7

Please type the following text:
'The quick brown fox jumps over the lazy dog'
Press Enter when you're ready to start typing.


Start typing now:
The quick brown fox jumps over the lazy dog

Sample 7/7

Please type the following text:
'The quick brown fox jumps over the lazy dog'
Press Enter when you're ready to start typing.


Start typing now:
The quick brown fox jumps over the lazy dog
```

- It then enters an authentication loop, where it repeatedly tests new typing samples against the trained model.

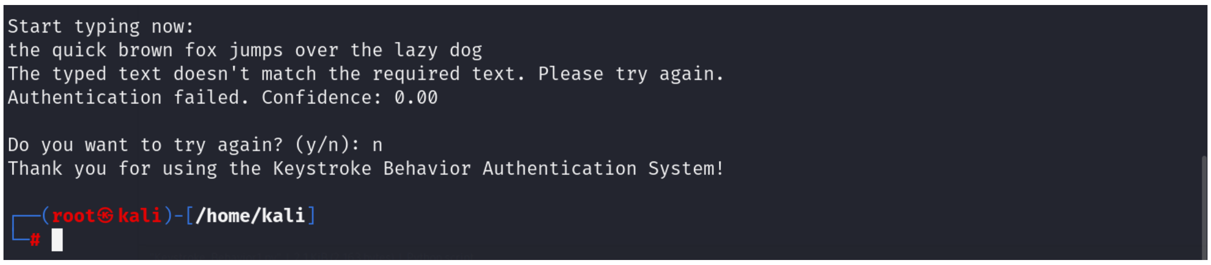


```
Authentication phase:

Please type the following text:
'The quick brown fox jumps over the lazy dog'
Press Enter when you're ready to start typing.
```

- The process continues until the user chooses to stop.



```
Start typing now:
The quick brown fox jumps over the lazy dog
Authentication features: [0.3441532378972963, 2.90568238180698, 1.0]
Scaled features: [[-0.4805607   0.42290797  0.        ]]
Authentication probability: 0.21805175587593018
Threshold: 0.03
Authentication successful! Confidence: 0.22

Do you want to try again? (y/n):
```



```
Start typing now:
the quick brown fox jumps over the lazy dog
The typed text doesn't match the required text. Please try again.
Authentication failed. Confidence: 0.00

Do you want to try again? (y/n): n
Thank you for using the Keystroke Behavior Authentication System!

┌──(root㉿kali)-[/home/kali]
└─#
```

If the authentication is too sensitive (Not authenticating the same user) reduce the threshold value and if it is less sensitive (authenticating any user) then increase the threshold value in the code.

The authentication probability is a measure of how confident the system is that the current typing pattern matches the patterns it was trained on. This probability ranges from 0 to 1, where 0 means the system is certain the typing pattern doesn't match, and 1 means it's certain the pattern does match. In our system, this probability is calculated by the machine learning model (the Multi-Layer Perceptron classifier) based on the features extracted from the user's typing pattern.

The threshold value, on the other hand, is the cutoff point we set to determine whether a user should be authenticated or not. If the authentication probability is equal to or higher than this threshold, the user is authenticated; if it's lower, the authentication fails.

Changing the threshold value directly affects the system's sensitivity and the likelihood of successful authentication. A lower threshold makes the system more lenient and increases the probability of authentication. For example, if we set the threshold to 0.1, it means we're willing to authenticate a user even if the system is only 10% confident that the typing pattern matches. This increases the chance of successful authentication but also increases the risk of false positives – authenticating users who shouldn't be granted access.

Conversely, a higher threshold makes the system more strict and decreases the probability of authentication. If we set the threshold to 0.9, for instance, the system would only authenticate users when it's 90% confident that the typing pattern matches. This reduces the risk of false positives but increases the chance of false negatives – rejecting legitimate users.

The choice of threshold involves a trade-off between security and usability. A very low threshold might make the system easy to use (as it's more likely to authenticate users) but less secure. A very high threshold might make the system more secure but potentially frustrating for legitimate users if their typing patterns vary slightly.

In practice, the optimal threshold depends on various factors, including the specific application, the desired level of security, and the consistency of users' typing patterns. It's often determined empirically by testing the system with both legitimate users and potential impostors, and finding a balance that minimizes both false positives and false negatives.

In our simplified system, we've set a very low threshold to prioritize usability, allowing you to experiment with the concept. In a real-world application, more sophisticated techniques would be used to determine and adjust this threshold, possibly even using adaptive thresholds that change based on factors like the user's typing consistency or the sensitivity of the protected resources.

The accuracy of this model can be improved drastically by increasing the amount of training data.