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, majority of smartphones had biometric sensors, such as fingerprint readers[17]. Many computers can also provide biometric authentication via face recognition, if connected to a web camera, e.g. via Windows Hello on Windows 10 or 11 [5]. 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 the user by their biometric data contained mostly within the keystroke data. The users in the study, which was 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 used in this thesis mostly concerned the two following groups: studies of keystroke data models and their effectiveness and the specialist literature on the topic of 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 a brief overview of the project, explaining its components and the relationships between them. It includes the following sections: Section 4.1 consists of the description of the server. Section 4.2 describes the mobile application used for user data collection and model validation. Section 4.3 contains a description of the Neural Network model used for user recognition, complete with the hyperparameters used in model training and validation. Section 4.4 describes the feature selection used for a model. Section 4.5 discusses the metrics used in the model testing on data gathered from users and the testing results. Section 4.6 concerns the user testing with the help of study participants and the study results. Chapter 5 is a conclusion to the thesis.

Work on this project was divided as follows: Jakub Grabowski created the mobile application, set up and coordinated the project, and researched biometrics for his thesis paper. Filip Kozłowski created the server and integrated the GNN model with it. He also planned and implemented communication between the server and the application. Krzysztof Matyla helped in creating the mobile application, provided testing for various parts of the project, and coordinated user testing. Igor Warszawski planned and implemented the GNN model used on the server. He also tested and validated the results, together with Filip Kozłowski.

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. [8]

Uniqueness of one's body is well known in biology. Features that may be used for person's identification are for example (FIX SOURCE: https://www.biometricsinstitute.org/what-is-biometrics/types-of-biometrics/):

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. (FIX SOURCE: https://developers.google.com/ml-kit/vision/face-detection, https://support.apple.com/en-us/102381)
4. **Fingerprints and finger shape** – fingerprints are widely used in forensics (FIX SOURCE: https://www.nist.gov/forensic-biometrics) and in digital scanners on mobile devices and laptops.

Other, less popular ways of identifying a person are for example: ear shape, gait, hand shape, heartbeat, keystroke dynamics, signatures, vein scans and voice recognition.

2.1 Keystroke Dynamics

One possible way to extract 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 [16]:

1. **Hold Time** – time between key press and release

2. **Down-Down Time** – time between first key press and second key press
3. **Up-Up Time** – time between first key release and second key release
4. **Up-Down Time** – time between first key release and second key press
5. **Down-Up Time** – time between first key press and second key release.

It can therefore be said that keystroke dynamics focus mostly on identifying 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 idea was already being tested in 1990 by Joyce and Gupta[10]. By 1997, clustering methods were already being used in experiments on user data on a small scale (42 profiles) by Monrose et al. [13]. Algorithms used for such data evolved after that point and the raise in popularity of neural networks.

Lu et al. [12] used a combined CNN+RNN approach to obtain the results of ERR of 2.36% on Buffalo dataset and 5.97% on Clarkson II datasets. Çeker and Upadhyaya [20] used 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. This method had an ERR of 2.3%.

A Convolutional Neural Network (CNN) has a fixed node ordering and operates on a grid – some input must firstly be mapped into 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 (like in Çeker and Upadhyaya) or with the users being free to type anything within some length constraints (like in Lu et al.). 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 [7]. 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.

The keystroke identification can also rely on other data gathered from the keyboard, such as the specifics of letters used, their average frequencies, most common connections between the letter or other statistics [19]. 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 $v_1, v_2 \in V$ [6]. 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. [11]

Some problems relating to such data can be solved using Convolutional Neural Networks – this can also be the case for keystroke dynamics data, such as with Lu et al. [12] or Sharma et al. [16]. 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. [15]. 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

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

  - $m_u^{(l)}$ represents the message computed for node u at layer l.
  - $\mathrm{MSG}^{(l)}$ 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

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

  - $h_v^{(l)}$ 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:

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

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

The optimal number of layers in GCN is usually small (FIX SOURCE). 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 (FIX SOURCE). 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 (FIX SOURCE). 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 in GNN

Supervised learning on graphs can be achieved by labeling either nodes, edges or whole graphs. 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 [11]. For nodes, predictions can be made directly using node embeddings – this can be done by using a classification layer, like a dense layer [15]. 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 [11].

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:

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

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 positives and false negatives, which from the perspective of this project differ in importance. False positives are considered more influential, as these kinds 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:

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

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

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

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. Citation needed. 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:

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

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

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

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

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

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, State of the Art models can actually perform well (FIXSOURCE) on such data. 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 (FIXSOURCE). Because of that, an important part of the project was a study of results and data gathered, which is presented in chapter 3.4 and 3.5.

TODO: find statistics and add them to sources

4.1 Use cases

TODO: add use cases and a short paragraph explaining reasoning behind the project.

4.2 Server structure and communication with the application

The server is written in Python and implemented using FastAPI [2], 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, acel_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 [1], 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 [4] , significantly more people in Poland, where the researchers are based in, use Android devices).

Technology used in the mobile application itself was Jetpack Compose, which is quoted by Google to be "Android's recommended modern toolkit for building native UI" [3]. Language used was Kotlin. Persisence was achieved by using Android Room, which provided an abstraction layer over SQLite database, which was 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`): Currently unused but useful for future developing of the project.

```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, followed by the presentation of the recognition rate percentage on a circular progress bar and a message indicating whether the model recognised the user or not.

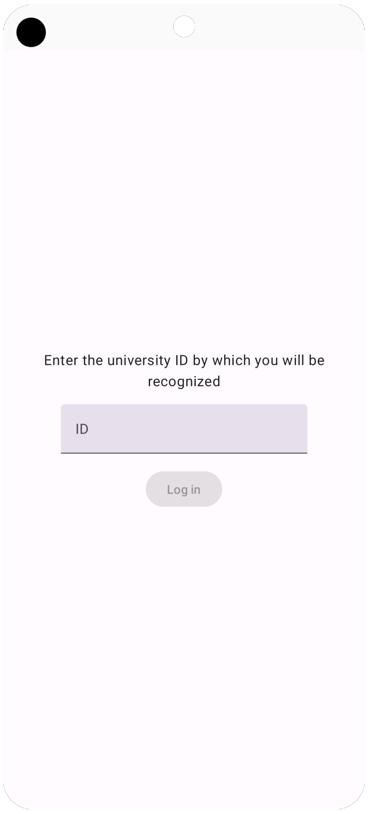

FIGURE 4.4: Login screen

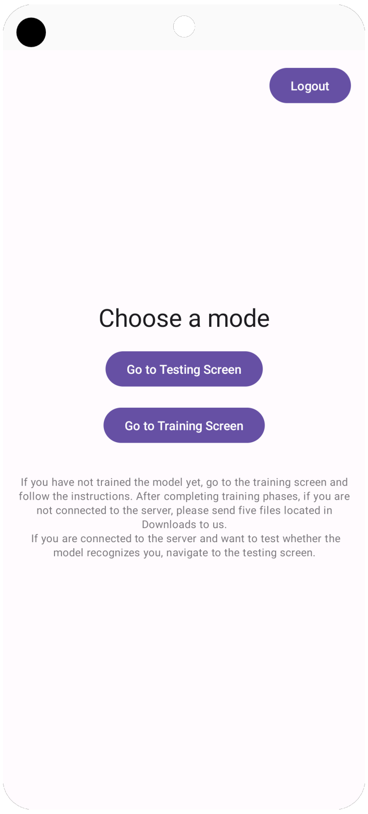

FIGURE 4.5: Home screen

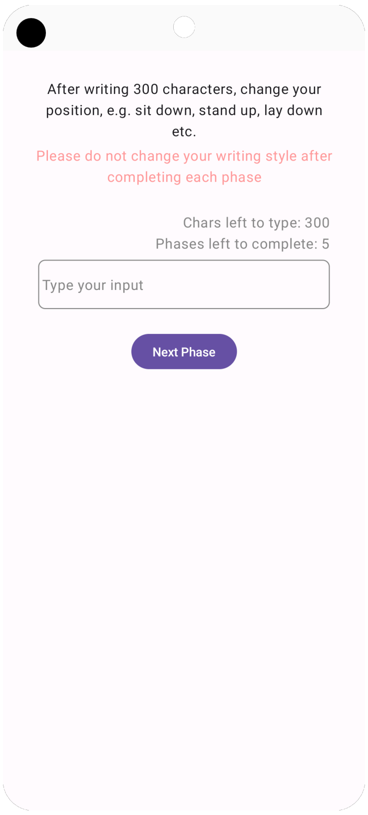

FIGURE 4.6: Training screen

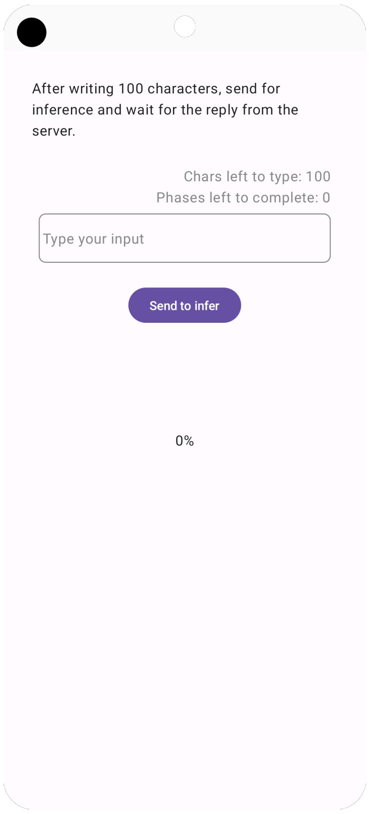

FIGURE 4.7: Testing screen

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` 4.8 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 4.9). 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.

```
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 4.8: 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 4.9: 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 mechanism is used to provide feedback to the user.

4.3.5 Testing screen

Testing screen... TODO Add screenshot and write sth about testing the model, how the % work and so on

Chapter 5

GCN Model

In this project, the goal was to use the graph networks that can naturally arise from keystroke data to – on a graph level – try to infere the users identity. A key characteristic of a project was that the use of any keyboard already installed on the user's mobile phone causes some problems 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. what data was actually mapped?

Moreover, this project focused on creating an collection of models, one model for each user that performs binary classification rather than one large model for multiclass classification. This decision was made for several reasons. Firstly, such scheme allows for models to be trained on demand, as soon as a new user provides all the training data to the mobile application. Secondly, new users do not force the whole model to be retrained, 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. Lastly, the one user per model scheme allows for inference to take place locally, on the target user's device. This would remove 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, such as TODO citation needed, the complexity of such solution was deemed to great and outside the scope of this project.

5.1 Choosing features for Neural Network model

TODO Some introdutions

5.1.1 Data exploration

All data analysis on the input goes here

5.1.2 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. TODO: add example graph visualization here 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 for that edge, which will be aggregated in later stages, to a form suitable for the GCN model. It would also be possible to model such pairs using a multiedge graph, as such models have been shown to perform well in other domains TODO zacytuj ”multi-edge graph for convolutional networks for power systems. However, we did not consider this approach.

TODO maybe visualization The structure of the resulting graph depends highly on the length of the input sequence. A shorter sequence produces smaller and sparser graphs, while a longer input sequences result in graphs with more nodes and edges.

Citation needed - some RNN paper that sequences of keys are important (something in Lu 2020?)

5.1.3 Edge atributes encoding

In order for the graph to be a valid input for a GCN network, edge attributes need to be converted into node features. We found two ways to encode aggregate 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. Add two-dimensional vector of values, of size [number of allowed characters, 2]. Each key that can be found 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 clear difference between these two approaches is the level of aggregation. Method 1 aggregates all the edge information into 2 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 complexity of the model.

5.1.4 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 the purpose of encoding, 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 – *, #, @, :, ;, ’, ”, (,), [,], {, }, <, >, /, \, —, &, %, $, ˆ, ˜, _, +, -, =.
- Others – all other symbols.

Similarly, there is more than one way to encode key information into node features. We considered three methods, all being variations on a one-hot encoding of keys:

1. Classic one-hot representation. Each unique cased character is mapped to different column in the vector, except for characters in the *Others* group, which map to one extra column.

2. Small alphabet representation. One hot encoding for cased *Letters* and *Special characters*. *Numbers* map to one column in the vector, *Symbols* and *Others* map to one column in the vector.

3. Base letter representation. All *Letters* are converted to lower case, diacritical marks are removed. These transformed letters are encoded in a one-hot vector. *Numbers* map to one column in the vector, *Symbols* and *Others* map to one column in the vector. Two additional columns are added to the input, one indicates whether the original character was a capitalized letter, the second whether it had a diacritical mark.

Citation needed: That paper that said node ids are nice. Again as before, these methods differ by 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. Cication found that such node identifiers helped the model to learn certain structures in the data. However, Slajdy z stanfordu ze tylko jak jest ograniczona liczba liter notes, that providing node identifiers as input features work well only for a small and known set of possible input nodes. While this requirement appears to hold true for this specific task, we found this is not the case. Many letters, which appear to be common, in fact do not appear in our dataset. TODO Which chars never appear. This means that the behavior of the models would be unpredictable for nodes with such identifiers. Some characters, for example EXMAPLEEEE like '(', appear only once, leading models to overfit and generalize poorly. Method 3 was specifically designed to deal with the low number of appearances of certain characters, especially uppercase letters. However, the extra column still allows for distinguishing between lower and uppercase variants of the same letter, which is important as the time to input these symbol differs greatly. Avg time for uppercase and lowercase letters Moreover, method 3 directly exposes the use of diacritical marks, which, similarly to uppercase letter, take longer to input Gimme DATAAAAA. Ładne zdanie które mówi wooow, diacritical marks are nice in polish, maybe data

ADD histogram of how many times each character appears
ADD Which chars never appear

5.1.5 Feature selection – accelerometer data

To improve the possible performance of the model, and to make further use of the capabilities of the mobile platform, we considered using accelerometer data as an input feature for the model. This portion of the input data comprised of three values, a measurement of the acceleration in the x, y and z plane 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 reaching low loss values, they failed to generalize, performing worse on validation and test datasets.

TODO, some accelerometer data here For this reason, accelerometer data was not used for training and evaluating models discussed later.

5.2 Graph Convolutional Network for user recognition

IW: Przesunąlem to na góre, wydaje mi sie ze to tutaj nie miało sensu moze mozemy tutaj dodac jakas teorie o GCN konkretnie

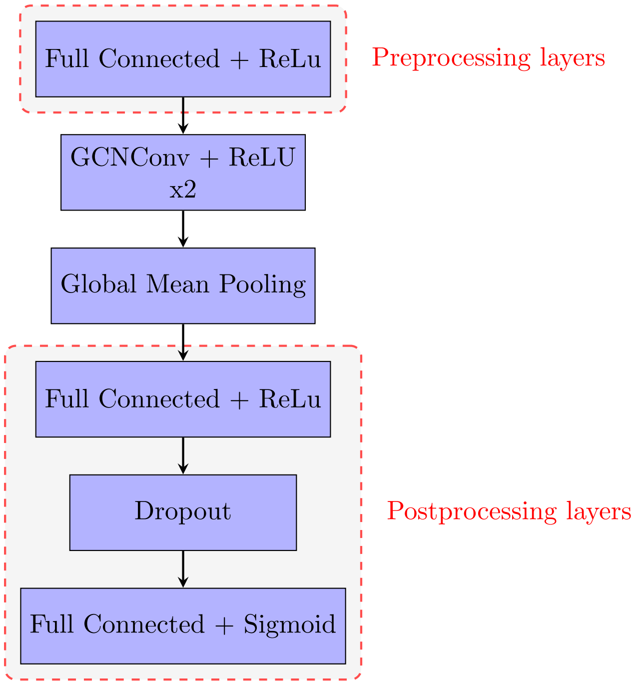

FIGURE 5.1: Model architecture

5.3 Metrics aggregation

Discuss how we aggregate metric across user models, how eer is calculated with a per model threshhold on sigmoid and then average. Results of confusion matrix are shown with a single threshold that is chosen for a lower average eer – maybe choose something else ?

5.4 Architecture

The basis of a model for graph level prediction comprises of a graph convolutional layers, a pooling function to aggregate node information and a classification head, which outputs a single value, that is than passed to a sigmoid function. Cite something The architecture used in this project is pictured below.

Add some text to this drawing

GCN + ReLU

The first three layers of the model, pictured in figure 5.1, are graph convolutional layers or GCNConv. The theory behind these layers was extensively discussed in chapter 3. After each layer a ReLU activation function was applied.

Global Mean Pooling

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

Preprocessing Layers

Considering the complexity of the input features, specifically the relationship between parts of the features used to represent characters discussed in 5.1.4, 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 [11].

Postprocessing Layers

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

Dropout

A dropout layer was added between the two fully connected layers, which randomly disables some of the neurons during the training process. Such layers have been shown to create more robust models, that generalize better by Baldi et al [9]. Dropout layers have also been shown to improve expressiveness in GNN's by Papp et al [14].

5.5 Training and fine-tuning

5.5.1 Data division

For the purpose of model validation, we used 5-fold cross-validation, thus splitting the training set described in TODOOO Numer sekcji into 5 parts and training on 4 of those parts. This number of folds corresponds to the number of 300-character blocks, that the users were asked to input. Such splits validate that the trained model is able to generalize to a part of the input that might have been written at a different time and style. 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.

5.5.2 Training parameters

Loss function

Binary Cross Entropy was chosen as the loss function that would be used in training, which is a common loss function 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 to operations are fused into one step in the implementation, by using $BCEWithLogitsLoss$, which is more numerically stable.

Optimizer

Adam was selected as the optimizer for updating model weights, which adapts the learning rate for each parameter. A learning rate of 0.001 was chosen empirically.

5.5.3 Tuning hyperparameters

The architecture described above allows of large number of hyperparameter choices, such as the size of preprocessing, convolutional and postprocessing layers. Furthermore, the variable length on input sequences and multiple possible feature encoding, mean that a full search over the hyperparameter space is impossible. For the scope of this project the biggest emphasis was placed on finding the optimal input features and input sequence length. Furthermore, the effect of class imbalance on the training process of the collection of models was also explored.

TODO: Nie wiem czy tutaj czy moze w sekcji z wynikami. Moge tu napisać ale chyba to nie ma sensu, lepiej jest chyba omówić wpływ parametru na wynik. Z sekcji Testing model on users bym zrobił chapter

Chapter 6

Results

This chapter is dedicated to the results of model testing on users, as well as the inpact of features and hyperparameter choices on its performance.

Methodology

Word this nicely The results below were calcutaled as an average of each models perfomance on a sequance of characters equal to the length on which the model was trained. This means that a some models perfomed inference on sequences of lenght 20, while others 40. While it was possible to compare scores based on an equal input lenght, it would require the setting another threshold, on how many of the input subsequnces need to be classified possitivly for the whole sequance to be classified as possitive. Such threshold was chosen for the aplication, however, we believe that adding another parameter to these results would only make them less interpretable.

Tutaj macierz pomyłem między wszyskimi modelami, ostateczne wyniki, najlepsze parametry etc Dezycja czy w sekcji – moze byc ale jak ją nazwać

Confusion matrix for all users

Roc curves

Points in roc space

Equal error rates

EER for each user, the threshold neccessary

6.1 Input features

This section will compare the impact of input encodings on the final model performance.

6.1.1 Input sequance lenght

6.1.2 Egde data encoding

6.1.3 Character encoding

6.1.4 Accelerometer Data

TLDR doesnt work, comparison for 2 users only - best performing on normal input Short discussion

6.2 Training and model hyperparameters

6.2.1 Class imbalance

Discuss how positive/negative example ratio had an impact on precission/recall. Maybe note that neg examples were sampled with an offset

6.2.2 Model size

Num of conv layer, layer size etc bigger wasnt better

6.3 Testing model on users

6.3.1 TODO title

TODO: What if we did classify the user on all their input (100 chars). Results would be a n users by n users matrix of 1's and 0's

6.3.2 Multiclass model

Maybe

6.3.3 Cross-smartphone user validation

TODO: what happens if two users train on smartphones that are not their own? What happens, if they cross-use their original model on another phone?

6.4 Discussion

TODO: discuss the findings.

Chapter 7

Conlusion

TODO for JG: not needed now, will write after the user tests are done.

Bibliography

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

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

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

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

[5] Configure windows hello in windows, 2025. Accessed: 2025-01-11.

[6] Graph (discrete mathematics), 2025. Accessed: 2025-01-11.

[7] Keyboard commands overview, 2025. Accessed: 2025-01-11.

[8] 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.

[9] 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.

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

[11] Jure Leskovec and other instructors. Cs224w: Machine learning with graphs. Stanford University, Online Course Materials, 2024.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] Ahmed Sherif. Number of biometric-enabled smartphones in north america and western europe from 2021 to 2024, 2025. Accessed: 2025-01-11.

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

[19] 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.

[20] 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.