POZNAN UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTING AND TELECOMMUNICATION
Institute of Computing Science

Bachelor's thesis

BIOMETRIC IDENTIFICATION OF A SMARTPHONE USER USING GRAPH NEURAL NETWORKS

Jakub Grabowski, 151825
Filip Kozłowski, 151823
Krzysztof Matyla, 151778
Igor Warszawski, 151585

Supervisor
dr hab. inż. Szymon Szczęsny, prof. PP

POZNAŃ 2025

Tutaj będzie karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Contents

Chapter 1

Introduction

Biometric data is a widely used – especially on mobile devices – for user authentication. It is also used for person recognition. As of 2020, the majority of smartphones had biometric sensors, such as fingerprint readers [1]. Many computers can also provide biometric authentication via face recognition, if connected to a webcam, e.g. via Windows Hello on Windows 10 or 11 [2]. These are, however, not the only possible recognition or authentication methods that use biometric data.

The project aimed to develop a model, along with a corresponding mobile app, capable of recognizing users based on their biometric data, primarily derived from keystroke patterns. Participants in the study, conducted as a part of the project, provided their data by entering long stretches of text as testing data. Models were created for each user, with the standard model testing procedures and validations. A subgroup of the study participants was also asked to verify the model in real-life testing by writing short paragraphs in the application, which were sent to the server for user verification.

The scope of the work was to create a mobile application capable of gathering the keystroke data, which could then be used by the server to create Graph Neural Network (GNN) models tasked with recognizing the user as opposed to other possible users. Also in the scope was performing a study on a group of participants who provided the data for the project and participated in the application and model demonstration and testing.

The sources referenced in this thesis primarily fall into two categories: studies on keystroke data models and their effectiveness, and specialist literature concerning Graph Neural Networks.

The thesis has the following structure:

- Chapter 2 consists of some theory concerning biometrics, especially in the context of user input data, with a small literature review about using biometrics for user recognition.
- Chapter 3 contains basic theoretics about Graph Convolutional Networks, which are used for user recognition in the model created for the project.
- Chapter 4 is an overview of the project, explaining its components and the relationships between them. It contains subchapters about project use cases, server architecture and mobile application architecture.
- Chapter 5 is concerned with the Neural Network model, its design and feature engineering.
- Chapter 6 contains results of the study conducted on the users, with subchapter dedicated to discussing the findings.
- Chapter 7 is a brief conclusion to the thesis.

The division of labor for this project was as follows:

- Jakub Grabowski created the mobile application, set up and coordinated the project, and researched biometrics for the thesis paper. He wrote chapters 1, 2, 7 and parts of chapters 3 and 6.
- Filip Kozłowski created the server and integrated the GNN model with it. He also planned and implemented communication between the server and the application. He wrote parts of chapters 3, 4 and 6.
- Krzysztof Matyla helped in creating the mobile application interface, provided testing for various parts of the project, and coordinated user testing. He wrote most of chapter 4.
- Igor Warszawski planned and implemented the GNN model used on the server. He also tested and validated the results, together with Filip Kozłowski. He wrote chapter 5 and parts of chapters 3 and 6.

Chapter 2

Biometrics in mobile devices: state of the art

Fundamental to the goal of the project was the use of biometric data in user identification. Biometric data can be defined as measurements of some unique characteristics of an individual. These can largely be divided into two main categories: physiological data, which is the measurement of the inherent characteristics of an individual's body, such as a fingerprint, an iris scan or a face scan, and behavioral data, which measures the person's movements, behaviors, speech patterns etc. [3]

The uniqueness of one's body is well established in biology. Features that may be used for identifying individuals include [4]:

1. **DNA** – found in cells of the living organisms, this acid carries genetic information.
2. **Eye features** – human iris, retina and scleral veins can be used in eye scans.
3. **Face** – full face scan is often used for user recognition, for example in mobile devices and laptops [5].
4. **Fingerprints and finger shape** – fingerprints are widely used in forensics [6] and in digital scanners on mobile devices and laptops.

Other, less common methods for identifying a person are may include ear shape, gait, hand shape, heartbeat, keystroke dynamics, signatures, vein scans and voice recognition.

2.1 Keystroke Dynamics

One method of extracting data from a person's behavior is via *keystroke dynamics*. This type of behavioral biometrics is acquired from a user by means of a keyboard or other typing device and records and extracts features from the way the keyboard is used. Most commonly used and almost universally applicable to any keyboard device is the measurement of timings between each character typed. If the user uses a physical keyboard, it is also convenient to derive the following features [7]:

1. **Hold Time**: time between pressing and releasing a key.
2. **Down-Down Time**: time between pressing one key and pressing the next key.
3. **Up-Up Time**: time between releasing one key and releasing the next key.

4. **Up-Down Time**: time between releasing one key and pressing the next key.

5. **Down-Up Time**: time between pressing one key and releasing the next key.

Keystroke dynamics focus primarily on identifying a user's rhythmic patterns in their keystrokes. Such data can be used in conjunction with for example a password or a passphrase as a means of additional protection against password theft – this concept was tested as in 1990 [8]. By 1997, clustering methods were already being used in experiments on user data on a small scale (42 profiles) [9]. Since then, algorithms used for such data evolved, with the raise in popularity of neural networks.

In 2020, a combined CNN+RNN approach was used to obtain the results of ERR of 2.36% on Buffalo dataset and 5.97% on Clarkson II datasets [10]. Other approach from 2017 involved a CNN to create a multiple classification model – this method is mostly suitable for smaller datasets with smaller number of users, as opposed to creating a personalized model for each user [11]. This method had an ERR of 2.3%.

A Convolutional Neural Network (CNN) has a fixed node ordering and operates on a grid. Input data must firstly be mapped onto such a grid to be used with a CNN. There are ways to map many types of data into such format. In Lu et al. this involved applying the convolution layers over feature vectors, which were constructed in the following manner: for pairs of keys pressed in succesion in the sequence, a feature vector is created with fields: ID of first key, ID of second key, hold duration of first key, hold duration of second key, DD time (time between first press and second press) and DU time (time between first press and second release). After applying the CNN layer, GRU layers were used.

Another aspect of the keystroke dynamics recognition methods is how and when the data is collected. The systems can either work with some specific strings being typed by the user [11] or with the users being free to type anything within some length constraints [10]. In this project, the second approach was chosen as the more realistic one.

With some keyboards it may be more difficult to gather all the possible features. Even basic feature, such as the hold time can prove difficult to gather when using for example GBoard on mobile devices, which does not naturally send key press and key release information to the application [12]. This information can thus only be gathered in approximation or by building another virtual keyboard application. This, however, has its drawbacks. The users are generally used to one type of keyboard (on mobile it may be for example GBoard or SwiftKey), so forcing them to use another type of keyboard may be detrimental. Same person may write somewhat differently on different keyboards and machines. This study includes a small subsection on cross-smartphone compatibility of the model, for example concerning two users using each others' smartphones.

While the model may be less accurate because of the lack of features, there can be some ways to mitigate it. Some other features can be added, which are largely specific to mobile devices, such as accelerometer data, or a larger sample can be used. A few of those options were considered by the researchers, and the results are discussed in the following chapters.

Keystroke identification can also rely on other data gathered from the keyboard, such as the character average frequencies, most common connections between characters or other statistics [13]. These statistics can be modeled in many ways. If the average Up-Up Time between two keys is gathered from the data, a graph can be formed, having additional features as see fit by the designers. Such graphs were constructed for the Neural Network models constructed in this study, which will be discussed in the next chapter.

Chapter 3

Graph Convolutional Networks

Graph can be defined as mathematical structure G consisting of a set of vertices V and a set of edges E, where each edge can be described as an unordered pair $\{v_1, v_2\}$ of some vertices $v_1, v_2 \in V$ for undirected graphs or an ordered pair (v_1, v_2) of some vertices $v_1, v_2 \in V$ [14]. Such structures, along with their many variations and generalizations, can be used for describing entities, which are related to each other in some way. An example of such model could be a computer network graph or citation network. Neurons can also be modelled in a similar way. Relation data can often be best described using such graphs [15].

Some problems relating to such data can be solved using Convolutional Neural Networks – this can also be the case for keystroke dynamics data [10][7]. However, it can be reasoned that the Graph Neural Networks can also perform such tasks, with connections in graph data being used more directly in the model itself.

3.1 Graph Neural Networks

Graph Neural Networks (GNNs) are designed for graph inputs. The resulting outputs are also graphs (specifically, they are node embeddings representing a graph), allowing for transforming information in the graph's nodes, edges and global context, such as metadata about the graph, aggregated information, graph features etc. [16] GNN do not change the connectivity of the input in the output.

Graphs in GNNs are represented with two main components: the adjacency matrix A and the matrix of node features $X \in \mathbb{R}^{|V| \times m}$, where m is the number of features for each node. The feature vector for a node can be any data describing it, such as age or gender for a social network graph. GNN models are constructed with layers, where each layer performs processing in two steps:

- Message computation: each node computes a message

$$m_u^{(l)} = \text{MSG}^{(l)}(h_u^{(l-1)}), \quad u \in N(v) \cup \{v\} \tag{3.1}$$

  where:

  - $m_u^{(l)}$ represents the message computed for node u at layer l.
  - $\text{MSG}^{(l)}$ is the message function at layer l.
  - $h_u^{(l-1)}$ is the feature vector of node u from the previous layer $(l-1)$.
  - $N(v)$ is a set of neighbors of node v.

- Message aggregation: each node aggregates messages from its neighbors

$$h_v^{(l)} = \text{AGG}^{(l)}\left(\left\{m_u^{(l)} : u \in N(v)\right\}, m_v^{(l)}\right) \tag{3.2}$$

  where:

  - $h_v^{(l)}$ is the updated feature (embedding) of node v at layer l.

  To prevent losing message from node v itself, the messege from node v is included after aggregating messages from all its neighbors.

- Additionally, an activation function ϕ (e.g. ReLU, sigmoid) is applied to the message or the aggregation.

For Graph Convolutional Networks message computation and aggregation can be represented with the following formula:

$$h_v^{(l)} = \phi\left(\sum_{u \in N(v)} \frac{1}{|N(v)|} W^{(l)} h_u^{(l-1)}\right) \tag{3.3}$$

where: $W^{(l)}$ is a learnable weight matrix used to transform the feature vector.

The optimal number of layers in GCN is usually small [17]. Adding too many layers does not necessarily improve performance and may even degrade it due to over-smoothing. The number of layers should be selected based on the specific problem and graph structure.

3.2 Convolutional Networks and Graph Convolutional Networks

Convolutional Neural Networks work with grid-like data structures, such as images, where each element (e.g., a pixel) has a defined spatial relationship with its neighbors. An image can be treated as a special type of graph where each pixel is a node connected to its neighboring pixels by edges. This analogy helps in understanding that while CNNs process regular grids with a fixed neighborhood size, they can be seen as a specific case of Graph Neural Networks operating on structured data. This regular structure allows convolutional filters to move systematically across the input, helping to extract different levels of features. As a result, CNNs excel at computer vision tasks. However, graphs do not provide the structured data layout required by CNNs. They can have varying numbers of neighbors and lack a consistent ordering, which makes applying CNNs directly ineffective for graph data.

Graph Convolutional Networks solve this problem by directly handling graph-structured data. Since nodes in graphs do not follow a specific order and can connect to different numbers of neighbors, GCNs use the graph's adjacency matrix to gather information from neighboring nodes. This method focuses on relationships between nodes (who is connected to whom) rather than their exact positions. Consequently, GCNs effectively capture both local and global structures in graphs, making them suitable for tasks like node classification, link prediction, and graph classification.

3.3 Graph-level prediction and classification in GNN

Supervised learning on graphs can be achieved by labeling either nodes, edges or whole graphs [15]. In typical training pipeline, an input graph is transformed into a node representation accepted by the network, which then transforms the data in the manner described above. Output node embeddings are then used to create a prediction head, which is used, together with labels and

some loss function and evaluation metrics, for the prediction task. There are different prediction heads for node-level, edge-level or graph-level prediction [15]. For nodes, predictions can be made directly using node embeddings – this can be done by using a classification layer, like a dense layer followed by a Softmax layer [16]. For the edges, this must be done on pairs of nodes. For global graph predictions a pooling of node embeddings can be performed. Options for pooling include for example global mean pooling, global max pooling or global sum pooling. For classification purposes, which can be thought as k-way prediction task, node-level classification can be achieved via cross entropy (CE) loss function for i-th example in training:

$$\mathrm{CE}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) = -\sum_{j=1}^{K} \mathbf{y}_j^{(i)} \log(\hat{\mathbf{y}}_j^{(i)}) \tag{3.4}$$

where:

- K is the number of classes
- $\mathbf{y}^{(i)}$ is a one-hot label encoding
- $\hat{\mathbf{y}}^{(i)}$ is a prediction after Softmax

Total loss over the training examples can then be calculated as a sum of cross entropy for each example.

3.4 Metrics

Choosing a correct metric for a machine learning model is an important step for testing its performance. Furthermore, in the case of this project, a metric for testing the whole collections of models needs to be selected. When considering the functioning of authentication system, these metrics are often used [18].

3.4.1 Accuracy

Accuracy measures the proportion of correct predictions (both positive and negative) over all predictions. It is calculated as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.5}$$

where:

- TP: True Positives (correctly accepted genuine users),
- TN: True Negatives (correctly rejected imposters),
- FP: False Positives (incorrectly accepted imposters),
- FN: False Negatives (incorrectly rejected genuine users).

While accuracy is a commonly used metric, during the evaluation of models it was found that this score gave little information about the performance of the models. Most importantly, accuracy does not distinguish between false positives and false negatives, which from the perspective of this project differ in importance. False positives are considered more influential, as this kind of errors would allow imposters to gain access to the system, while false negatives would only force the user to repeat the authentication procedure.

3.4.2 Precision and Recall (True Positive Rate, TPR)

Precision measures the proportion of correctly predicted positive samples among all samples predicted as positive. It is given by:

$$Precision = \frac{TP}{TP + FP} \tag{3.6}$$

Recall measures the proportion of correctly predicted positive samples among all actual positive samples. It is calculated as:

$$Recall = \frac{TP}{TP + FN} \tag{3.7}$$

These measures are commonly used in machine learning and information retrieval setting, under the assumption that the negative class, and consequently, the number of true negatives, does not matter as much as the positive class. The focus on the correct recognition of the positive class matches the use function of models in this project.
As such, precision and recall were used when training models, to measure the performance on validation set and tuning hyperparameters.

3.4.3 False Acceptance Rate (FAR) and False Rejection Rate (FRR)

The False Acceptance Rate (FAR) represents the proportion of negative samples (imposters) that are incorrectly classified as positive. It is calculated as:

$$FAR = \frac{FP}{FP + TN} \tag{3.8}$$

The False Rejection Rate (FRR) represents the proportion of positive samples (genuine users) that are incorrectly classified as negative. It is calculated as:

$$FRR = \frac{FN}{TP + FN} \tag{3.9}$$

From the perspective of this project, FAR and FRR were considered to be the most important and informative metric. These metrics directly represent the security and usability of a system from the perspective of its users and therefore showcase the performance of the trained models in their intended. As such, the performance of the collection of models will be evaluated using these metrics.

3.4.4 Equal Error Rate (EER)

In authentication systems, there is often a trade-off between FAR and FRR. By lowering the decision threshold, more users, both genuine and imposters would be authenticated, thus lowering FRR and increasing FAR. Conversely, a higher decision threshold would lead to a increased FRR and decreased FAR.
The Equal Error Rate is the value of these metrics at a threshold where FAR = FRR. If no such threshold can be found, EER can is calculated as

$$EER = \frac{FFR + FAR}{2} \tag{3.10}$$

at a threshold where the difference between FFR + FAR is minimal.

The equal error rate is a useful metric which allow for easy comparison between different methods and applications. Naturally, the threshold used to measure the EER is not the optimal

value for the perspective of the final application. As noted in the section about accuracy, the number of false negatives is the major concern for this project. Therefore, a higher false rejection rate is acceptable, if it results in a comparable drop in the false acceptance rate. Despite this fact, the EER will also be reported along with model performance, as it is a common metric used in authentication systems and allows comparing the outcomes to other findings using Keystroke Dynamics.

Chapter 4

Gathering keystroke data on mobile devices

There are many ways to recognise a phone user using biometrics, such as scanning fingerprints or facial recognition. It is very useful for security purposes. The ease of use and reliability have made passwords less popular and led to their replacement by biometrics. However, since other biometric methods are also available, it is reasonable to test if biometrics derived from writing button press intervals and phone orientation could also be a reliable way to recognise the user. To collect data and test the results, the mobile application was created. The main goal of the application is to gather data with an easy-to-use, intuitive interface, send the data to a server for training purposes, check if the model recognises the user.

As previously stated in previous chapters, State of the Art models can actually perform well on such data [10]. These models are however usually trained on data gathered from physical keyboards. Additionally, the Neural Network model created for user identification was chosen to be based on Graph Convolutional Networks, which differ from models used by many researchers in the past. Because of that, an important part of the project was a study of results and data gathered, which is presented later.

4.1 Use cases

The main goal of this application was the identification of users based on their distinctive typing behavior, which is known as keystroke dynamics. By using the machine learning models and encrypted server transmission for analysing the collected data, the application aimed to provide an additional layer of security beyond passwords or basic biometrics. This project tried to establish whether this type of behavioral biometric can be a reliable way of user authentication.
The following use cases illustrate how the user would interact with the application and its features.

- **Logging into the application**
  - **Purpose:** Allowing the user to log in and associate the application with a predefined `ID`.
  - **Steps:**
    * The user opens the application.
    * On the `Login Screen`, the user enters their `ID` in the input field.
    * The user clicks the `Login` button to proceed.
    * The application stores given `ID` for further operations.

  - **Result:** The user is logged in and is redirected to the `Home Screen`.

- **Data collection from key presses**
  - **Purpose:** Storing users keyboards interaction data for analysis.
  - **Steps:**
    * The user navigates to `Training Screen`.
    * The user types a predefined number of characters in total throughout 5 phases to complete training.
    * The application registers data for every key press, including:
      · Key ID (e.g., `A`, `h`, `3`)
      · Timestamp of key press action
      · Press duration
      · Accelerator data (X, Y, Z axis)
    * Data is stored in `KeyPressEntity` object for further processing.
  - **Result:** Full key press data is saved in the application, ready to be transformed into `TSV` format, transmitted to the server, or stored locally.

- **Testing how well the model recognises the user**
  - **Purpose:** Verifying if the user entering data is the one associated with their `ID`.
  - **Steps:**
    * The user navigates to `Testing Screen`.
    * The user types a predefined number of characters into the input field.
    * The application registers the key press data.
    * The data is transformed into TSV format, stored locally, and sent to the server:
      · The application ensures the connection is secured with `SSL/TLS`.
      · The `POST` method is used to send the data.
    * The server processes the data using the trained model.
    * The server sends back response to the application, including:
      · Information on whether the user was recognised.
      · The percentage of compliance with the user.
  - **Result:** The application displays the result to the user.

4.2 Server structure and communication with the application

The server is written in Python and implemented using FastAPI [19], a high-performance asynchronous framework for building APIs. The primary roles of the server include receiving keystroke data from the mobile application, interacting with the SQLite database for data storage and retrieval, processing the keystrokes and extracting relevant features, training and validating Graph Neural Network models, and performing inference to verify user identity. The functionality related to data extraction, model training, and inference is described in Chapter 5.

The server communicates with the mobile application using HTTP POST requests. All communication is secured using SSL encryption to ensure data integrity and privacy during transmission.

4.2.1 Endpoints and their functionality

The server provides three endpoints for interaction with the mobile application. All endpoints share the same parameters: a query parameter 'username' identifying the user and a raw TSV file in the request body.

- POST `/upload_tsv`: This endpoint allows the mobile application to upload keystroke data in TSV format. The server parses the TSV content into a string, verifies its structure, and loads it into the SQLite database. Additionally, the data is stored in a designated directory. A confirmation message is returned if the data is successfully processed and stored. An error message is returned if the data cannot be processed or stored due to validation issues or other errors.
- POST `/train`: This endpoint has the same functionality as `/upload_tsv` but additionally invokes the training process for a user-specific GNN model. It should be called with the last portion of data to ensure that the model is trained on a complete dataset. The server stores and validates the last portion of the training data before invoking the function responsible for training. A success message is returned upon the successful completion of model training. An error message is returned if the training process fails.
- POST `/inference`: This endpoint is responsible for invoking the inference process on a user-specific GNN model. It performs user verification by running inference on the provided keystroke data and returns a prediction score along with a classification result indicating whether the user was correctly identified.

4.2.2 Database layer

Besides saving users' keystroke data as TSV files in a specified directory, the server uses SQLite as the database management system to store the data. The database is managed by the 'database_utils' module, which provides functions for creating tables, inserting data, and retrieving stored information.

The only table in the database is 'key_press', which records individual keystroke events. The table includes the following fields:

- `user_id (TEXT)`: Identifier of the user.
- `key (TEXT)`: Key pressed by the user.
- `press_time (TIMESTAMP)`: Timestamp of when the key was pressed.
- `duration (INTEGER)`: Duration of the key press in milliseconds.
- `accel_x, accel_y, accel_z (REAL)`: Accelerometer data captured during the key press.
- `timestamp (TIMESTAMP)`: Timestamp of when the record was added to the database.

The 'timestamp' field is essential in ensuring that training examples consist of key presses from a single writing session without mixing data from different sessions. This separation is important for proper feature extraction and model training.

Key functions implemented in the 'database_utils' module include:

- `create_table()`: Creates the 'key_press' table if it does not already exist.
- `drop_table()`: Deletes the 'key_press' table.

- `add_tsv_values()`: Inserts keystroke data into the database.
- `load_str()`: Processes TSV data provided as a string and inserts it into the database.
- `load_file() and load_dir()`: Load keystroke data from TSV file or directory with TSV files and insert them into the database.

4.2.3 Server deployment

The server can be deployed either locally or on a remote host. The main server script 'server.py' uses 'uvicorn' to run the FastAPI application. SSL/TLS encryption is configured to secure all communications, with the SSL key and certificate specified in the 'main()' function. The server listens on port 8000 and supports HTTPS requests by default.

4.3 Mobile application for data gathering and model testing

The application was written for Android devices supporting Android 8.1 or newer. As of 2024 [20], more than 93% of Android devices should be compatible. The Android platform was chosen, as it was easier to test on and find a study group of the Android users as opposed to the iOS users (according to [21], significantly more people in Poland, where the researchers are based in, use Android devices).

Technology used in the mobile application itself is Jetpack Compose, which is quoted by Google to be "Android's recommended modern toolkit for building native UI" [22]. Language used is Kotlin. Persisence is achieved by using Android Room, which provides an abstraction layer over SQLite database, which is used for data collection.

4.3.1 Model View ViewModel and DataStore

The application uses Model-View-ViewModel (MVVM) provided by Jetpack Compose design pattern to support a clear separation of concerns.

- **Model:** Data is modeled using `KeyPressEntity` class, which represents a single key press event. It includes:
  - **Key** (`String`): The key pressed by the user.
  - **Press Time** (`Long`): The exact timestamp of the key press event.
  - **Duration** (`Long`): The time elapsed since the last key press event.
  - **Accelerometer Data** (`Float`): Not used currently but could be useful for the future development of the application.

```kotlin
data class KeyPressEntity(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val key: String,
    val pressTime: Long,
    val duration: Long,
    val accelX: Float,  // Accelerometer X axis
    val accelY: Float,  // Accelerometer Y axis
    val accelZ: Float,  // Accelerometer Z axis
)
```

FIGURE 4.1: KeyPressEntity.kt

The `KeyPressEntity` is stored in a local SQLite database via Room.

- **View:** The user interface is implemented using **Jetpack Compose**, a declarative UI framework. Key components of the view contain:
  - **Input Fields:** Lets users enter their credentials (University ID) and use the application for training or testing by pressing keys.
  - **Completion progress:** Informs users on what phase they are and displays progress of completion, linked to the `phasesCompleted` state in the `MainViewModel`.
  - **Buttons:** Used for logging in, logging out, jumping phases and sending or downloading the data collected through training or testing stage.
- **ViewModel:** This role is fulfilled by `MainViewModel`, which manages the application logic, handles interactions between the model and the view, and maintains the state of the app. The `MainViewModel` class manages this operations through:
  - **Logic Handling:** Methods such as `login()`, `logout()`, `clearDatabase()`, and `onKeyPress()` are responsible for managing user state and data.
  - **State Management:** Stores states `isLoggedIn`, `username`, and `phasesCompleted`, which are used to dynamically update the user interface.
  - **Data Management:** Connects with the `keyPressDao` database to process data. `onKeyPress` saves key press events into the database, `exportDataToTsv` exports the collected data into TSV files.

In the app **DataStore** is used for storing login state and the user's ID. It has been implemented in `UserPreferences` class, and stores data such as:

- `LOGGED_IN_KEY` - login state
- `USERNAME_KEY` - user's ID.

```kotlin
companion object {
    val LOGGED_IN_KEY = booleanPreferencesKey( name: "logged_in")
    val USERNAME_KEY = stringPreferencesKey( name: "username")
}
```

FIGURE 4.2: UserPreferences.kt

This data is stored in the app's preferences file and can be accessed via dataStore object using:

- `isLoggedIn` - returns login state as `Flow<Boolean>`
- `username` - returns user's ID as `Flow<String>`
- `setLoggedIn()` - saves login state and user's ID into DataStore

```kotlin
// Get the login state
val isLoggedIn: Flow<Boolean> = dataStore.data
    .map { preferences ->
        preferences[LOGGED_IN_KEY] ?: false
    }

// Get the login state
val username: Flow<String> = dataStore.data
    .map { preferences ->
        preferences[USERNAME_KEY] ?: "nn"
    }

// Save the login state
kubag
suspend fun setLoggedIn(loggedIn: Boolean, username: String) {
    dataStore.edit { preferences ->
        preferences[LOGGED_IN_KEY] = loggedIn
        preferences[USERNAME_KEY] = username
    }
}
```

FIGURE 4.3: UserPreferences.kt

The use of DataStore enabled the data to be stored securely, accessed and modified easily, and it is always available, which makes it a reliable and efficient way to manage user preferences and app state.

4.3.2 User Interface Design

The application design follows a minimalistic approach to make it intuitive and easy to use for everyone.

- `Login Screen` 4.4
  After launching the application for the first time, the user is presented with the `Login screen`. It contains `TextInput` field for entering the university ID, which was evenly distributed among contributors to simplify testing, and the `Log in` button which stores the ID and navigates the user to the `Home Screen`.
- `Home Screen` 4.5
  The home screen displays three buttons and a simple note explaining what the user should do. The `Logout` button navigates back to the `Login Screen`, while two other buttons lead to either testing or training screens.
- `Training Screen` 4.6
  The training screen is designed for collecting data for training purposes. It includes `TextInput`

field for typing user input, a `Button` to proceed to the next phase, and `Text` indicators showing how many chars are needed to complete the phase (300 each phase) and how many phases remain (5 phases in total) to complete the process of collecting training data. Additionally, there are two notes instructing the user to maintain the writing style throughout the whole process and to change the position after each phase while writing (explained in subsection 4.3.3).
To ensure that typing is done in the most natural way, the default android keyboard is used.

- `Testing Screen` 4.7
  The testing screen includes `TextInput` field for typing the test input, a `Button` that sends the input to the server and stores it locally, and `Text` indicators showing how many characters need to be written (in this case, 100). After fulfilling the requirements, the user sends their input to the server, which evaluates it against the trained model. The server then returns feedback and the user is presented with a recognition rate percentage on a circular progress bar and a message indicating whether the model recognised them or not 4.8.

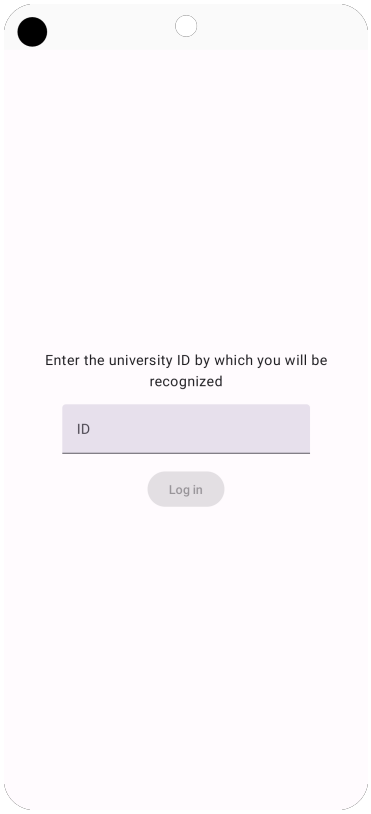


FIGURE 4.4: Login screen

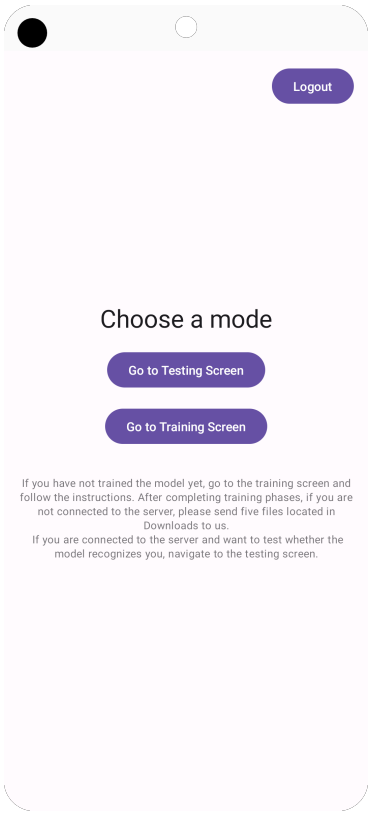


FIGURE 4.5: Home screen

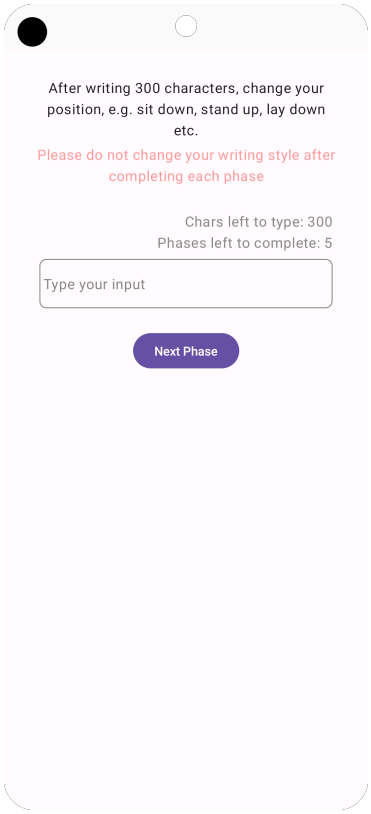


FIGURE 4.6: Training screen

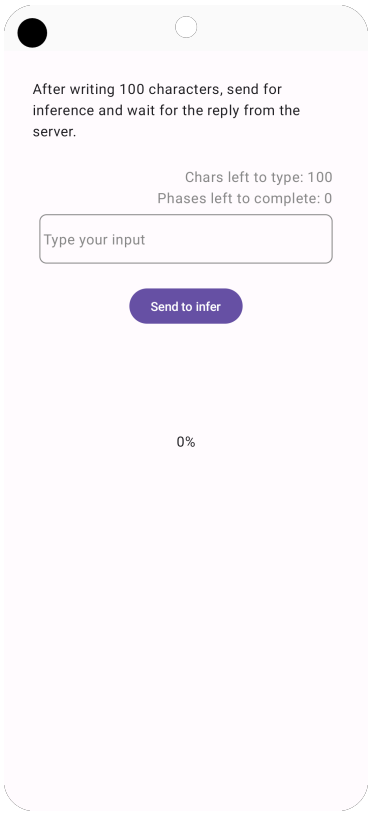


FIGURE 4.7: Testing screen

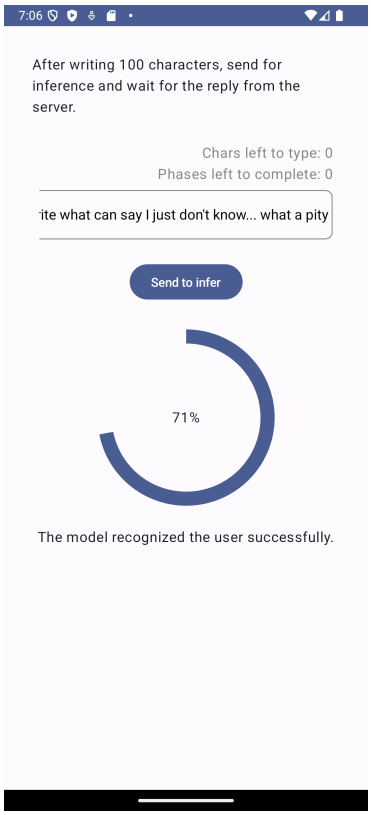


FIGURE 4.8: Testing screen example

4.3.3 Data Collection Process

Data collection occurs in two stages, training and testing

- Training data collection begins on the `Training Screen` 4.6, where the user is asked to input meaningful sentences. The process consists of 5 phases. Each phase requires the user to type 300 characters. Once the requirement is met, the user progresses to the next phase until all 5 phases are completed (1500 characters in total). Additionally, there is a note instructing the user to maintain a consistent writing style throughout all phases. Also the user is asked to change their position after each phase while writing. This is important for accelerometer data collection (which is not used at the moment), as it helps exclude situations where the phone is lying on the table or being held in an atypical way.
- Testing data collection takes place on the `Testing Screen` 4.7, where the user is required to write 100 characters, again in meaningful sentences. Once this is done, the testing phase is complete.

After each phase, the collected data is saved in a `.tsv` file, sent to the server, and stored locally in the phone's downloads directory. The `exportDataToTsv` 4.9 function from `MainViewModel.kt` handles the export of key press data.
Firstly it retrieves latest key press events using the `keyPressDao.getNLatestKeyPresses` method, converting the data into a `.tsv` format using the `keyPressesToTsv` function.
Depending on which phase the user is in, the function determines the different type of operation to perform.

- If the user is in inference phase, the data will be used for inference.
- If the user is in training phase, the data will be used for training.
- If the number of completed phases exceeds the required amount, the function exits without performing any other action.

After processing data `saveTsvToDownloads` stores data locally, and `sendTsvToFastApi` sends data to the server (An example of the `.tsv` file containing the saved data is shown in Figure 4.10). The username, and the relevant phase is included in the file name.
This function ensures that after each phase of training and testing, the data is collected, stored, and transmitted.

```kotlin
fun exportDataToTsv(
    context: Context, minPhases: Int, symWritten: Int, onResponse: (String) -> Unit) {
    viewModelScope.launch {
        val keyPresses = keyPressDao.getNLatestKeyPresses(symWritten)
        val tsvData = keyPressesToTsv(keyPresses)
        var apiString = "upload_tsv"
        var phases = phasesCompleted.intValue

        if (minPhases == -1) {
            apiString = "inference"
            phases = -1
        }
        else if (phasesCompleted.intValue == minPhases) {
            apiString = "train"
        }
        else if (phasesCompleted.intValue > minPhases) return@launch

        username.take( count: 1).collect { user ->
            Log.i( tag: "TAG",  msg: "In export to tsv function")
            saveTsvToDownloads(context, user, phases, tsvData)
            sendTsvToFastApi(tsvData, user, apiString, context, onResponse)
        }
    }
}
```

FIGURE 4.9: MainViewModel.kt

key	press_time	duration	accel_x	accel_y	accel_z
h	1733570714489	166	-0.56100005	4.97895	8.565001
u	1733570714323	116	-0.822	4.842	9.27105
d	1733570714207	626	-0.57795	4.81605	7.9189506
SP	1733570713581	1873	-0.171	4.78695	8.7949505
d	1733570711708	104	-0.22605	5.1460505	8.62695
l	1733570711604	168	-0.279	5.1439505	8.74005
r	1733570711436	177	0.00795	5.2830005	9.58695
o	1733570711259	121	0.26205	5.0070004	8.304001
w	1733570711138	793	-0.11595	5.19705	8.57505
SP	1733570710345	115	-0.21900001	5.24205	7.8919506
e	1733570710230	151	-0.22305001	5.2200003	7.9570503
r	1733570710079	138	-0.33795002	5.22795	9.183001
i	1733570709941	112	-0.23805001	5.2249503	8.698951
t	1733570709829	245	-0.19695	5.21595	8.677051
n	1733570709584	92	-0.14400001	4.9549503	8.326051
e	1733570709492	531	-0.26895002	4.9669504	8.598001
SP	1733570708961	129	-0.12795001	5.0479503	8.112
e	1733570708832	232	-0.50805	5.15805	8.479051
h	1733570708600	100	-0.75705004	4.9429502	8.191051
t	1733570708500	133	-0.37905002	5.328	8.67
SP	1733570708367	171	-0.27105	5.1529503	8.26305
n	1733570708196	178	-0.64995	5.01195	8.163
i	1733570708018	399	-0.82395005	5.00505	8.982
SP	1733570707619	338	-0.47205	4.99095	8.75595
s	1733570707281	96	-0.702	5.06505	8.30595
i	1733570707185	198	-0.71205	4.485	9.42405
DEL	1733570706987	150	0.039	5.0620503	8.566951

FIGURE 4.10: An example of the `.tsv` file containing saved data.

4.3.4 Communication with the server

The application communicates with the server using an HTTPS connection.

- **Server URL and Request Structure**
  - The server is accessed via an HTTPS endpoint. The base URL is defined as `https://192.168.1.100:8000`.
  - The API endpoint is dynamically created with the use of a route and query parameters to the base URL. For example, the endpoint for sending data contains the username as a query parameter:
    `https://192.168.1.100:8000/<api_string>?username=<username>`
  - The data is sent using the POST method.
- **Data format**
  The data sent to the server is stored in `.tsv` (tab-separated values) file, containing headers and the detailed information about key presses.
- **Secure Connection Setup**
  - The application uses `OkHttpClient` library for handling network requests.
  - A `.cert` certificate (stored in `res/raw/cert`) is used to establish a secure and trustworthy `SSL/TLS` connection.
- **Sending request**
  - Requests are executed asynchronously using the `enqueue` method.
  - If successful, the server's response is processed, and the application displays the result to the user.
  - On failure, the error is logged, and the user is notified.
- **Error Handling**
  - Network errors (e.g., problems with connection) and server errors are logged for easier debugging.
  - A callback system is implemented to communicate feedback to the user.

4.3.5 Potential uses

The application analysing the typing habits of users (keystroke dynamics) has a wide range of applications in a various fields, providing an additional layer of security. It could for example be used to verify user's identity in online banking or for continuous identity checks during sessions in corporate environments. Keystroke dynamics has potential to become a successful and user-friendly security measure.

Chapter 5

GCN Model

In this project, the goal was to use the graph networks that can naturally arise from keystroke data to infer the user's identity at a graph level. A key characteristic of a project was that the use of any keyboard already installed on the user's mobile phone causes some issues with gathering keystroke temporal data, as mentioned in the second chapter. Because of that, this project used data involving only Up-Up times, with additional use of accelerometer data being considered by the researchers and discussed in the next section.

Moreover, this project focused on creating a collection of models, one model for each user that performs binary classification rather than a single large model for multiclass classification. This decision was made for several reasons. First, this scheme allows for models to be trained on demand, as soon as a new user provides all the training data to the mobile application. Second, new users do not require retraining of the entire model, as only one new model needs to be created, and provided all models have learned their target users sufficiently, they would be able to reject such new users without further tuning. Finally, the one user per model scheme allows for inference to take place locally, on the target user's device. This would eliminate the need for remote communication with the server, thus increasing the mobile application's reliability and security. On-device inference is an area of active research [23]. However, the complexity of such a solution was deemed too great and beyond the scope of this project.

5.1 Choosing features for Neural Network model

A major part of this project was dedicated to finding the correct way to represent the collected user data in the form of a graph. The most interesting of these representations are discussed in this section.

5.1.1 Graph creation and feature encoding

The input for graph creation consists of two main parts, the duration of time between individual key presses, and the character of the key that was pressed. A natural way to represent such input was to map each unique character in the input sequence to a node in the graph. Directed edges were added between nodes that represent characters appearing after each other in the input sequence. Each time the same pair of characters appears in the input text, it maps to the same edge. For each such pair, the duration is added to a list of attributes associated with that edge, which will be aggregated in later stages into a form suitable for the GCN model. It is also possible to model such pairs using a multiedge graph, as such models have been shown to perform well in other domains. However, this approach was not considered in this study.

A simple visualization of such graph, created from the input text: "Cat and dog and .", with arbitrary keystroke durations, is shown in figure 5.1.

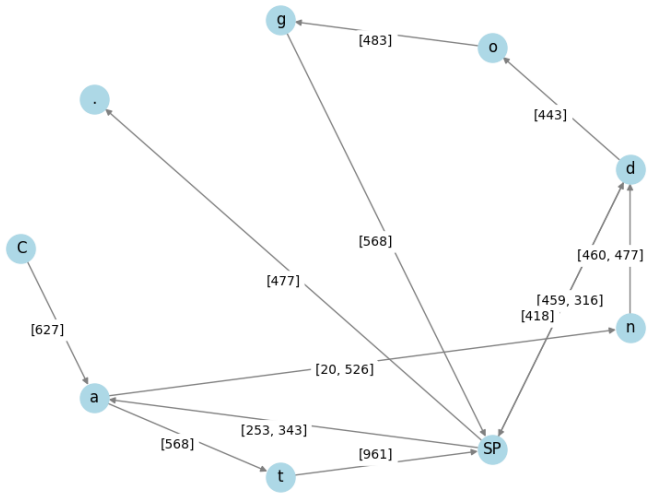


FIGURE 5.1: Graph for the sequence "Cat and dog and ."

The structure of the resulting graph highly depends on the length of the input sequence. A shorter sequences produce smaller and sparser graphs, while a longer input sequences result in graphs with more nodes and edges. An example taken from the training dataset is shown below: one graph with 40 characters and another with 70. Edge weights have been omitted for clarity.

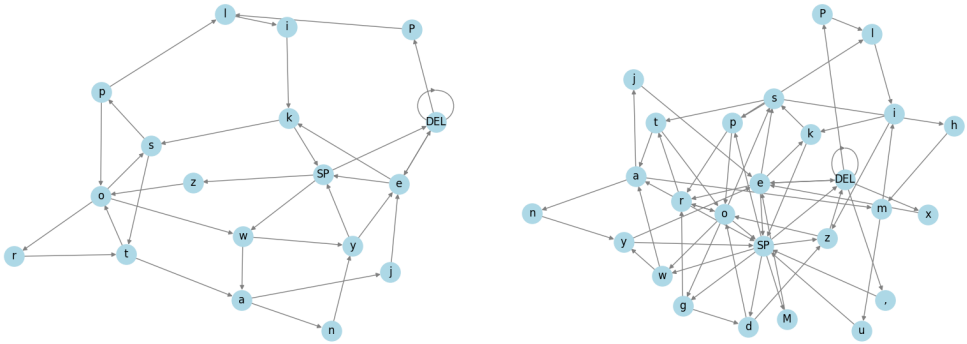


FIGURE 5.2: Comparison of graphs constructed with different input sequence lengths of 40 and 70.

5.1.2 Edge attributes encoding

In order for the graph to be a valid input for a GCN network, edge attributes need to be converted into node features. Two methods were identified to encode this information into node features. For each node i:

1. Two values representing the average duration before and after the key represented by i was pressed.

2. A two-dimensional vector of values with a size of [number of allowed characters, 2]. Each key that appears in the input is assigned a number. The n-th row in the vector corresponds to a node, with a key assigned the number n, now called node j. The n-th row contains two

values: the average duration on the edge from i to j and the average duration on the edge from j to i. The values for which edges do not exist were assigned 0.

The key difference between these two approaches is the level of aggregation. Method 1 aggregates all the edge information into two values, while method 2 aggregates it into a vector of values, although it imposes some limitations, such as assigning each key a unique index into this input vector. Furthermore, method 2 increases the overall size of the input data and the complexity of the model.

5.1.3 Character attributes encoding

As this project focuses on recognizing users of the Polish language, the character encodings were designed to make use of this fact. For encoding purposes, characters were divided into several groups:

- Letters – characters *a-z*, including diacritics.
- Numbers – characters *0-9*.
- Special characters – *space*, *tab*, *newline*, *backspace*, *dot*, *comma*, *exclamation mark*, *question mark*.
- Symbols – *, #, @, :, ;, ', ", (,), [,], {, }, <, >, /, \, —, &, %, $, ^, ~, _, +, -, =.
- Others – all other symbols.

Similarly, there is more than one way to encode key information into node features. Three methods were considered, all of which are variations of one-hot encoding of keys:

1. Full One-Hot Encoding. Each unique cased character is mapped to a different column in the vector, except for characters in the *Others* group, which are mapped to an additional column.
2. Compact Alphabet Encoding. A more compact version of the *Full One-Hot Encoding*. One-hot encoding is used for cased *Letters* and *Special characters*. *Numbers* are grouped and mapped to a single column in the vector. *Symbols* and *Others* are grouped together and mapped to another column in the vector.
3. Normalized Base Letter Encoding. *Letters* are simplified by converting them to lowercase and removing diacritical marks. These transformed letters are encoded in a one-hot vector. *Numbers* are mapped to a single column in the vector, *Symbols* and *Others* are mapped in the same way as in the *Compact Alphabet Encoding*. Two additional columns are added to indicate whether the original character was capitalized and if it had a diacritical mark.

These methods differ by their degree of aggregation. Methods 2 and 3 group certain letters together, mapping multiple characters to the same values, while method 1 provides a unique, one-hot encoded identifier for each node [24]. found that such node identifiers helped the model to learn certain structures in the data. However, providing node identifiers as input features works well only for a small and defined set of possible input nodes [15]. While this requirement appears to hold true for this specific task, it was found not to be the case. Many letters that appear to be common are, in fact, infrequent in the dataset. This is especially true for capitalised letters. For example the letter *'f'* appears 134 times, whereas *'F'* appears only 6 times. This is even more true for symbols, many of which appear only once in the entire dataset, these being: *'{'*, *'%'*, *'*'*,

':' and *'#'*. This means that the behavior of the models would be unpredictable for nodes with such identifiers.
Method 2 addresses the problem in the case of symbols and numbers. It does not, however help with processing infrequent capitalized diacritical letters. For example, *'l'* appears 875 times, *"ł"* 138 times, *"L"* 10 times and *"Ł"* only twice. With such a low number of occurances it is unlikely that the models could learn the relationship between these letters.
Method 3 was specifically designed to deal with this problem, which is achieved by grouping them together. The extra columns still allow for distinguishing between varaints of the same letter, which is important because the time to input these symbol differs greatly, especially in the case of diacriticals, which take on average of 898 miliseconds to enter, compared to 346 for letters without diacritical marks. Moreover, method 3 directly exposes the use of diacritical marks and uppercase letters as features, which, in theory, could allow the model to learn the impact of these features independently of the base letters.

5.1.4 Feature selection: accelerometer data

To improve the potential performance of the model, and to make further use of the capabilities of the mobile platform, accelerometer data was considered as an input feature for the model. This portion of the input data consisted of three values, a measurement of the acceleration in the x, y and z axes at the moment a keystroke was registered. These values were aggregated as an average for each node. Although these models performed well during training, quickly achieving low loss values, they failed to generalize, performing worse on validation and test datasets. For this reason, accelerometer data was not used for training and evaluating models discussed later.

5.2 Architecture

The foundation of a model for graph-level prediction consists of graph convolutional layers, a pooling function to aggregate node information, and a classification head, which outputs a single value, which is then passed to a sigmoid function. The architecture used in this project is illustrated in figure 5.3.

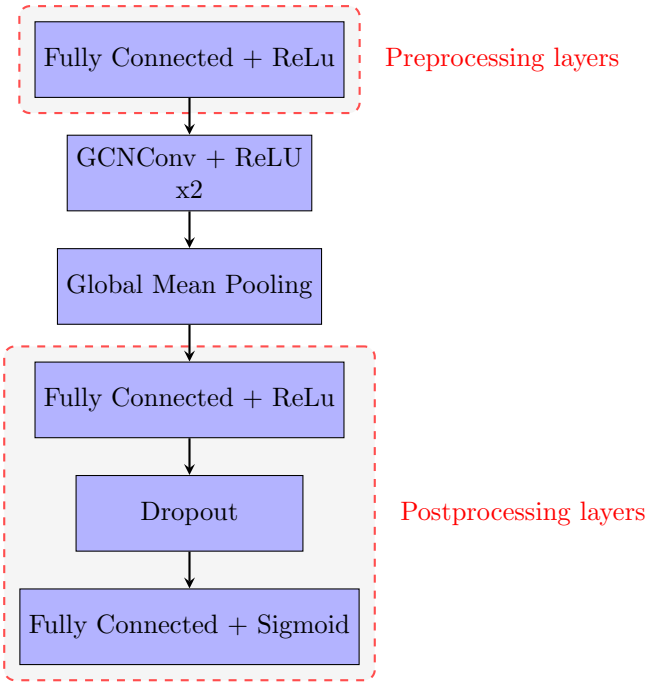


FIGURE 5.3: Model architecture

5.2.1 GCN + ReLU

The first two layers of the model, pictured in figure 5.3, are graph convolutional layers or GCNConv layers. The theory behind these layers was extensively discussed in chapter 3. After each layer a ReLU activation function was applied.

5.2.2 Global Mean Pooling

After two convolutional layers, a global mean pooling layer was applied to aggregate the node embeddings into a single vector.

5.2.3 Preprocessing layers

Considering the complexity of the input features, specifically the relationship between parts of the features used to represent characters discussed in subsection 5.1.3, a preprocessing fully connected layer was added to the model. Preprocessing layers can improve performance in cases where encoding node features is necessary, such as text inputs [15].

5.2.4 Postprocessing layers

The selected architecture uses two fully connected layers to transform the aggregated output into a single value, which is passed through a sigmoid activation function. The first fully connected layer was added after empirical experimentation which showed an improvement in performance.

5.2.5 Dropout

A dropout layer was added between the two fully connected layers, which randomly disable some of the neurons during the training process. Such layers have been shown to create more robust

models, that generalize better, as demonstrated by [25]. Additionally, dropout layers have also been found to improve expressiveness in GNNs [26].

5.3 Training and fine-tuning

5.3.1 Data division

For model validation, 5-fold cross-validation was used, splitting the training set described in section 4.3.3 into five parts, with training performed on four of them. The number of folds corresponds to the number of 300-character blocks that users were asked to input. Such splits validate that the trained model is able to generalize to a part of the input that may have been written at different times and in styles. Furthermore, it closely resembles the way the model was tested, that is on a completely different input sequence, collected in a separate part of the application.

5.3.2 Training parameters

5.3.3 Loss function

Binary Cross Entropy was chosen as the loss function for training, as it is commonly used with binary classifiers. The use of this loss function requires the model to output the probability of picking the positive class, therefore a sigmoid function must be applied to the output of the final layer. These two operations are combined into a single step in the implementation, by using $BCEWithLogitsLoss$, which provides greater numerical stability [27].

5.3.4 Optimizer

Adam was selected as the optimizer for updating model weights, as it adapts the learning rate for each parameter. The learning rate was empirically set to 0.001.

5.3.5 Tuning hyperparameters

The architecture described above allows for a large number of hyperparameter choices, such as the size of preprocessing, convolutional, and postprocessing layers. Furthermore, the variable length of input sequences and multiple possible feature encoding methods mean that a comprehensive search over the hyperparameter space is infeasible. Within the scope of this project the biggest emphasis was placed on finding the optimal input features and input sequence length, thereby identifying the best graph representation. Additionally, the effect of class imbalance on the training process of the collection of models was also explored.
All models were trained using 5–fold cross–validation, with the validation results averaged across folds. These were then used to compare the collections of models and to select hyperparameter values.

Chapter 6

Results

Evaluating the performance of machine learning models is crucial for understanding their effectiveness and limitations in practical applications. In this study, the developed models were tested on user data to assess their accuracy and generalization capabilities. Various factors, such as feature representation, input sequence length, and class balance, can significantly influence model performance.

6.1 Model performance on user data

This section focuses on the results of testing the collection of models that achieved the best performance during validation. It aims to show how this solution performs on the unseen test dataset and highlights the large differences between individual models within the collection of models.

6.1.1 Methodology

The results below were calculated as an average of each model's performance on sequences of characters of the same length as those on which the model was trained. The results presented below were obtained using the combination of input features listed in table 6.1, which yielded the best performance during model validation. The impact of each hyperparameter choice is discussed separately in dedicated subsections and compared using the testing provided by users. It is important to note that due to the large number of possible configurations, as well as the limited computing power that was available, not all combinations were tested.

Hyperparameter	Value
Input length	*40*
Edge encoding method	*Two values per node*
Character encoding method	*Normalized Base Letter Encoding*
Decision threshold	*0.8*

Table 6.1: Input feature choices.

6.1.2 FAR and FRR scores

The table below presents the false authentication rate and false acceptance rate for all models. The motivation for using these metrics was discussed in subsection 3.4.3.

User ID	False Acceptance Rate	False Rejection Rate
21	0.242	0.119
22	0.063	0.000
23	0.200	0.479
24	0.042	0.658
25	0.050	0.015
26	0.086	0.032
40	0.080	0.459
41	0.057	0.355
42	0.135	0.379
60	0.083	0.608
61	0.066	0.065
62	0.044	0.032
81	0.086	0.802
82	0.045	0.024
83	0.116	0.322
85	0.128	0.144
86	0.037	0.000
Average	0.092	0.264

TABLE 6.2: FAR and FRR scores for all models.

The same data can be visualized as a plot of the false acceptance rate versus the false rejection rate, as shown in figure 6.1.

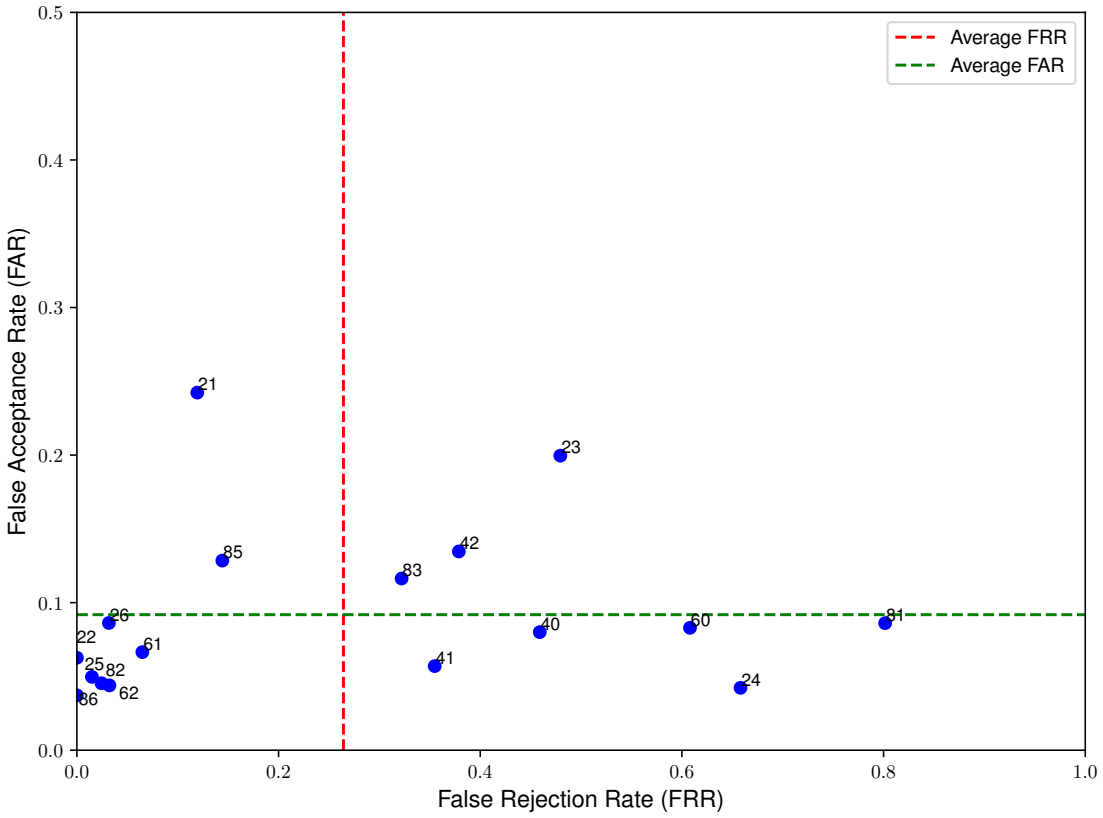


FIGURE 6.1: False acceptance rate versus False rejection rate.

Figure 6.1 demonstrates a large discrepancy between the models. At the same threshold level, some models were able to achieve a FAR and FRR score below 10%, while others failed to recognize more than half of the examples in the positive class.

This first group consists of models for users 86, 62, 82, 25, 26, and 61. These models were able to effectively learn and generalize the test dataset well. The other major group of models was those with an unacceptably high FAR exceeding 45% and a below-average FAR score. These group includes models for users 81, 24, 60, 40 and 23. The reason for such high FAR could be twofold, either the models simply failed to generalize to unseen positive examples, or they did but with a lower confidence level than the established threshold. Other models have varying performance and are difficult to categorize into coherent groups.

6.1.3 Equal Error Rate

To calculate the equal error rate, the decision threshold needs to be adjusted, which can be performed globally, for all models or individually at a per–model level. Both methods of calculating this metric were analyzed.

The global EER was calculated to illustrate the performance of the entire collection with a single value.

Equal Error Rate: 0.157

Threshold: 0.20

A low decision threshold was necessary to achieve an equal error rate. This suggests that most models are unable to recognize positive examples with a high level of confidence. Such a low value would not be practical in an applied setting and within the context of the system developed in this project.

The per-model EER can be used to examine the differences among the models. It is calculated by finding the equal error rate decision threshold for each model separately. The results are shown in table 6.3.

User ID	Equal Error Rate	Decision threshold
21	0.124	0.95
22	0.026	0.95
23	0.245	0.40
24	0.270	0.05
25	0.037	0.85
26	0.059	0.80
40	0.200	0.05
41	0.081	0.35
42	0.163	0.45
60	0.176	0.05
61	0.066	0.80
62	0.038	0.85
81	0.392	0.05
82	0.026	0.95
83	0.220	0.05
85	0.131	0.75
86	0.028	0.90
Average	0.133	–

TABLE 6.3: Per model equal error rate.

Table 6.3 illustrates the differences in the models' ability to generalize unseen positive examples. This is further demonstrated by examining the changes in FAR and FRR scores with respect to the decision threshold, which is show in figure 6.2.

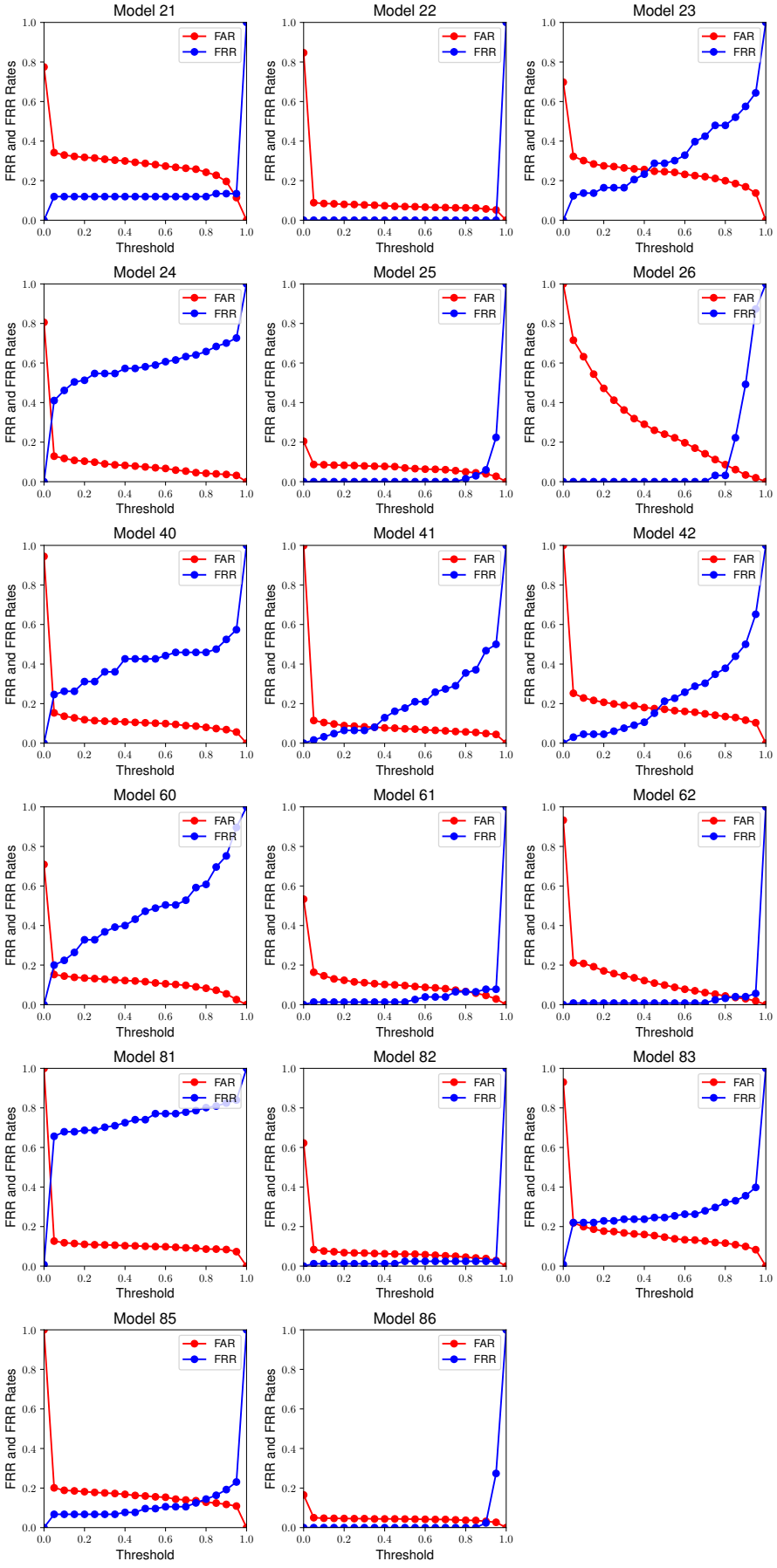


FIGURE 6.2: Change of FAR and FRR score with respect to decision threshold.

These plots show the relationship between FAR, FRR, and the decision threshold in more detail. The point at which the two lines cross is the EER. It is worth noting the large variation in the threshold at which the EER is achieved. Models for users 21, 22, 25, 26, 61, 62, 82, 85, and 86 reach an equal error rate at or above a threshold of 0.8, which indicates a high degree of confidence in the prediction of positive class. These models have a lower EER than models with lower threshold values. To illustrate this, table 6.3, sorted by EER, is given below.

User ID	Equal Error Rate	Decision threshold
22	0.026	0.95
82	0.026	0.95
86	0.028	0.9
25	0.037	0.85
62	0.038	0.85
26	0.059	0.8
61	0.066	0.8
41	0.081	0.35
21	0.124	0.95
85	0.131	0.75
42	0.163	0.45
60	0.176	0.05
40	0.2	0.05
83	0.22	0.05
23	0.245	0.4
24	0.27	0.05
81	0.392	0.05

TABLE 6.4: Sorted per model equal error rate.

An outlier in this trend is model 41, which achieves an equal error rate of 8.1% at a 0.35 decision threshold, outperforming models with much higher decision thresholds. Outliers like this suggest that a better approach than choosing one decision threshold could be to determine them on a per-model basis. Such a threshold could be chosen once, on some portion of the user training data through cross-validation, although such an approach would lengthen the training process. Alternatively, such a value could be adjusted dynamically, based on how often a user fails to be authenticated using the model. Such considerations could be an area of further research as they fall outside the scope of this project. Additionally, because a global decision results in poor performance for otherwise well-performing models, it can be concluded that the average per-model EER is a better metric to use than the global EER and, as such, it will be used in the rest of this chapter. Another interesting example is model 21, for which the FRR is almost constant, at a value of around 12%. This indicates that all examples are recognized with a high degree of confidence, except for those 12%, which may indicate that the user changes writing styles for part of the final input or that the training data failed to capture some characteristic of the writing style.

6.1.4 Confusion matrix for all users

To evaluate each model's performance, each model was tested using the input of every user. The results of this test are shown in figure 6.3. The values in this matrix represent the percentage of examples classified as positive. It is important to note that the values in this matrix have a different interpretation depending on their position. The values on the diagonal represent the percentage of correctly classified positive examples (True Positives) – the recall of the model. In an ideal classifier, these values would equal 100. The values outside the diagonal represent the

percentage of incorrectly classified negative examples (False Positives). In an ideal classifier, these values would equal 0.

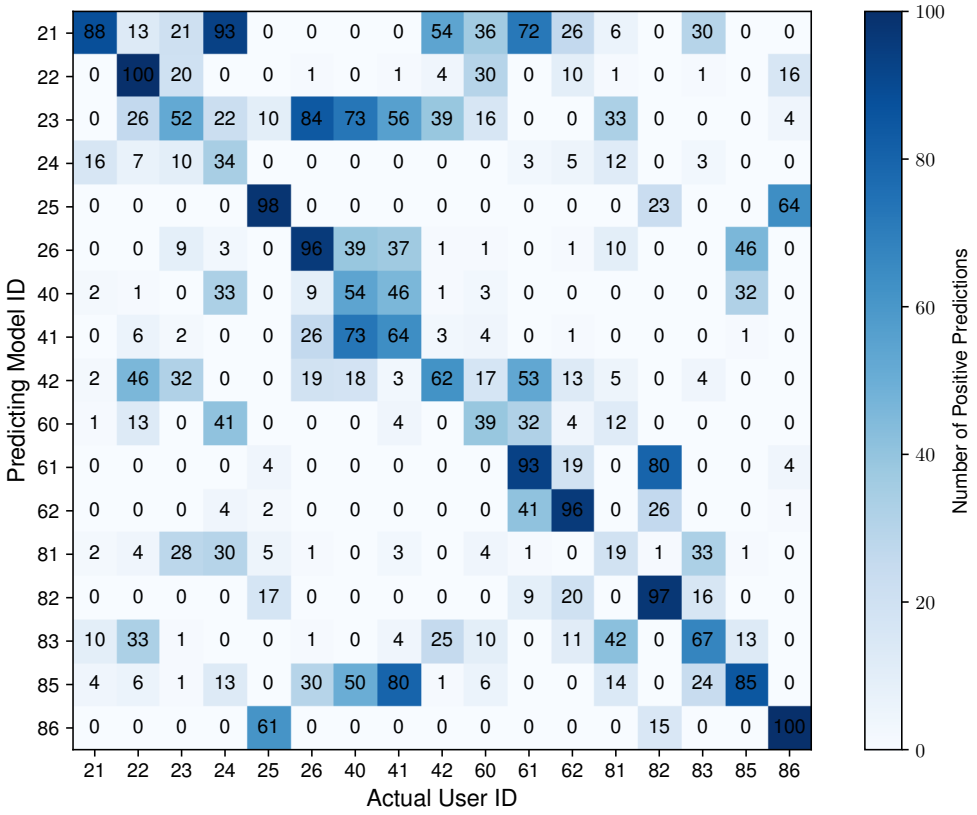


FIGURE 6.3: Matrix showing the percentage of examples classified as positive for all model–user pairs.

Figure 6.3 illustrates the different types of errors made by individual models. Specifically, it highlights which users share similarities, leading to confusion in model predictions. For example, models for users 25 and 86 perform very well for all inputs except for each other. However, this is not true for all models, as the model for user 61 makes mistakes when classifying the input for user 82. The inverse is not observed, as the model for user 82 does not misclassify user 61's data at a higher rate than others.

6.2 Input features

This section compares the impact of input encodings on the final model performance, as measured by the average FAR and FRR, as well as the average per-model EER value. The impact of these changes was measured by changing only a single hyperparameter, while all others remained the same, as shown in table 6.1. The results shown below compare performance on the testing dataset, allowing these values can be contrasted with what was discussed in the previous section. As mentioned previously, this is not how these features were selected.

6.2.1 Input sequence length

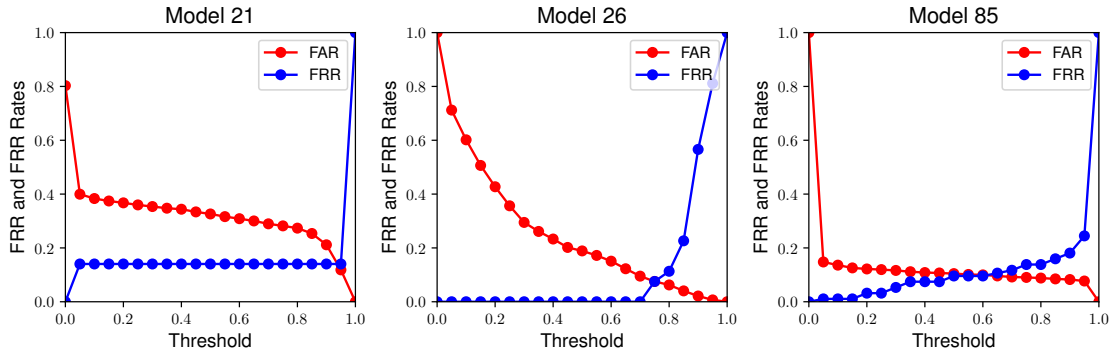
The theoretical impact of input length, as discussed in the section on graph creation, suggests that an optimal sequence length exists. Making the sequence shorter would result in graphs that do not have enough structure, for example, a chain of nodes or not enough edge data to make the

correct prediction possible. Making the sequence longer would result in a graph that is too dense, or it would cause the averages calculated per node to become too aggregated.

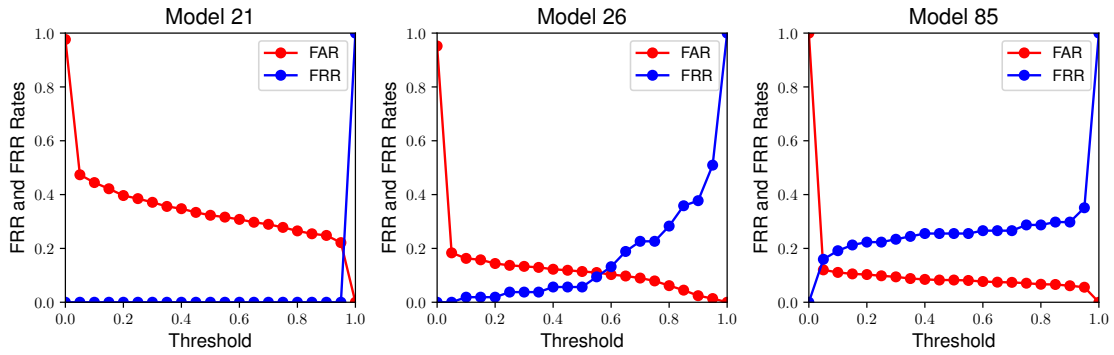
Input sequence length	FAR	FRR	EER
30	0.101	0.332	0.180
40	0.092	0.264	0.133
50	0.098	0.232	0.135
60	0.079	0.307	0.158
70	0.083	0.391	0.176

TABLE 6.5: Comparison of performance with different lengths of input sequence.

Table 6.5 partially supports the theoretical expectation, as models trained on input sequences of length 30, 60 and 70 achieve much higher EER scores than those trained with lengths of 40 and 50. There is a very small difference in the performance of these model collections on the aggregated metrics. With the difference in the EER scores being this low, it is unclear which collection of models performed better on the testing dataset. However, interesting differences appear when compared on a per-model basis, as the changes in models were not uniform. The impact of three selected models is shown in figure 6.4.



(A) Plots for input of length 40



(B) Plots for input of length 50

FIGURE 6.4: Comparison of selected FAR and FRR plots for different input lengths

The model for user 21 shows a clear improvement as the unrecognized examples discussed in subsection 6.1.3 are now correctly classified. Conversely, the model for user 26 now classifies its positive examples with less confidence and showed an increase in EER from 5.9% to 10.2%. For model 85 the change in length did not impact the EER. However, it moved the decision threshold significantly. What these comparisons demonstrate is that it is hard to reason about

model performance across different input sequence lengths and a small change has a large impact even on a per-model basis.

6.2.2 Character encoding

The results of comparing methods of encoding character information in nodes are shown in table 6.6.

Character encoding method	FAR	FRR	EER
Normalized Base Letter Encoding	0.092	0.264	0.133
Compact Alphabet Encoding	0.082	0.339	0.161
Full One-Hot Encoding	0.070	0.397	0.197

Table 6.6: Comparison of performance with different edge data encoding methods.

Table 6.6 shows that more aggregative methods outperform those that are less compact. The lower FAR might suggest that these models try to capture more complex features and thus overfit the data.

6.2.3 Edge data encoding

Edge encoding method	FAR	FRR	EER
Two dimensional vector of values per node	0.069	0.666	0.395
Two values per node	0.092	0.264	0.133

Table 6.7: Comparison of performance with different edge data encoding methods.

Table 6.8 shows the superiority of the *Two values per node* encoding method. While the average FAR remains comparable, FRR, and thus the EER, differ by a large amount. This might be caused by the same reasons as the difference between character encoding methods. The method with less aggregation performs worse, as the model might not have seen all possible two-letter combinations and failed to generalize.

6.3 Class imbalance

Another aspect of experimentation was class balance in the training data. The baseline model was trained on a dataset with twice as many positive examples as negative examples. A bigger proportion of positive examples was selected experimentally, as it resulted in models that performed better in recognizing the positive class.
The negative examples were sampled uniformly from all other users, with an offset such that the negative examples for a user were taken from the whole input text. This approach is effective for the number of users in the current dataset. However, as the number of users grows, and the length of the input sequence stays fixed, at some point each model needs to be trained only on a subset of the negative class input texts. At this point, some models may fail to generalize to unseen negative examples. While solving such a problem lies outside the scope of this project, the impact of using more negative examples in the training process was measured and the results are presented in table 6.8.

Positive to Negative Ratio	FAR	FRR	EER
2:1	0.092	0.264	0.133
1:1	0.080	0.340	0.177
1:2	0.033	0.506	0.220

TABLE 6.8: Impact of positive to negative ratio on model collection performance.

A higher percentage of negative examples were classified correctly, as indicated by the lower FAR; however, this came at the expense of a significant drop in the FRR, with almost half of the positive examples being misclassified in the case of *1:2* class balance. This means that the collection of models generalized poorly to unseen positive examples, to a degree that would be unacceptable. This can also be seen when comparing the average recall scores across these two collections of models, which dropped from *0.7357* to *0.692* and *0.494* when the number of negative examples were doubled.

6.4 Discussion

The findings presented in the study can provide an insight into the developed model. It is evident that the model performed significantly worse for certain users compared to others. There could be many reasons for such behavior. One important fact is that some users wrote using more than one language at the time or wrote in a different language from the rest. Most users in the study wrote their training and testing samples in Polish only, but users such as 22, 24 and 42 did not. Of these, 24 and 42 wrote both in English and Polish. Of those, user 24 seems especially interesting, given that they wrote most of the training data in English, but their testing sample was written in Polish exclusively.

It should be reiterated that for this model it was especially important for the users to have the lowest possible False Acceptance Rate (FAR). In a theoretic system using such a model, a user being able to pass for another user is a more dangerous situation, than a user not being able to authenticate themselves. The second problem may be mitigated by user being authenticated with longer texts, or with multiple tries being averaged.

Some users are seemingly more similar to other users. For some especially problematic users, such as user 24, the likelihood of them being classified as user 21 far outweights the likelihood of them being classified as themselves. Interestingly, the reverse was not nearly as likely: user 21 was classified well and only rarely could be classifed as 24. This could be possibly explained by user 24 having their testing sample in a different language from the training data. Model seemed to classify this user as another Polish user because of that. User 22, who used English exclusively, was not problematic, thus giving some credence to this hypothesis.

Another problematic user was user 40, who wrote letter "y" for an extended period of time, thus invalidating much of the sample gathered, and also wrote in capital letters only for the last 300 characters. This user had relatively high False Rejection Rate of 0.459, possibly caused by the worse training data.

Users switching hands, writing some paragraphs with one hand and some with both or switching position such that their hand was performing differently (e.g., by lying on their side) could not be accounted for in this analysis - only stark changes in user's posture could be detected via the accelerometer, but these could also be explained by natural movements, which should not have any effect on the user's hand position relative to the screen. This problem is typical of mobile devices and should not exist in the computer and keyboard setting.

6.4.1 Possible problems and further studies

As previously stated, there could be many possible causes for such behavior. Many of such hypotheses could be tested, but a significantly larger number of participants would be required for such a study. For a truly significant test of the approach, each participant would also need to provide more data in the sample. This could prove challenging, since, according participant of this study, providing a sample as small as 1500 characters was already tiresome. Some users resulted to writing semantic noise, such as repeating one or two letter for an extended period of time, which made the data gathered from that part of a sample essentially useless. In some studies [3], this is somewhat mitigated by users typing some sentence or password repeatedly, but this study concerned itself with free-form writing. In other studies [10], datasets with larger sample sizes were used, including the Clarkson II dataset with 103 users contributing 12.9 million keystrokes over 2.5 years, and the Buffalo dataset with 157 participants providing an average of 17,000 keystrokes across three sessions.

Possible way of gathering better quality data from more motivated participants could be construction of a chatbot of some sort, similar to a significantly simplified version of Replika [28] or Cleverbot [29], complete with a dedicated mobile application and a keyboard designed specifically for the app – long use of the application would make users write more naturally over time, even on a foreign keyboard. The chatbot would entice the users to use the application more often, thus gathering more data. This approach, however, could lead to some security and privacy concerns. A limited approach is also possibly viable, with a simple chatGPT model being used as a conversational companion for the user only for a short duration, such as a 30min session.

There may be some fundamental problems with the data being gathered, as well. It may be that a limited set of features employed in this study is insufficient to perform as well as state of the art models, such as [10]. That model achieved an FAR of 2.83% and 5.31%, an FRR of 1.89% and 6.61%, and an EER of 2.36% and 5.97% for the Buffalo and Clarkson II datasets, respectively. In comparison, the model used in this study resulted in an FAR of 9.2%, an FRR of 26.4%, and an EER of 13.3%. The observed performance differences may result from the dataset size and the available feature set. Expanding the feature set to include accelerometer data could improve performance but introduces several challenges. Collecting a significantly larger dataset would be necessary to cover various conditions, such as typing while sitting, lying down, or standing. This would help ensure that the model generalizes correctly across different scenarios.

True evaluation of the method is thus possible only with a larger sample size and text lengths. Gathering users proficient in different languages could also be helpful for analyzing the effects of two different languages being mixed in the model and possible adverse effects of such an approach. Different sets a letters used and different letter frequencies may lead to atypical data and problems in user recognition.

Chapter 7

Conlusion

In summary, the approach used in this project and study needs to be further studied to be effectively evaluated. The effects of such an approach seem promising, but are currently behind the state-of-the-art models. A larger study would also be a source of validation or rebutal to many hypotheses posed in the discussion chapter about the possible sources of error in model's performance on some specific users. Larger dataset specifically could also benefit the scientific community at large, if the users would agree to releasing it under an open source licence.

It is worth highlighting that for many users, the model performed well. Further refinement or combining the model with other neural network types, feature engineering or modification to data gathering process could also lead to improved results.

The research team considers the project successful in demonstrating that the proposed approach has merit and warrants further exploration.

Bibliography

[1] Ahmed Sherif. Number of biometric-enabled smartphones in north america and western europe from 2021 to 2024. [on-line] `https://www.statista.com/statistics/1226088/north-america-western-europe-biometric-enabled-phones/`, 2025. Accessed: 2025-01-11.

[2] Configure windows hello in windows. [on-line] `https://support.microsoft.com/en-us/windows/configure-windows-hello-dae28983-8242-bb2a-d3d1-87c9d265a5f0`, 2025. Accessed: 2025-01-11.

[3] Hussien AbdelRaouf, Samia Allaoua Chelloug, Ammar Muthanna, Noura Semary, Khalid Amin, and Mina Ibrahim. Efficient convolutional neural network-based keystroke dynamics for boosting user authentication. *Sensors*, 23(10), 2023.

[4] Types of biometrics. [on-line] `https://www.biometricsinstitute.org/what-is-biometrics/types-of-biometrics/`, 2025. Accessed: 2025-01-11.

[5] Ml kit: Face detection. [on-line] `https://developers.google.com/ml-kit/vision/face-detection`, 2025. Accessed: 2025-01-11.

[6] Forensic biometrics. [on-line] `https://www.nist.gov/forensic-biometrics`, 2025. Accessed: 2025-01-11.

[7] Atharva Sharma, Martin Jureček, and Mark Stamp. Keystroke dynamics for user identification. 2023.

[8] R. Joyce and G. Gupta. Identity authentication based on keystroke latencies. *Communications of the ACM*, 33:168–176, 1990.

[9] Fabian Monrose and Aviel Rubin. Authentication via keystroke dynamics. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, CCS '97, pages 48–56, New York, NY, USA, 1997. Association for Computing Machinery.

[10] Xiaofeng Lu, Shengfei Zhang, Pan Hui, and Pietro Lio. Continuous authentication by free-text keystroke based on cnn and rnn. *Computers & Security*, 96:101861, 2020.

[11] Hayreddin Çeker and Shambhu Upadhyaya. Sensitivity analysis in keystroke dynamics using convolutional neural networks. In *2017 IEEE Workshop on Information Forensics and Security (WIFS)*, pages 1–6, 2017.

[12] Keyboard commands overview. [on-line] `https://developer.android.com/develop/ui/views/touch-and-input/keyboard-input/commands`, 2025. Accessed: 2025-01-11.

[13] C. Wang, H. Tang, H. Y. Zhu, J. H. Zheng, and C. J. Jiang. Behavioral authentication for security and safety. *Security and Safety*, 3:2024003, 2024.

[14] Graph (discrete mathematics). [on-line] `https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)`, 2025. Accessed: 2025-01-11.

[15] Jure et al. Leskovec. Cs224w: Machine learning with graphs. [on-line] `https://web.stanford.edu/class/cs224w/index.html`, 2024.

[16] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A gentle introduction to graph neural networks. *Distill*, 2021. https://distill.pub/2021/gnn-intro.

[17] Determining the optimal number of gat and gcn layers for node classification in graph neural networks.

[18] Issa Traore. *Continuous Authentication Using Biometrics: Data, Models, and Metrics: Data, Models, and Metrics*. Igi Global, 2011.

[19] Fastapi. [on-line] `https://fastapi.tiangolo.com`. Accessed: 2025-01-11.

[20] Android distribution chart. [on-line] `https://composables.com/android-distribution-chart`. Accessed: 2025-01-11.

[21] Mobile operating system market share in poland. [on-line] `https://gs.statcounter.com/os-market-share/mobile/poland`. Accessed: 2025-01-11.

[22] Jetpack compose: Modern toolkit for native ui. [on-line] `https://developer.android.com/compose`. Accessed: 2025-01-11.

[23] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. On-device neural net inference with mobile gpus. In *Efficient Deep Learning for Computer Vision CVPR 2019 (ECV2019)*, 2019.

[24] Jiaxuan You, Jonathan Gomes-Selman, Rex Ying, and Jure Leskovec. Identity-aware graph neural networks. 2021.

[25] Pierre Baldi and Peter J Sadowski. Understanding dropout. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[26] Pál András Papp, Karolis Martinkus, Lukas Faber, and Roger Wattenhofer. Dropgnn: Random dropouts increase the expressiveness of graph neural networks. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 21997–22009. Curran Associates, Inc., 2021.

[27] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, and Evgeni et al. Burovski. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024.

[28] Replika. [on-line] `https://replika.com/`. Accessed: 2025-01-11.

[29] Cleverbot. [on-line] `https://www.cleverbot.com/`. Accessed: 2025-01-11.