Assignment - 2

Phone data

This assignment utilizes the **Phone data** extracted from the Keyboard 51-60 dataset.

Step 1:- Data Aggregation and Preparation

```python
import os
import pandas as pd

# Path to the extracted folder
folder_path = "Keyboard51-60"

# Initialize a list to store dataframes
desktop_dataframes = []

# Loop through all subfolders (user folders)
for subfolder in os.listdir(folder_path):
    subfolder_path = os.path.join(folder_path, subfolder)
    if os.path.isdir(subfolder_path):  # Check if it is a folder
        userID = subfolder  # Extract userID from the folder name
        # Look for files containing "phone" in their name
        for file in os.listdir(subfolder_path):
            if "phone" in file.lower():
                file_path = os.path.join(subfolder_path, file)
                # Read the CSV file into a DataFrame
                df = pd.read_csv(file_path)
                # Add the userID column to the DataFrame
                df['userID'] = userID
                # Append the DataFrame to the list
                desktop_dataframes.append(df)

# Combine all desktop dataframes into a single dataframe
combined_desktop_data = pd.concat(desktop_dataframes, ignore_index=True)

# Save the combined dataframe to a new CSV file
output_file_path = "combined_phone_data.csv"
combined_desktop_data.to_csv(output_file_path, index=False)

print(f"Combined desktop data saved to: {output_file_path}")
```

- The code aggregates phone keystroke data from multiple users. It reads CSV files containing "phone" in their filenames from user-specific subfolders.

- It adds a userID column to each DataFrame based on the subfolder name.
- All DataFrames are combined into a single DataFrame for further analysis.
- The combined data is saved to combined_phone_data.csv.

Purpose - To prepare a consolidated dataset of phone keystroke events from multiple users.

Step 2:- Data Windowing and Overlap Creation

```python
import pandas as pd  # Import the pandas library for data manipulation

# Load the dataset from the CSV file into a pandas DataFrame
data = pd.read_csv('combined_phone_data.csv')

# Convert the 'timestamp' column to datetime objects for accurate time calculations
data['timestamp'] = pd.to_datetime(data['timestamp'])

# Sort the data by 'userID' and 'timestamp' to ensure chronological order within each user
data.sort_values(by=['userID', 'timestamp'], inplace=True)

# Define parameters for creating overlapping windows
window_size = 1000             # Each sample will consist of 1000 keystroke events
overlap = 0.75                 # 75% overlap between consecutive samples
step_size = int(window_size * (1 - overlap))  # Calculate step size based on overlap (250 events)

# Initialize lists to store the samples and their corresponding labels (user IDs)
samples = []
sample_labels = []

# Iterate over each unique user in the dataset
for user_id in data['userID'].unique():
    # Extract data for the current user
    user_data = data[data['userID'] == user_id].reset_index(drop=True)
    num_events = len(user_data)  # Total number of keystroke events for this user

    # Calculate the number of samples that can be created from this user's data
    num_samples = (num_events - window_size) // step_size + 1

    # Loop to create overlapping samples for the current user
    for i in range(num_samples):
        # Calculate the starting and ending indices for the current window
        start_index = i * step_size
        end_index = start_index + window_size

        # Extract the sample data for the current window
        sample = user_data.iloc[start_index:end_index]

        # Append the sample to the list of samples
        samples.append(sample)

        # Append the user ID as the label for this sample
        sample_labels.append(user_id)
```

- The code processes a consolidated dataset of phone keystroke events for technical preparation.
- Converts the 'timestamp' column into datetime objects to ensure accurate time-based calculations.
- Sorts the data by 'userID' and 'timestamp' to maintain chronological order within each user's events.
- Defines windowing parameters:
    - **Window size:** 1000 events per sample.
    - **Overlap:** 75% between consecutive samples.
    - **Step size:** 250 events for sample progression.
- Iterates through the dataset:
    - Extracts user-specific data for each unique userID.
    - Calculates the number of overlapping samples possible for each user's data.
    - Creates samples by slicing overlapping windows of events.
    - Appends each sample along with its userID to respective lists for storage.

Purpose:

- To generate labeled, overlapping keystroke event samples for each user, enabling:
    - Temporal consistency in data analysis.
    - Robust feature extraction for behavioral modeling.
    - Preparation of structured input for machine learning algorithms targeting user identification.

Step 3: - Outlier Detection and Hold Time Processing

```python
# Initialize a list to store the processed features for each sample
processed_samples = []
keys_of_interest = ['t', 'a', 'e', 'i', 'h']

# Function to calculate hold times for a given sample
def calculate_hold_times(sample):
    hold_times = {key: [] for key in keys_of_interest}  # Initialize hold times for each key
    key_down_times = {}  # Dictionary to store the timestamp of 'DOWN' events

    # Iterate over the events in the sample
    for _, event in sample.iterrows():
        key = event['keyPressed']
        state = event['keyState']
        timestamp = event['timestamp']

        if key in keys_of_interest:
            if state == 'DOWN':
                # Record the time when the key was pressed down
                key_down_times[key] = timestamp
            elif state == 'UP' and key in key_down_times:
                # Calculate hold time when the key is released
                hold_time = (timestamp - key_down_times[key]).total_seconds()
                hold_times[key].append(hold_time)
                # Remove the key from the down times dictionary
                del key_down_times[key]
    return hold_times

# Function to remove outliers using the IQR method
def remove_outliers(hold_times):
    cleaned_hold_times = {}
    for key, times in hold_times.items():
        times_array = np.array(times)
        if len(times_array) > 0:
            # Calculate Q1 and Q3
            Q1 = np.percentile(times_array, 25)
            Q3 = np.percentile(times_array, 75)
            IQR = Q3 - Q1
            # Define bounds for non-outliers
            lower_bound = Q1 - 1.5 * IQR
            upper_bound = Q3 + 1.5 * IQR
            # Filter out the outliers
            non_outliers = times_array[(times_array >= lower_bound) & (times_array <= upper_bound)]
            cleaned_hold_times[key] = non_outliers
        else:
            # If no times recorded for this key, assign an empty array
            cleaned_hold_times[key] = np.array([])
    return cleaned_hold_times

# Process each sample
for sample in samples:
    # Step 1: Calculate hold times for the sample
    hold_times = calculate_hold_times(sample)

    # Step 2: Remove outliers from hold times
    cleaned_hold_times = remove_outliers(hold_times)

    # Store the cleaned hold times for feature extraction in the next step
    processed_samples.append(cleaned_hold_times)
```

- Key Hold Time Calculation:
    - Computes the hold time for keys 't', 'a', 'e', 'i', and 'h' within each sample.
    - Measures the time difference between DOWN and UP events for each key.
- Outlier Detection:
    - Implements the Interquartile Range (IQR) method for outlier removal.
    - Calculates Q1, Q3, and IQR for hold times of each key.
    - Filters out hold times outside the range [Q1 - 1.5 * IQR, Q3 + 1.5 * IQR]

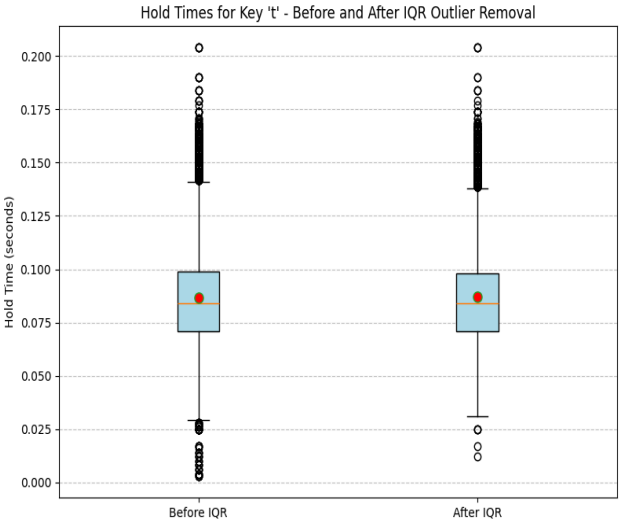
Purpose:

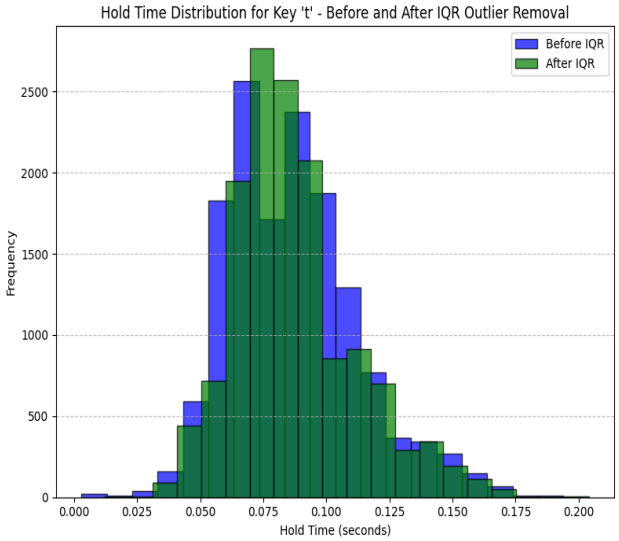
- Removes extreme and noisy data points to improve modeling accuracy.
- Prepares clean hold time data for statistical analysis (e.g., mean and standard deviation).

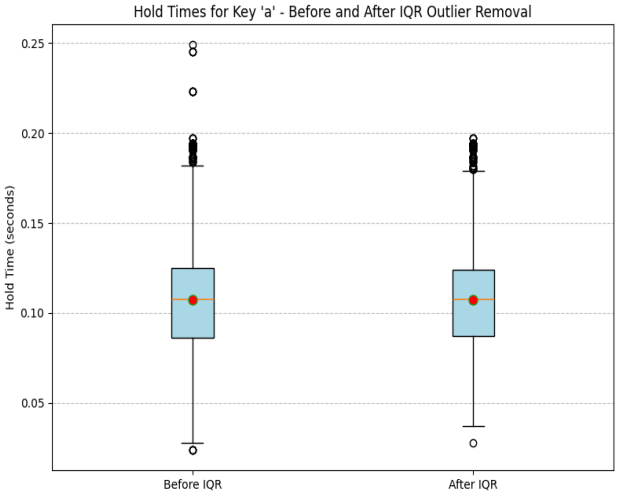
Visualization of Data Before and After Outlier Removal Using IQR

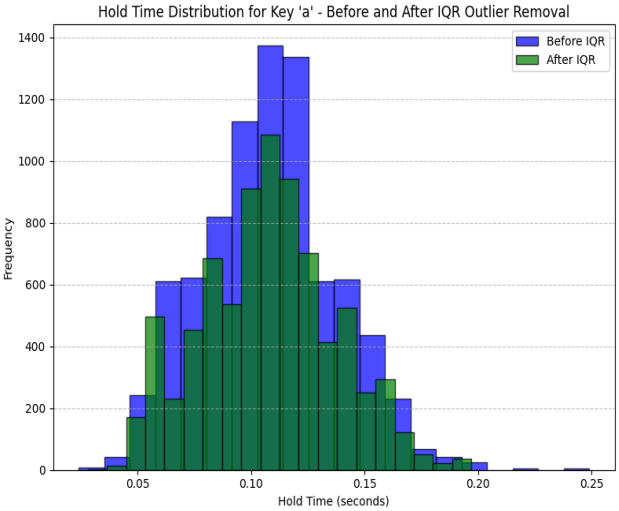
```
# Visualize hold times before and after outlier removal
for key in keys_of_interest:
    # Collect data for all samples before and after outlier removal
    hold_times_before = []
    hold_times_after = []

    for sample in samples:
        # Calculate hold times for the sample
        hold_times = calculate_hold_times(sample)
        hold_times_before.extend(hold_times[key])

    for cleaned_hold_times in processed_samples:
        hold_times_after.extend(cleaned_hold_times[key])

    # Convert to NumPy arrays for visualization
    hold_times_before = np.array(hold_times_before)
    hold_times_after = np.array(hold_times_after)

    # Create side-by-side box plots
    plt.figure(figsize=(10, 5))
    plt.boxplot(
        [hold_times_before, hold_times_after],
        labels=['Before IQR', 'After IQR'],
        showmeans=True,
        patch_artist=True,
        boxprops=dict(facecolor='lightblue'),
        meanprops=dict(marker='o', markerfacecolor='red', markersize=8),
    )
    plt.title(f"Hold Times for Key '{key}' - Before and After IQR Outlier Removal")
    plt.ylabel("Hold Time (seconds)")
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.show()

    # Create histograms to visualize the distribution
    plt.figure(figsize=(12, 6))
    plt.hist(hold_times_before, bins=20, alpha=0.7, label='Before IQR', color='blue', edgecolor='black')
    plt.hist(hold_times_after, bins=20, alpha=0.7, label='After IQR', color='green', edgecolor='black')
    plt.title(f"Hold Time Distribution for Key '{key}' - Before and After IQR Outlier Removal")
    plt.xlabel("Hold Time (seconds)")
    plt.ylabel("Frequency")
    plt.legend()
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.show()
```

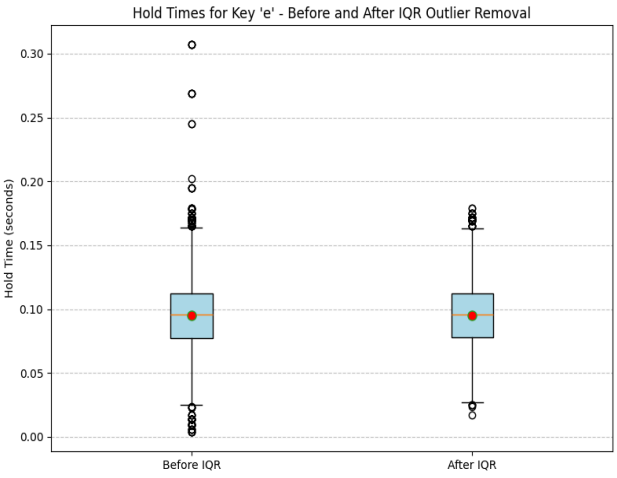
Results

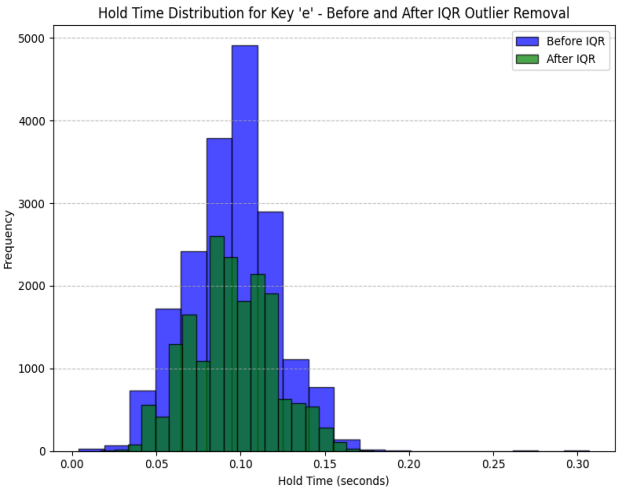

Hold Times for Key 't' - Before and After IQR Outlier Removal
Hold Time (seconds)
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
Before IQR
After IQR

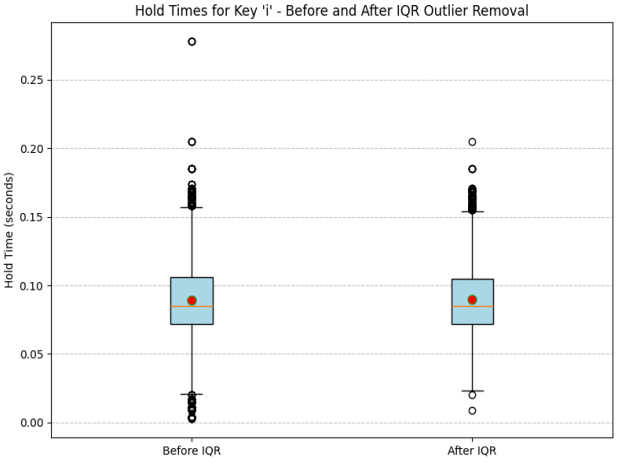

Hold Time Distribution for Key 't' - Before and After IQR Outlier Removal
Before IQR
After IQR
Frequency
2500
2000
1500
1000
500
0
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
Hold Time (seconds)

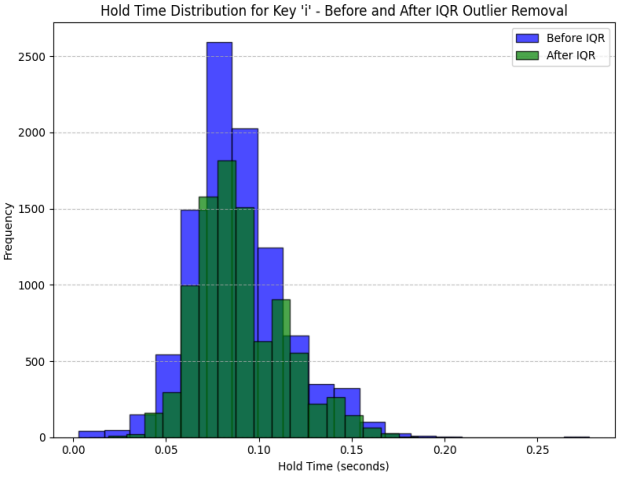

Hold Times for Key 'a' - Before and After IQR Outlier Removal
Hold Time (seconds)
0.25
0.20
0.15
0.10
0.05
Before IQR
After IQR

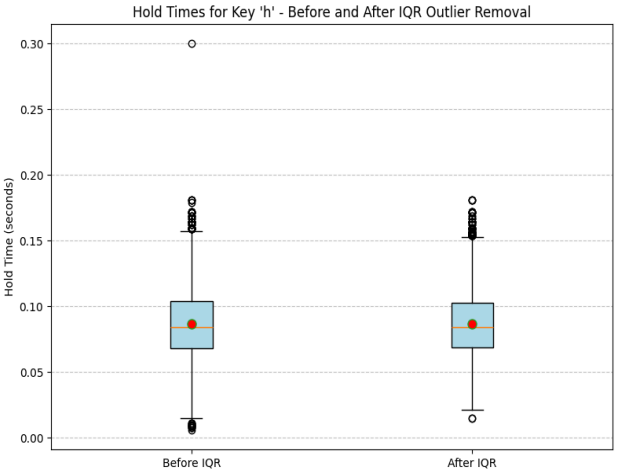

Hold Time Distribution for Key 'a' - Before and After IQR Outlier Removal
Before IQR
After IQR
Frequency
1400
1200
1000
800
600
400
200
0
0.05
0.10
0.15
0.20
0.25
Hold Time (seconds)

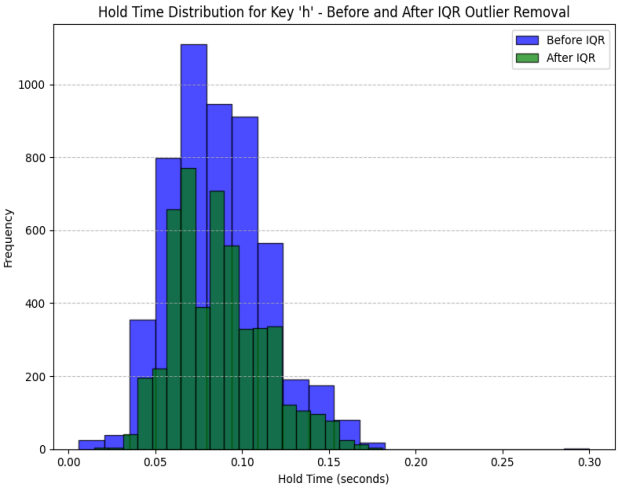

Hold Times for Key 'e' - Before and After IQR Outlier Removal
Hold Time (seconds)
0.00
0.05
0.10
0.15
0.20
0.25
0.30
Before IQR
After IQR


Hold Time Distribution for Key 'e' - Before and After IQR Outlier Removal
Before IQR
After IQR
Frequency
0
1000
2000
3000
4000
5000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
Hold Time (seconds)


Hold Times for Key 'i' - Before and After IQR Outlier Removal
Hold Time (seconds)
0.00
0.05
0.10
0.15
0.20
0.25
Before IQR
After IQR


Hold Time Distribution for Key 'i' - Before and After IQR Outlier Removal
Before IQR
After IQR
Frequency
0
500
1000
1500
2000
2500
0.00
0.05
0.10
0.15
0.20
0.25
Hold Time (seconds)


Hold Times for Key 'h' - Before and After IQR Outlier Removal
Hold Time (seconds)
0.00
0.05
0.10
0.15
0.20
0.25
0.30
Before IQR
After IQR


Hold Time Distribution for Key 'h' - Before and After IQR Outlier Removal
Before IQR
After IQR
Frequency
0
200
400
600
800
1000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
Hold Time (seconds)

Step 4:- Feature Extraction

```python
import numpy as np
import pandas as pd

# Initialize lists to store feature vectors and corresponding labels
feature_vectors = []
labels = []

# Keys of interest
keys_of_interest = ['t', 'a', 'e', 'i', 'h']

# Iterate over each processed sample and its corresponding label
for idx, cleaned_hold_times in enumerate(processed_samples):
    features = []
    for key in keys_of_interest:
        times = cleaned_hold_times.get(key, np.array([]))
        if len(times) > 0:
            # Calculate mean and standard deviation
            mean_time = np.mean(times)
            std_time = np.std(times)
        else:
            # Handle missing data by assigning zero
            mean_time = 0.0
            std_time = 0.0
        # Append the features for this key
        features.extend([mean_time, std_time])
    # Append the feature vector and label to the lists
    feature_vectors.append(features)
    labels.append(sample_labels[idx])

# Convert feature vectors and labels to NumPy arrays or pandas DataFrames
feature_vectors = np.array(feature_vectors)
labels = np.array(labels)

# Optionally, create a pandas DataFrame for better visualization
feature_names = []
for key in keys_of_interest:
    feature_names.extend([f'{key}_mean', f'{key}_std'])

features_df = pd.DataFrame(feature_vectors, columns=feature_names)
labels_series = pd.Series(labels, name='userID')
```

Feature list

	t_mean	t_std	a_mean	a_std	e_mean	e_std	i_mean	i_std	h_mean	h_std
0	0.069544	0.012623	0.108720	0.011411	0.099556	0.014590	0.082357	0.013880	0.083143	0.016754
1	0.067218	0.011296	0.109148	0.009164	0.098475	0.013154	0.083407	0.016802	0.078667	0.015047
2	0.066075	0.010749	0.105500	0.009795	0.096952	0.013340	0.083857	0.016329	0.079524	0.016014
3	0.067925	0.010944	0.104107	0.010903	0.097246	0.012695	0.088600	0.015645	0.080238	0.017238
4	0.071132	0.010347	0.102152	0.010779	0.096102	0.013257	0.088844	0.015545	0.080450	0.017078
...	...	...	...	...	...	...	...	...	...	...
359	0.119409	0.025980	0.139033	0.025930	0.113026	0.017580	0.105667	0.012136	0.120778	0.022282
360	0.115714	0.023879	0.130476	0.018115	0.106923	0.015259	0.115533	0.020896	0.120095	0.021152
361	0.110786	0.017889	0.116526	0.019712	0.102814	0.014458	0.111000	0.018251	0.119190	0.024655
362	0.112775	0.018109	0.117174	0.018948	0.102300	0.013759	0.112143	0.022681	0.116500	0.021487
363	0.107765	0.019566	0.113280	0.017294	0.107810	0.016706	0.114909	0.022617	0.122500	0.020637

364 rows × 10 columns

labels

```
0      51
1      51
2      51
3      51
4      51
       ..
359    60
360    60
361    60
362    60
363    60
Name: userID, Length: 364, dtype: int64
```

- Feature Vector Creation:
    - Extracts features (mean and standard deviation of hold times) for keys of interest: 't', 'a', 'e', 'i', 'h'.
    - Processes each cleaned sample to compute statistical features for the specified keys.
    - Handles missing data by assigning zeros for keys with no recorded hold times after outlier removal.

- Data Structuring:
    - Combines features for all keys into a unified feature vector for each sample.
    - Associates each feature vector with its corresponding user label (userID).
- Output Representation:
    - Converts feature vectors and labels into:
        - NumPy arrays: For efficient numerical operations.
        - Pandas DataFrame: Provides better visualization and inspection of feature names and corresponding values.

Purpose

- Transforms raw hold time data into structured statistical features.
- Extracted features provide unique behavioral patterns for each user

Step 5:- Model Training and Evaluation Using Stratified K-Fold Cross-Validation

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.regularizers import l2
import matplotlib.pyplot as plt
import tensorflow as tf

# Step 1: Data Preparation

# Feature Scaling using StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(feature_vectors)

# Encoding labels (user IDs) to integers using LabelEncoder
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(labels)

# Check class distribution for imbalance
unique, counts = np.unique(y_encoded, return_counts=True)
class_distribution = dict(zip(unique, counts))
print("Class Distribution:", class_distribution)

# Convert integer labels to one-hot encoded vectors
y_one_hot = to_categorical(y_encoded, num_classes=10)

# Step 2: Initialize Stratified K-Fold Cross-Validation
skf = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

# Initialize lists to store test accuracies and histories for each fold
test_accuracies = []
all_histories = []
```

```
fold_no = 1
for train_index, val_index in skf.split(X_scaled, y_encoded):
    print(f"\nFold {fold_no}")
    # Split the data into training and validation sets for the current fold
    X_train_fold, X_val_fold = X_scaled[train_index], X_scaled[val_index]
    y_train_fold, y_val_fold = y_one_hot[train_index], y_one_hot[val_index]

    # Step 3: Building the Model
    model = Sequential()
    # First hidden layer with Batch Normalization and L2 Regularization
    model.add(Dense(10, input_dim=X_scaled.shape[1], kernel_regularizer=l2(0.001)))
    model.add(BatchNormalization())
    model.add(tf.keras.layers.Activation('relu'))
    # Second hidden layer with Batch Normalization and L2 Regularization
    model.add(Dense(10, kernel_regularizer=l2(0.001)))
    model.add(BatchNormalization())
    model.add(tf.keras.layers.Activation('relu'))
    # Output layer
    model.add(Dense(10, activation='softmax'))

    # Step 4: Compiling the Model
    optimizer = Adam(learning_rate=0.001)
    model.compile(
        optimizer=optimizer,
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )

    # Step 5: Define EarlyStopping Callback
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=20,
        restore_best_weights=True,
        verbose=1
    )

    # Step 6: Training the Model with Early Stopping
    history = model.fit(
        X_train_fold,
        y_train_fold,
        epochs=200,
        batch_size=32,
        validation_data=(X_val_fold, y_val_fold),
        callbacks=[early_stopping],
        verbose=1
    )

    # Save the history
    all_histories.append(history)
```

```
    # Step 7: Evaluating the Model
    loss, accuracy = model.evaluate(X_val_fold, y_val_fold, verbose=0)
    print(f"Validation Accuracy for Fold {fold_no}: {accuracy * 100:.2f}%")

    # Append the accuracy to the list
    test_accuracies.append(accuracy)

    # Plot training & validation accuracy and loss values
    epochs_range = range(1, len(history.history['accuracy']) + 1)
    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)
    plt.plot(epochs_range, history.history['loss'], label='Training Loss')
    plt.plot(epochs_range, history.history['val_loss'], label='Validation Loss')
    plt.title(f'Fold {fold_no} - Training vs Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(epochs_range, history.history['accuracy'], label='Training Accuracy')
    plt.plot(epochs_range, history.history['val_accuracy'], label='Validation Accuracy')
    plt.title(f'Fold {fold_no} - Training vs Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.tight_layout()
    plt.show()

    fold_no += 1

# Step 8: Reporting Results
# Calculate the average accuracy across all folds
average_accuracy = np.mean(test_accuracies)
print(f'\nAverage Validation Accuracy over {skf.get_n_splits()} folds: {average_accuracy * 100:.2f}%')
```

Summary:

- Data Scaling and Encoding:
    - Scales the feature data using StandardScaler to normalize feature values.
    - Encodes user labels into integer format using LabelEncoder.
    - Converts integer labels into one-hot encoded vectors for multi-class classification.
- Stratified K-Fold Cross-Validation: Splits the dataset into 3 folds, ensuring balanced class distribution in each fold.
- Model Definition: Constructs a Sequential neural network with:
    - **Two fully connected hidden layers (10 neurons each) with Batch Normalization and L2 Regularization.**
    - **ReLU activation** for hidden layers to introduce non-linearity.
    - A **softmax output layer** with 10 neurons for multi-class classification.
- Model Compilation:
    - Uses the **Adam optimizer with a learning rate of 0.001**.

    - Configures the categorical cross-entropy loss function for multi-class classification tasks.
    - Tracks accuracy as a performance metric.
- Training and Early Stopping:
    - Trains the model for a maximum of **200 epochs** with a **batch size of 32.**
    - Implements an **EarlyStopping callback to monitor validation loss** and stop training when improvement ceases for 20 consecutive epochs, restoring the best weights.
- Validation and Visualization:
    - Evaluates the model on the validation set for each fold and records the accuracy.
- Result Aggregation: Calculates and prints the average validation accuracy across all folds.

Results

Model architecture

Model: "sequential_21"

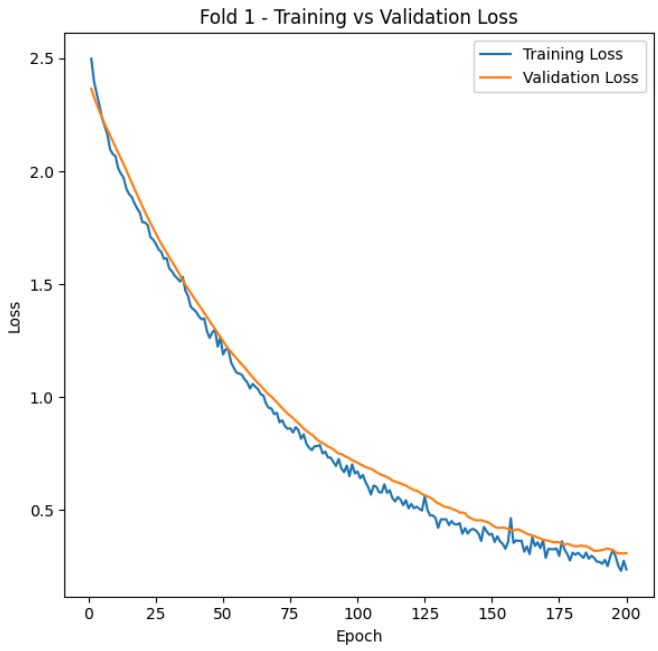
Layer (type)	Output Shape	Param #
dense_63 (Dense)	(None, 10)	110
batch_normalization_42 (BatchNormalization)	(None, 10)	40
activation_42 (Activation)	(None, 10)	0
dense_64 (Dense)	(None, 10)	110
batch_normalization_43 (BatchNormalization)	(None, 10)	40
activation_43 (Activation)	(None, 10)	0
dense_65 (Dense)	(None, 10)	110

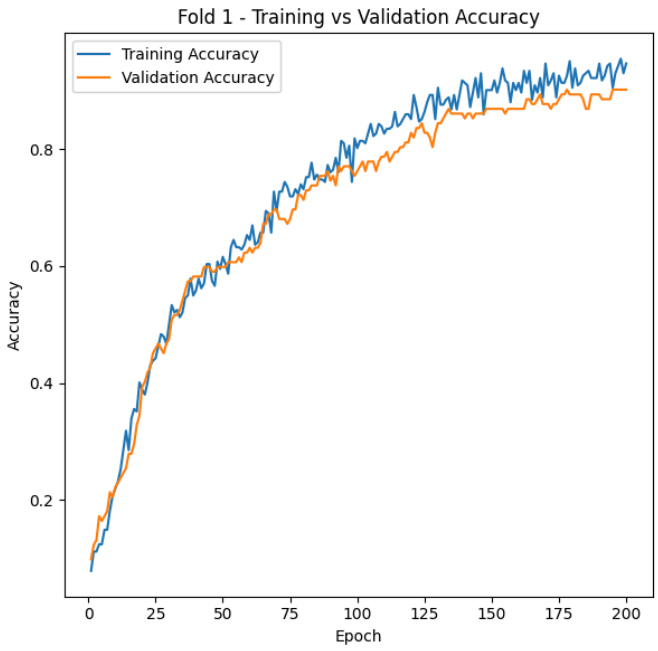
Total params: 410 (1.60 KB)

Trainable params: 370 (1.45 KB)

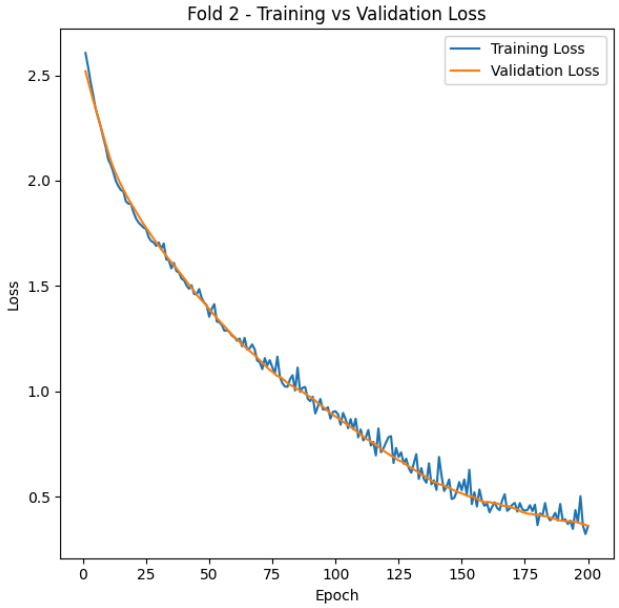
Non-trainable params: 40 (160.00 B)

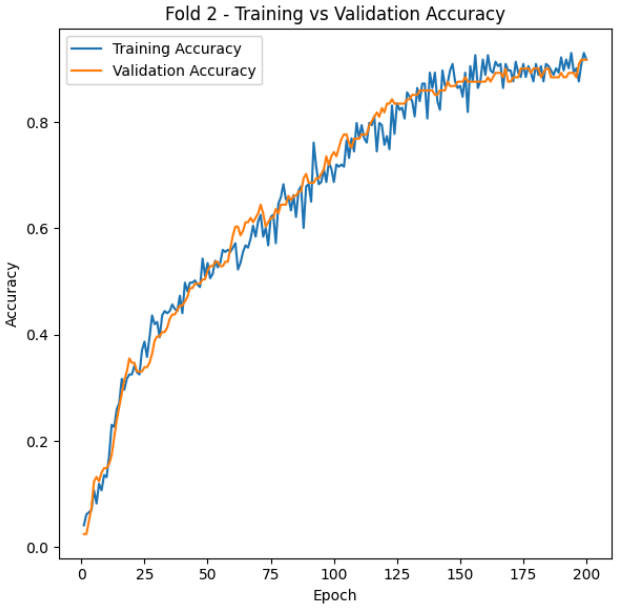
```
Fold 1
Epoch 1/200
/Users/rashmi/Library/Python/3.9/lib/python/site-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argum
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
8/8 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.0852 - loss: 2.4808 - val_accuracy: 0.0984 - val_loss: 2.3657
Epoch 2/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1138 - loss: 2.4116 - val_accuracy: 0.1230 - val_loss: 2.3290
Epoch 3/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1224 - loss: 2.3415 - val_accuracy: 0.1311 - val_loss: 2.2965
Epoch 4/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1248 - loss: 2.3117 - val_accuracy: 0.1721 - val_loss: 2.2664
Epoch 5/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1023 - loss: 2.2451 - val_accuracy: 0.1639 - val_loss: 2.2371
Epoch 6/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1272 - loss: 2.2371 - val_accuracy: 0.1721 - val_loss: 2.2095
Epoch 7/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1521 - loss: 2.1598 - val_accuracy: 0.1803 - val_loss: 2.1835
Epoch 8/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1735 - loss: 2.1039 - val_accuracy: 0.2131 - val_loss: 2.1575
Epoch 9/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2318 - loss: 2.0769 - val_accuracy: 0.2049 - val_loss: 2.1326
Epoch 10/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1972 - loss: 2.1210 - val_accuracy: 0.2213 - val_loss: 2.1077
Epoch 11/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2410 - loss: 2.0306 - val_accuracy: 0.2295 - val_loss: 2.0826
Epoch 12/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2734 - loss: 1.9639 - val_accuracy: 0.2377 - val_loss: 2.0570
Epoch 13/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2702 - loss: 1.9647 - val_accuracy: 0.2459 - val_loss: 2.0310
...
Epoch 200/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9478 - loss: 0.2441 - val_accuracy: 0.9016 - val_loss: 0.3099
Restoring model weights from the end of the best epoch: 199.
Validation Accuracy for Fold 1: 90.16%
```



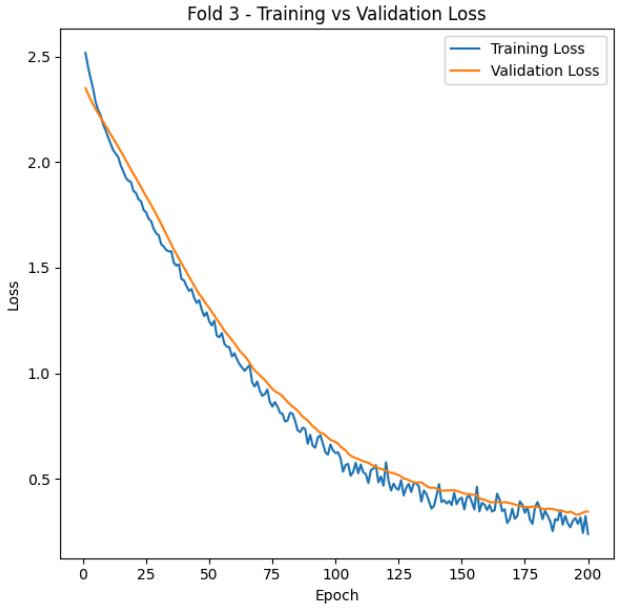


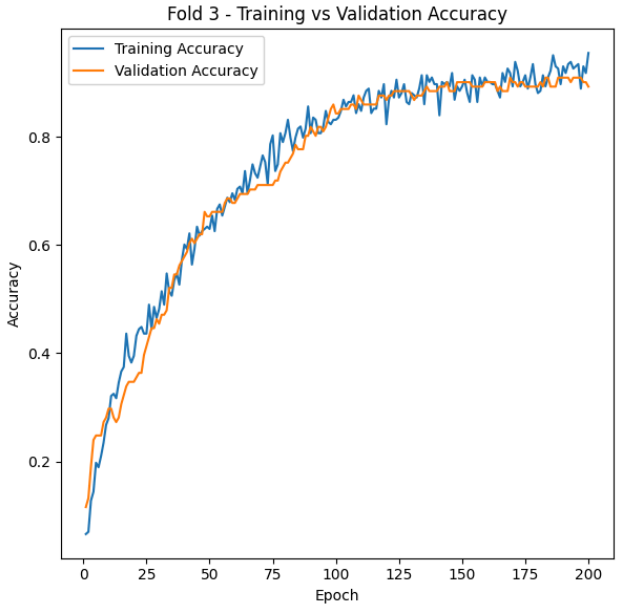
```
Fold 2
Epoch 1/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.0373 - loss: 2.5859 - val_accuracy: 0.0248 - val_loss: 2.5195
Epoch 2/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0575 - loss: 2.5425 - val_accuracy: 0.0248 - val_loss: 2.4738
Epoch 3/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0579 - loss: 2.4960 - val_accuracy: 0.0496 - val_loss: 2.4281
Epoch 4/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0878 - loss: 2.3893 - val_accuracy: 0.0744 - val_loss: 2.3830
Epoch 5/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1181 - loss: 2.3396 - val_accuracy: 0.1240 - val_loss: 2.3415
Epoch 6/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0753 - loss: 2.3262 - val_accuracy: 0.1322 - val_loss: 2.2995
Epoch 7/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1485 - loss: 2.2324 - val_accuracy: 0.1240 - val_loss: 2.2573
Epoch 8/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1098 - loss: 2.2045 - val_accuracy: 0.1405 - val_loss: 2.2150
Epoch 9/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1394 - loss: 2.1579 - val_accuracy: 0.1488 - val_loss: 2.1734
Epoch 10/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1362 - loss: 2.1233 - val_accuracy: 0.1488 - val_loss: 2.1338
Epoch 11/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1543 - loss: 2.0977 - val_accuracy: 0.1570 - val_loss: 2.0957
Epoch 12/200
...
Epoch 200/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9391 - loss: 0.3431 - val_accuracy: 0.9174 - val_loss: 0.3612
Restoring model weights from the end of the best epoch: 200.
Validation Accuracy for Fold 2: 91.74%
```





```
Fold 3
Epoch 1/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - accuracy: 0.0535 - loss: 2.5264 - val_accuracy: 0.1157 - val_loss: 2.3493
Epoch 2/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0604 - loss: 2.4591 - val_accuracy: 0.1322 - val_loss: 2.3231
Epoch 3/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1198 - loss: 2.4000 - val_accuracy: 0.1901 - val_loss: 2.2969
Epoch 4/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1431 - loss: 2.3535 - val_accuracy: 0.2397 - val_loss: 2.2727
Epoch 5/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1991 - loss: 2.3080 - val_accuracy: 0.2479 - val_loss: 2.2507
Epoch 6/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2023 - loss: 2.2312 - val_accuracy: 0.2479 - val_loss: 2.2309
Epoch 7/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2227 - loss: 2.2195 - val_accuracy: 0.2479 - val_loss: 2.2117
Epoch 8/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2082 - loss: 2.1745 - val_accuracy: 0.2727 - val_loss: 2.1912
Epoch 9/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2767 - loss: 2.1536 - val_accuracy: 0.2810 - val_loss: 2.1710
Epoch 10/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2646 - loss: 2.1577 - val_accuracy: 0.2975 - val_loss: 2.1517
Epoch 11/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.3206 - loss: 2.0780 - val_accuracy: 0.2975 - val_loss: 2.1321
Epoch 12/200
...
Epoch 200/200
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9463 - loss: 0.2424 - val_accuracy: 0.8926 - val_loss: 0.3456
Restoring model weights from the end of the best epoch: 196.
Validation Accuracy for Fold 3: 90.91%
```






```
Average Validation Accuracy over 3 folds: 90.94%
```

Results

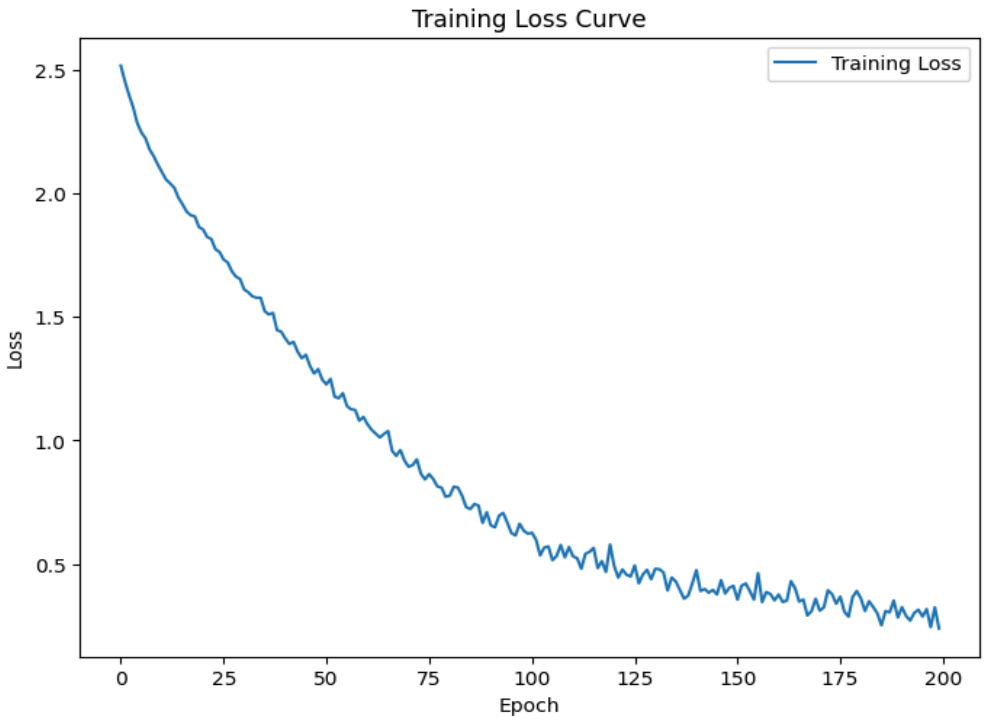
1. Class Distribution:
    - The dataset has 10 user IDs with different numbers of samples.
    - The distribution is slightly imbalanced but manageable due to the stratified cross-validation approach.
2. Validation Accuracy Across Folds:
    - **Fold 1: 90.16%**
    - **Fold 2: 91.74%**
    - **Fold 3: 90.91%**
    - Consistent accuracy across all folds shows that the model generalizes well.
3. Average Validation Accuracy:
    - The model achieved an **average validation accuracy of 90.94%**, indicating strong performance in identifying users based on keystroke patterns.
4. Training Trends:
    - The model steadily improves accuracy in the early epochs.
    - Validation loss and accuracy stabilize as the model learns meaningful patterns, showing effective training without overfitting.
5. Generalization:
    - Early stopping restored the best model weights, preventing overfitting.
    - Validation and training accuracy trends align, indicating good generalization across unseen data.

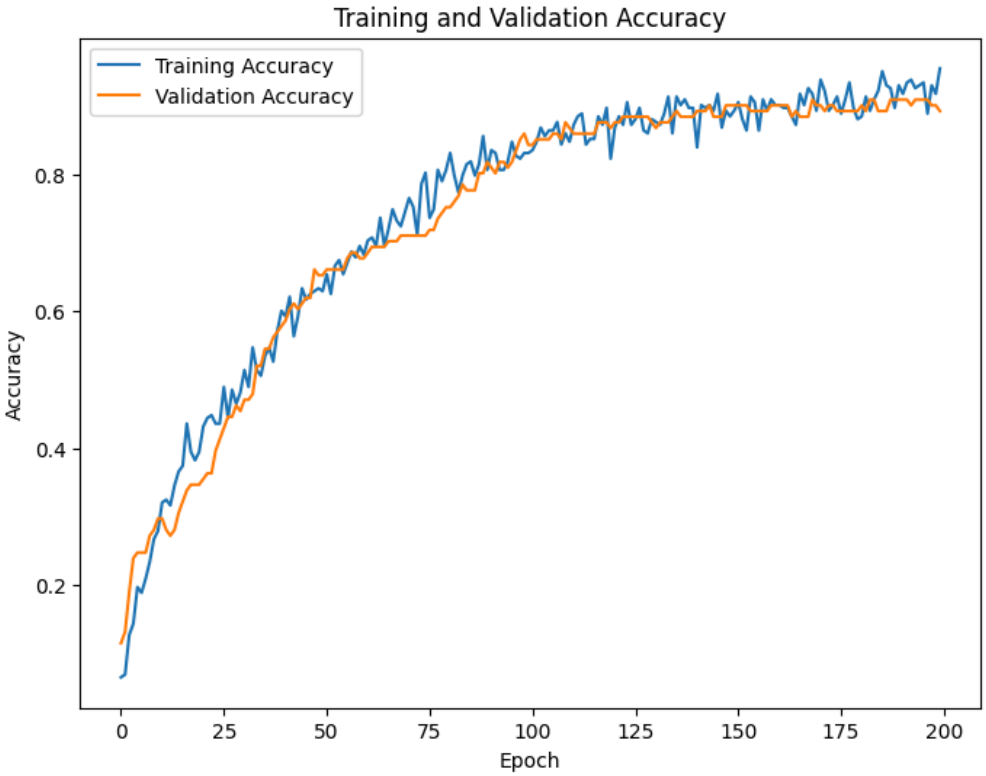
Step 6:- Visualisations

```
import matplotlib.pyplot as plt

# Plot the training loss curve w.r.t epoch
plt.figure(figsize=(8, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.title('Training Loss Curve')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plot the training and validation accuracy w.r.t epoch
plt.figure(figsize=(8, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Summary:

- Training Loss Curve:
    - This snippet visualizes the decrease in the training loss across epochs, showcasing how the model learns to minimize the error during training.
    - The loss curve provides insights into the efficiency of the optimization process.
- Training and Validation Accuracy:
    - This part of the code plots both training and validation accuracy curves across epochs.
    - It helps to monitor the model's ability to generalize unseen data (validation set) while simultaneously tracking the improvement in the training set.


Training Loss Curve
Training Loss
Loss
Epoch
2.5
2.0
1.5
1.0
0.5
0
25
50
75
100
125
150
175
200


Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
Epoch
0.8
0.6
0.4
0.2
0
25
50
75
100
125
150
175
200

Results

1. Training Loss Curve:
    - The training loss decreases steadily over the epochs, showing a smooth learning process.
    - The decline indicates the model effectively minimizes the error on the training data.
2. Training and Validation Accuracy:
    - Both training and validation accuracy improve significantly over time, demonstrating that the model is learning meaningful patterns in the data.
    - The alignment between the training and validation accuracy curves indicates minimal overfitting.
    - The final accuracy for both training and validation converges near the maximum, confirming the model's stability and generalization capability.

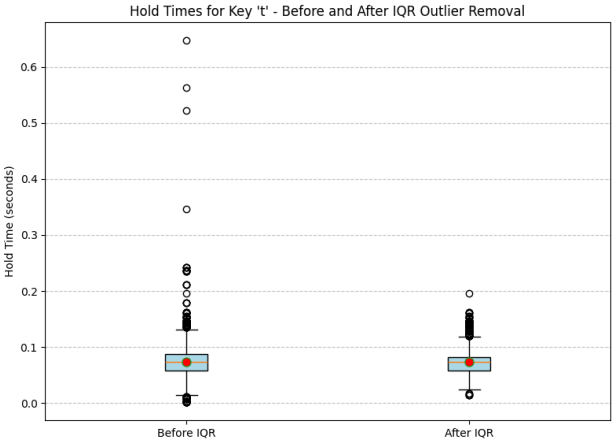
Tablet data

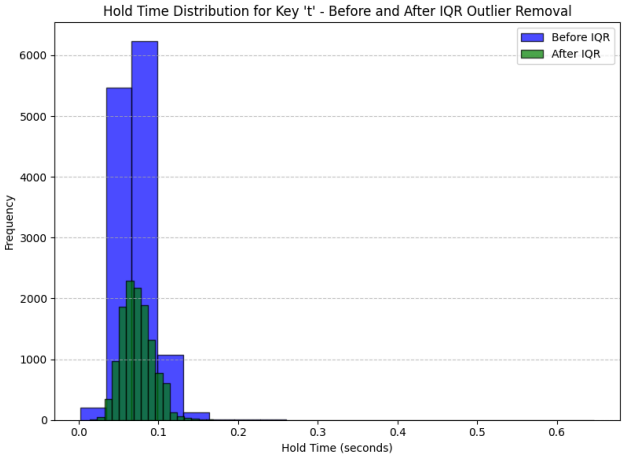
A similar approach and model were applied to the **tablet data** from the Keyboard51-60 dataset.

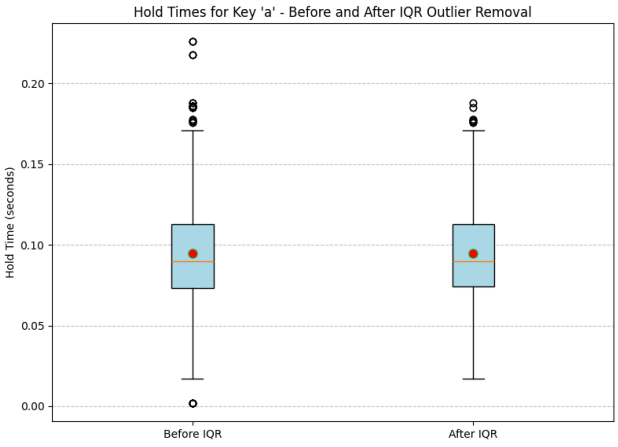
The steps include data preprocessing, feature extraction, outlier removal using the IQR method, and model training with stratified cross-validation.

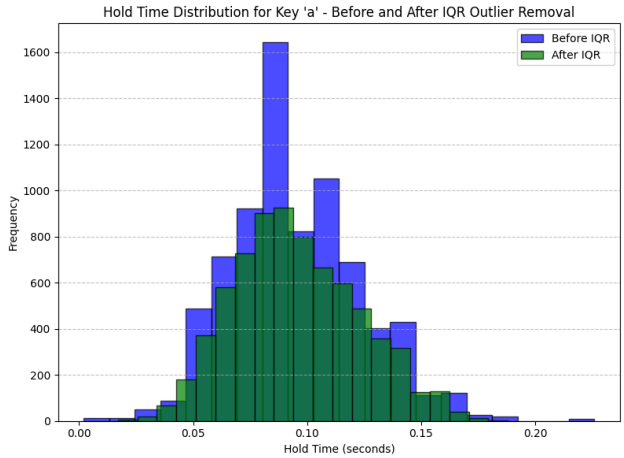
Below are the detailed steps, results, and insights:

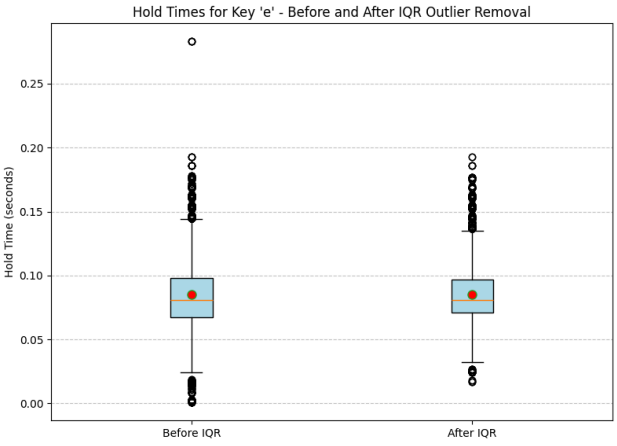
Comparison of key features before and after applying the outlier removal method.

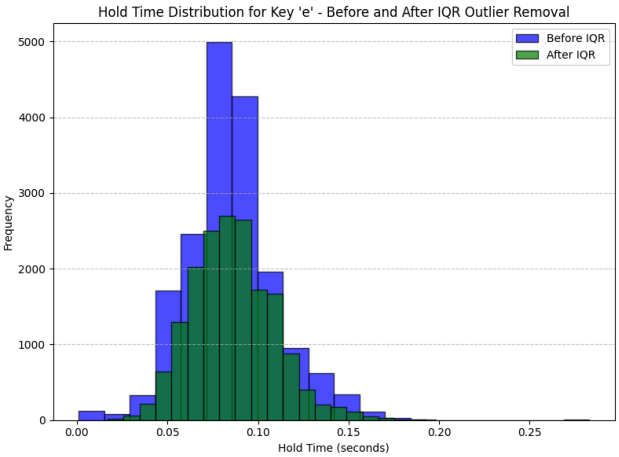


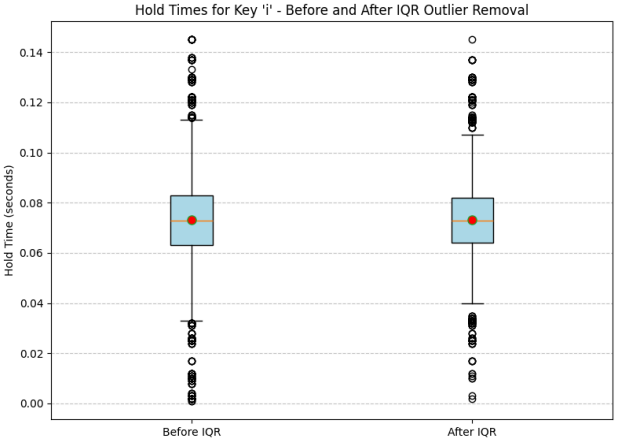


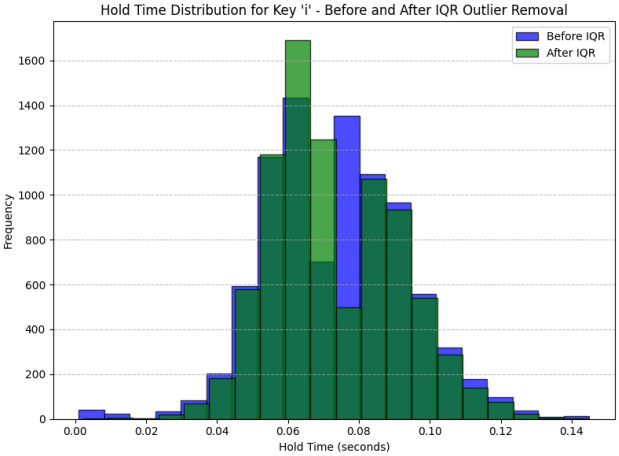


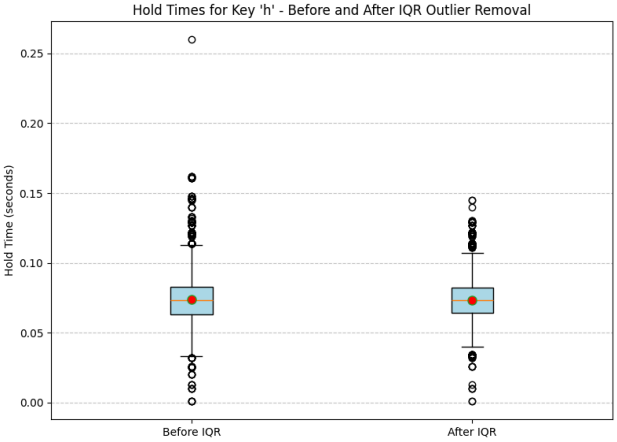


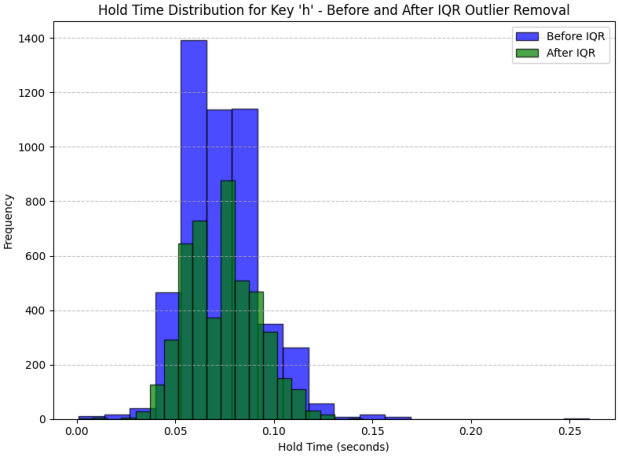

Hold Times for Key 'e' - Before and After IQR Outlier Removal
Hold Time (seconds)
Before IQR
After IQR


Hold Time Distribution for Key 'e' - Before and After IQR Outlier Removal
Before IQR
After IQR
Frequency
Hold Time (seconds)


Hold Times for Key 'i' - Before and After IQR Outlier Removal
Hold Time (seconds)
Before IQR
After IQR


Hold Time Distribution for Key 'i' - Before and After IQR Outlier Removal
Before IQR
After IQR
Frequency
Hold Time (seconds)


Hold Times for Key 'h' - Before and After IQR Outlier Removal
Hold Time (seconds)
Before IQR
After IQR


Hold Time Distribution for Key 'h' - Before and After IQR Outlier Removal
Before IQR
After IQR
Frequency
Hold Time (seconds)

K-fold model training with accuracy and loss visualizations for each iteration.

Model: "sequential_12"

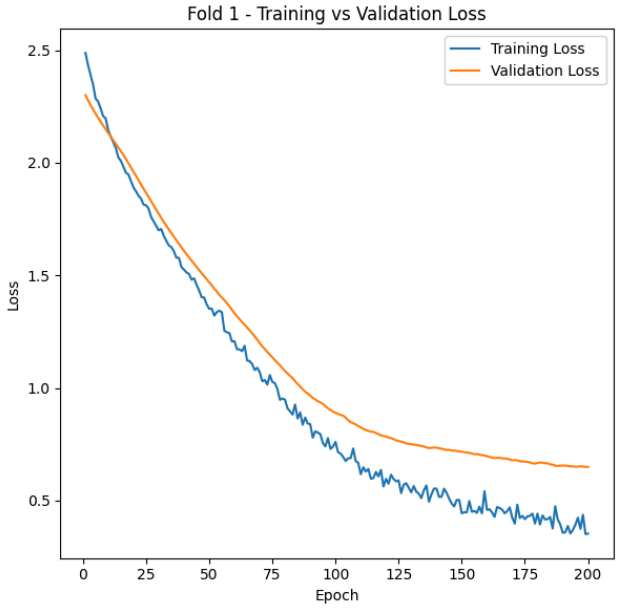
Layer (type)	Output Shape	Param #
dense_36 (Dense)	(None, 10)	110
batch_normalization_24 (BatchNormalization)	(None, 10)	40
activation_24 (Activation)	(None, 10)	0
dense_37 (Dense)	(None, 10)	110
batch_normalization_25 (BatchNormalization)	(None, 10)	40
activation_25 (Activation)	(None, 10)	0
dense_38 (Dense)	(None, 10)	110

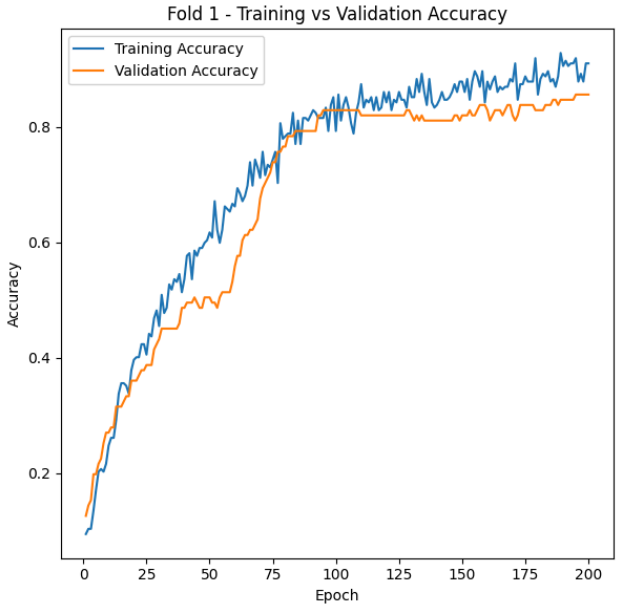
Total params: 410 (1.60 KB)

Trainable params: 370 (1.45 KB)

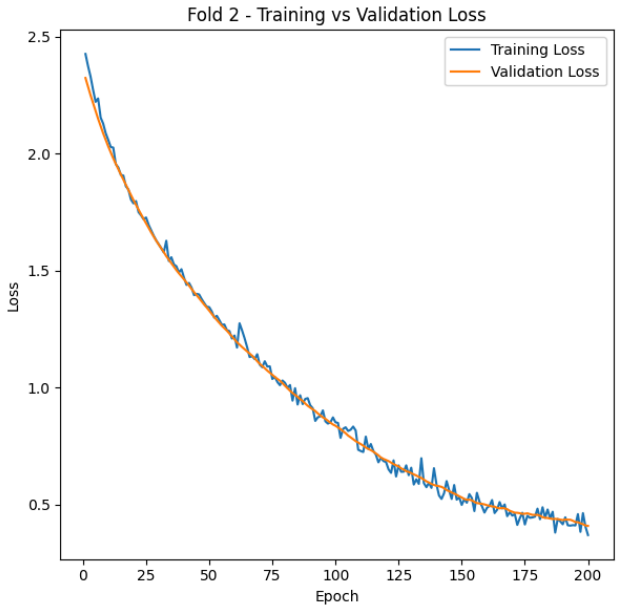
Non-trainable params: 40 (160.00 B)

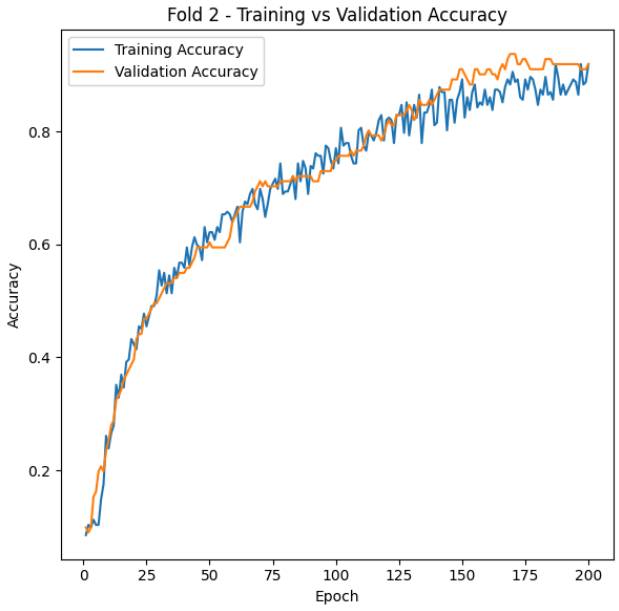
```
Fold 1
Epoch 1/200
/Users/rashmi/Library/Python/3.9/lib/python/site-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequenti
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
7/7 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - accuracy: 0.0988 - loss: 2.5001 - val_accuracy: 0.1261 - val_loss: 2.3011
Epoch 2/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0979 - loss: 2.4618 - val_accuracy: 0.1441 - val_loss: 2.2780
Epoch 3/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1067 - loss: 2.4084 - val_accuracy: 0.1532 - val_loss: 2.2568
Epoch 4/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1421 - loss: 2.3750 - val_accuracy: 0.1982 - val_loss: 2.2371
Epoch 5/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1742 - loss: 2.3090 - val_accuracy: 0.1982 - val_loss: 2.2181
Epoch 6/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1905 - loss: 2.2742 - val_accuracy: 0.2162 - val_loss: 2.2001
Epoch 7/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1822 - loss: 2.2606 - val_accuracy: 0.2252 - val_loss: 2.1828
Epoch 8/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2156 - loss: 2.1733 - val_accuracy: 0.2523 - val_loss: 2.1654
Epoch 9/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.1868 - loss: 2.2280 - val_accuracy: 0.2703 - val_loss: 2.1489
Epoch 10/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2585 - loss: 2.1714 - val_accuracy: 0.2703 - val_loss: 2.1327
Epoch 11/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2702 - loss: 2.0959 - val_accuracy: 0.2793 - val_loss: 2.1169
Epoch 12/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2485 - loss: 2.0730 - val_accuracy: 0.2793 - val_loss: 2.1014
Epoch 13/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2792 - loss: 2.0939 - val_accuracy: 0.3153 - val_loss: 2.0852
...
Epoch 200/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9264 - loss: 0.3519 - val_accuracy: 0.8559 - val_loss: 0.6492
Restoring model weights from the end of the best epoch: 199.
Validation Accuracy for Fold 1: 85.59%
```





```
Fold 2
Epoch 1/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step - accuracy: 0.0721 - loss: 2.4467 - val_accuracy: 0.0991 - val_loss: 2.3243
Epoch 2/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0737 - loss: 2.4176 - val_accuracy: 0.0901 - val_loss: 2.2877
Epoch 3/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0935 - loss: 2.3200 - val_accuracy: 0.0991 - val_loss: 2.2507
Epoch 4/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1085 - loss: 2.2774 - val_accuracy: 0.1532 - val_loss: 2.2165
Epoch 5/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0899 - loss: 2.2317 - val_accuracy: 0.1622 - val_loss: 2.1818
Epoch 6/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1159 - loss: 2.2915 - val_accuracy: 0.1982 - val_loss: 2.1496
Epoch 7/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1301 - loss: 2.2087 - val_accuracy: 0.2072 - val_loss: 2.1186
Epoch 8/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.1575 - loss: 2.1190 - val_accuracy: 0.1982 - val_loss: 2.0878
Epoch 9/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2481 - loss: 2.0867 - val_accuracy: 0.2342 - val_loss: 2.0594
Epoch 10/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2340 - loss: 2.0600 - val_accuracy: 0.2523 - val_loss: 2.0319
Epoch 11/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.2411 - loss: 2.0684 - val_accuracy: 0.2793 - val_loss: 2.0059
Epoch 12/200
...
Epoch 200/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9111 - loss: 0.3421 - val_accuracy: 0.9189 - val_loss: 0.4092
Restoring model weights from the end of the best epoch: 200.
Validation Accuracy for Fold 2: 91.89%
```





```
Fold 3
Epoch 1/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - accuracy: 0.0508 - loss: 2.6277 - val_accuracy: 0.0721 - val_loss: 2.4935
Epoch 2/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.0815 - loss: 2.5073 - val_accuracy: 0.1171 - val_loss: 2.4689
Epoch 3/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.1316 - loss: 2.4376 - val_accuracy: 0.1261 - val_loss: 2.4430
Epoch 4/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.0915 - loss: 2.5027 - val_accuracy: 0.1441 - val_loss: 2.4150
Epoch 5/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.1344 - loss: 2.3761 - val_accuracy: 0.1441 - val_loss: 2.3877
Epoch 6/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.1733 - loss: 2.3483 - val_accuracy: 0.1712 - val_loss: 2.3599
Epoch 7/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.1847 - loss: 2.2918 - val_accuracy: 0.1622 - val_loss: 2.3312
Epoch 8/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.2196 - loss: 2.2767 - val_accuracy: 0.1892 - val_loss: 2.3026
Epoch 9/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2684 - loss: 2.2061 - val_accuracy: 0.1802 - val_loss: 2.2731
Epoch 10/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2669 - loss: 2.1970 - val_accuracy: 0.1892 - val_loss: 2.2426
Epoch 11/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.2883 - loss: 2.1127 - val_accuracy: 0.1982 - val_loss: 2.2124
Epoch 12/200
...
Epoch 200/200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9199 - loss: 0.3474 - val_accuracy: 0.8739 - val_loss: 0.4518
Restoring model weights from the end of the best epoch: 200.
Validation Accuracy for Fold 3: 87.39%
```


```
Average Validation Accuracy over 3 folds: 88.29%
```

Validation Accuracy Across Folds:

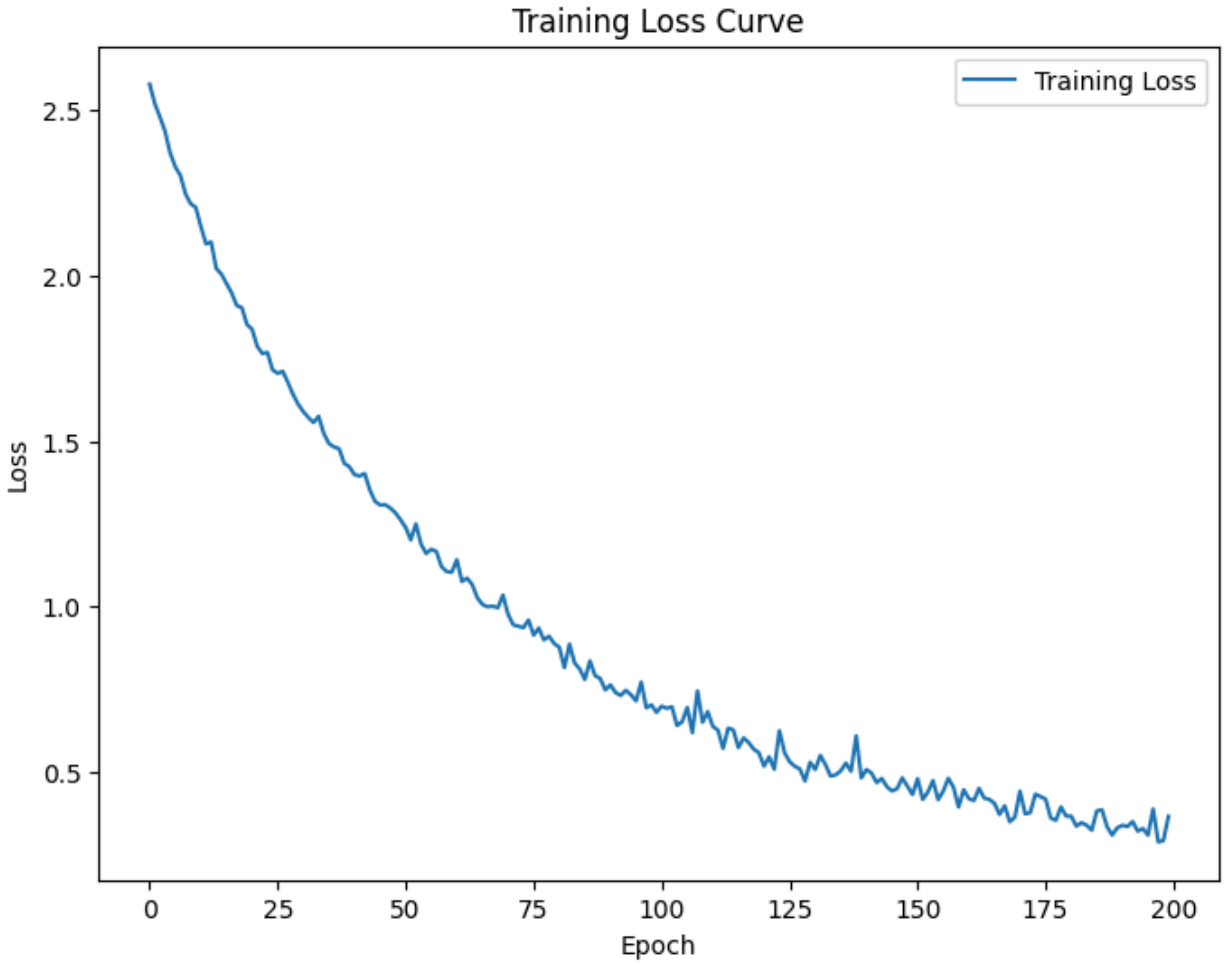
- **Fold 1: 87.39%**

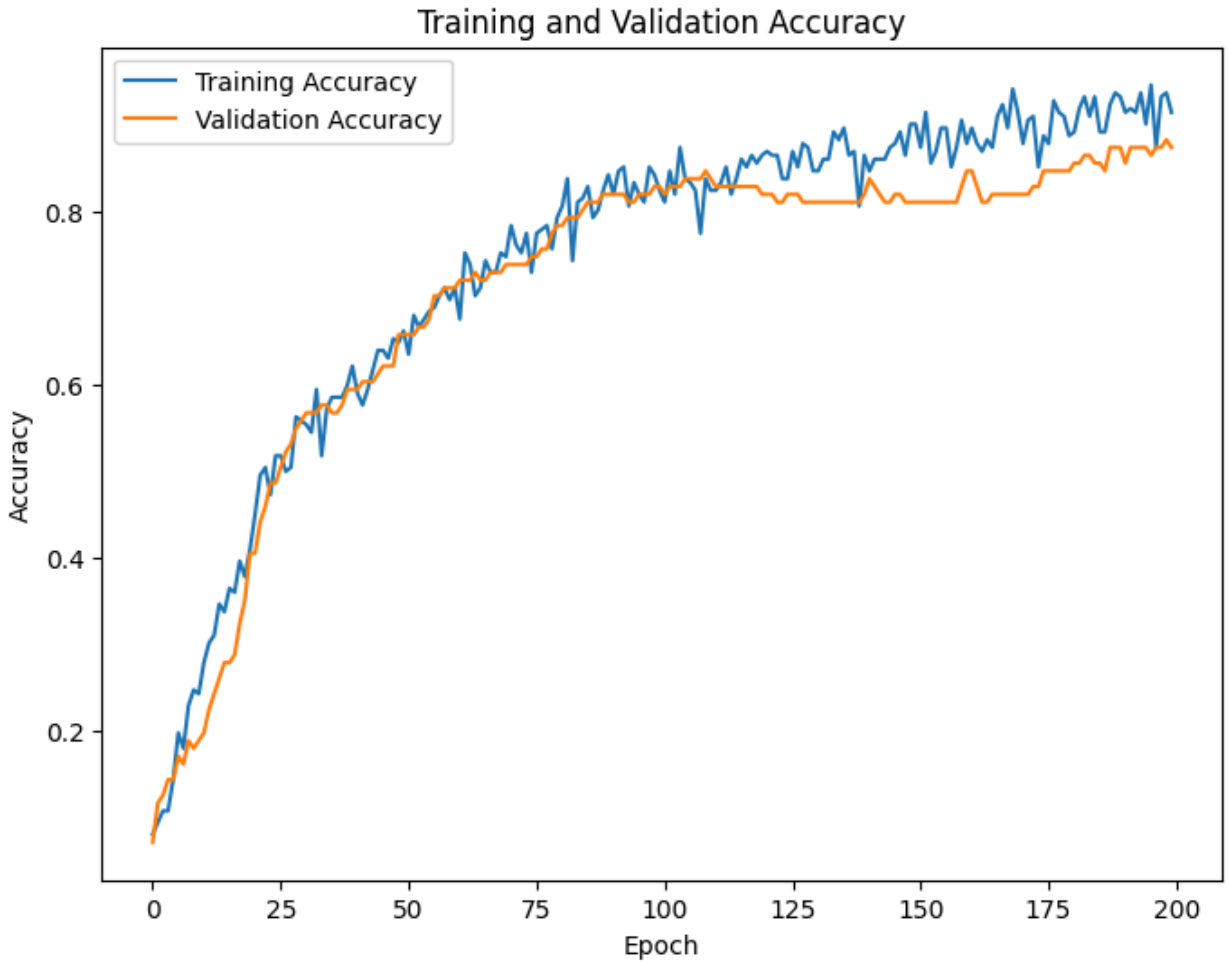
- **Fold 2: 88.29%**
- **Fold 3: 87.39%**

Average Validation Accuracy:

- The model achieved an **average validation accuracy of 87.69%**, indicating strong performance in identifying users based on keystroke patterns.

Visualizations of training and validation accuracy along with loss metrics




Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
0.2
0.4
0.6
0.8
0
25
50
75
100
125
150
175
200
Epoch