ML-Project: Touch-Keyboard Sidechannel

This report gives an overview of a machine learning approach to using inertial time series data to recover touch inputs. It starts with an introduction to the problem and the data used in section 1. Data and features are described in detail in section 2. ML methods and loss functions used are described in section 3. The final section, 4, compares the models and discusses the results.

1 Problem

Modern smartphones are equipped with a multitude of high quality sensors. While these provide significant functionality for the end user, they also open novel attack vectors. For example, when typing on a touch keyboard, the phone moves slightly under each tap. This movement is picked up by a smartphone's acceleration and rotation sensors. Using raw motion data, an attacker can infer keystrokes and text strings. Therefore, an app running in the background performing statistical analysis on raw data from acceleration and rotation sensors could potentially act as a keylogger and record any confidential messages or passwords. [1]

1.1 Data aggregation

While many papers have already dealt with this topic [2–4] there is no public dataset available for our particular use case. As such, we programmed a simple Android app to record data for our model. The app provides a text field and records acceleration and rotation data from onboard sensors on input. When a key is pressed, acceleration and rotation data in a 1000ms window before and after input as well as a label indicating the value of the pressed key, are stored in a single .csv file. These files are later concatenated into a full dataset. We have generated ~1400 sets of acceleration and rotation datapoints. Below we have a visualization of datapoints with the label "1":

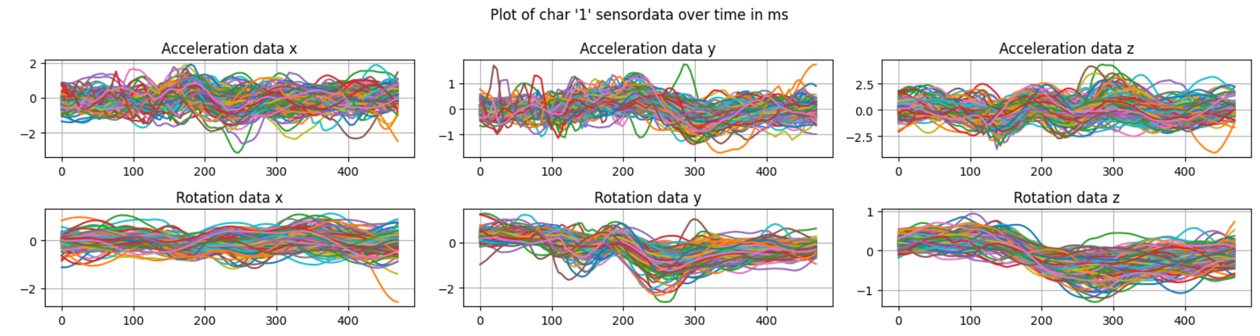


Figure 1: Recorded Data when pressing '1'

1.2 Data

The data is recorded with a single Android device, which is able to record rotation data at a rate of 422 Hz and acceleration data at a rate of 211 Hz. For the purpose of this paper, only individual digits on a number pad, such as would be used to unlock a device, were accepted as valid input. Each dataset, labelled with the numeric value of the corresponding keypress, comprises the following information:

Timestamp	Sensortype (ACCEL / ROT)	X_{Value}	Y_{Value}	Z_{Value}	Label

Timestamp is in ns from start of the dataset capture window, X/Y/Z recorded as float w/ 6 digit precision. For the acceleration, this value represents the force of acceleration along the given axis in units of $\frac{m}{s^2}$, captured from the device's virtual accelerometer (which removes acceleration due to gravity from the raw accelerometer data). For the rotation, this represents the phone's rate of rotation around a given axis in $\frac{rad}{s}$.

1.3 Machine learning task

We are performing a supervised learning task in order to classify button presses on a numeric keypad given only continuous sensor data. Our labelling system is a single numeric value from 0-9. There is one problem with the data: Presses on adjacent buttons might result in nearly identical sample data. For example, the buttons '1' and '4' are next to each other and in edge-cases yield very similar data, which makes reliable classification very hard (see figure 4 and 5)

2 Feature selection

2.1 Preprocessing

Timestamps are reformatted from raw epochal Android time data to a ms scale starting from 0, representing the contextual timescale of a given dataset, and, as the rotation and acceleration data are sampled at 422Hz and 211Hz respectively, we downsample the rotation data using linear interpolation in order to have a cohesive dataset. Finally, leading and trailing data points are dropped to get a coherent, time-synchronous dataset describing the button press $\pm 250ms$:

Timestamp (ms)	X_{Accel}	Y_{Accel}	Z_{Accel}	X_{Rot}	Y_{Rot}	Z_{Rot}

2.2 Feature Extraction

Multivariate time series data pose several challenges in extracting features that preserve the relationships between correlated waveforms.[5]. We initially attempted to perform low-component PCA analysis on the preprocessed data and modelling the resultant dimension-reduced data as a feature set, but our results were not particularly impressive. Instead, we made use of a preexisting Python library[6] for time series feature extraction over our dataset.

Using the time series feature extraction library (tsfel), we traverse a vertical stack of all data, sampling 100 discrete data points over each 250ms time series for each axis of each sensor. We compute 184 features[7] from each set of data points using tsfel (see Fig. 3 for an example). Of these features, we drop those that are constant across the training set, resulting in a test feature set with dimensions 984×1032, each row of which indicates a single time window in our training set and each column of which represents a feature of a particular sensor axis over that window.

3 Methods

3.1 Machine learning methods

We have chosen a Random Forest Classifier[8] (RFC) as the first model for this categorization problem. An RFC constructs a set of decision trees through randomized selection with replacement of a subset of training data, and at each node a randomized subset of features of a size $p_{node} = \sqrt{p_{parent}}$ is used to determine its predictive split. Each of these decision trees then votes on the highest label probabilities for a given feature vector, and the ensemble of votes from different trees yields a net probability for categorization. As we have a significant quantity of time series decompositions in our feature set with little domain knowledge about their potential correlations, the randomized composition and weighting of the decision trees constructed by this method is useful. In addition, the RFC method ensures that outliers in the data have minimal impact and tends to lower model variance.

As our second model, we have chosen the Histogram-based Gradient Boosting Classifier (HGBC)[9], which is an ensemble model that also uses decision trees, but with distinct behavior in tree construction. First, the method takes the training features and creates a histogram of feature values, reducing the computational overhead for tree optimization and the impact of outliers in the data. Next, a decision tree ('base learner') is constructed from these histograms, with nodal split points for prediction at the boundaries between bins. The loss function of the predictions for this tree is used as a gradient to construct ('boost') a second tree, modifying subsequent decision weights to minimize loss, and this continues for n trees (until the loss difference falls below an epsilon or a maximum iterations hyperparameter is reached), at which point the ensemble of constructed trees votes on a final categorization for the input feature vectors. Additional trees are weighted in ensemble voting by a learning rate (.1 by default), to reduce the influence of individual trees and to leave space for future trees to improve the model.[10] The iterative model correction offered by an HGBC allows us to optimize a non-linear feature weighting, which is appropriate for time series data, while avoiding the impact of noisy features and bias from features that may have some correlation.[11]

3.2 Loss function

Our RFC and HGBC models make use of the log loss function, which in the context of an individual decision tree indicates the quality of a split. At each node we minimize the equation

$$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(p_i) + (1 - y_i) log(1 - p_i)] \tag{1}$$

With N as the number of predictions, y_i as the target label for a given feature vector i, and p_i as the predicted categorization estimator of the feature vector i. In simple language, the use of log loss describes the node's predictive accuracy by measuring the accuracy of using a feature set to predict the class labels of an input, and in minimizing this value we can determine the ideal criteria for predictive split at the node.

With the HGBC, a gradient of the loss function in the direction of loss minimization is added to each subsequent decision tree to iteratively improve decision weights and subsequent predictive accuracy.

Log loss is easily differentiable and penalizes false positives. We settled on log loss for our RFC model through hyperparameter optimization with a validation set, while the differentiability of log loss makes it optimal for HGBC.

3.3 Dataset Splitting

The data is split using scikit-learn's train_test_split function with a 67% train/test ratio, and the training set is further split with the same function into training and validation sets with an 80% train/val ratio. The train_test_split uses shuffle split to randomize and separate the data into a given ratio. The 54-13-33 train-val-test ratio is commonly used for machine learning applications with comparatively small datasets[12]. It is important to split the data randomly, since the data samples are ordered by creation date and some unwanted correlations might exist. Also, as we have a somewhat limited dataset, methodology such as k-fold cross validation or other subsampling methods could potentially lead to overfitting and high variance among smaller training sets.

3.4 Hyperparameter Optimization & Error

We made use of a validation set to optimize hyperparameters for both models. The RFC was optimized against the validation set using RandomizedSearchCV with a wide range of input parameters and a sufficiently high iteration count. As such, classification accuracy on the test set increased by about 5%. There is considerable seemingly random behaviour of the accuracy and the out-of-bag error when optimizing the parameters. (see figure 7) Even with large iteration scores, random search didn't seem to converge to a single parameter range, therefore we decided do not use grid-parameter-search as allocating more resources to the random search yielded better results than using a grid-search on an unbound parameter space. With the HGBC model, we used a held-out validation set to optimize the iterative run time of our model, minimizing train error while testing against a validation set for each iteration until steady model performance was achieved within a small epsilon for 5 consecutive iterations (see Fig. 8). As RFC and HGBC both tend towards overfitting on the training set as a function of model design, out-of-bag error and train/val error optimization were used, respectively, for assessment of the models. Out-of-bag error refers to the error of random forest decision trees against held-out training features not used in the given tree, and as such is generalizable to the error of the model on future unseen feature data.

4 Results

The combination of time-series feature extraction and RFC yielded promising results. The test set accuracy for exact matches obtained by HGBC is 73.5%, and for RFC, 67.2%. When using a 4-neighborhood evaluation (see figure 4) with a distance smaller than one, the success rate is 92%. The highest confusion is at the labels 8 $\rightarrow$ 0 and 7 $\rightarrow$ 5, which in both cases are buttons next to each other with a similar motion when pressed with the right thumb.

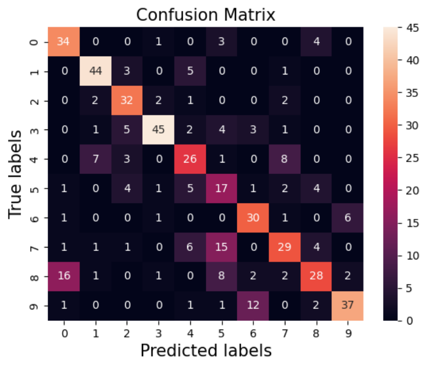


Figure 2: Confusion matrix of time series features and random forest classifier

If we consider our problem space, we can take the arbitrary goal of unlocking a phone's numeric keypad with a 4-button sequence within a window of 5 successive entries, as this is a common behavior for most mobile platforms. We can find the probability to successfully unlock a phone with these constrains, P_s, using the probability of categorical key prediction p_c as follows:

$$p_u = p_c^4 \quad (2) \qquad P_f = (1-p_u)^5 \quad (3) \qquad P_s = 1 - P_f \quad (4) \qquad P_s = 1-(1-p_c^4)^5 \quad (5)$$

With p_u as the probability of an individual 4-key unlock attempt and P_f as the probability of failure in a 5-entry window. As per equation 5, with our given results, the RFC model affords us a 61% chance of success and the HGBC yields an 81% chance, which we find to be an acceptable threshold for success, though to approach a 95% success rate, we must have a predictive accuracy exceeding 82%, which would take some substantial model refinement.

It is clear that our model shows promise, and we would hope that further data collection would lead to improved model performance – magnetometer readings as an additional set of raw data points could potentially improve our model's inference of phone kinetics. Also, as per our research[5], there may be a more optimal selection method for multivariate time series data.

References

[1] Emmanuel Owusu et al. 'ACCessory: Password inference using accelerometers on smartphones'. In: (Feb. 2012). DOI: 10.1145/2162081.2162095 (cit. on p. 1).

[2] Liang Cai and Hao Chen. 'TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion'. In: *6th USENIX Workshop on Hot Topics in Security (HotSec 11)*. San Francisco, CA: USENIX Association, Aug. 2011. URL: https://www.usenix.org/conference/hotsec11/touchlogger-inferring-keystrokes-touch-screen-smartphone-motion (cit. on p. 1).

[3] Abdul Rehman Javed et al. 'AlphaLogger: detecting motion-based side-channel attack using smartphone keystrokes'. In: *Journal of Ambient Intelligence and Humanized Computing* (2020), pp. 1–14. URL: https://api.semanticscholar.org/CorpusID:212719620 (cit. on p. 1).

[4] Liang Cai, Sridhar Machiraju and Hao Chen. 'Defending against Sensor-Sniffing Attacks on Mobile Phones'. In: *Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds*. MobiHeld '09. Barcelona, Spain: Association for Computing Machinery, 2009, pp. 31–36. ISBN: 9781605584447. DOI: 10.1145/1592606.1592614. URL: https://doi.org/10.1145/1592606.1592614 (cit. on p. 1).

[5] Bahavathy Kathirgamanathan and Padraig Cunningham. 'A Feature Selection Method for Multi-dimension Time-Series Data'. In: (2020), pp. 220–231. DOI: 10.1007/978-3-030-65742-0_15. URL: https://doi.org/10.1007%2F978-3-030-65742-0_15 (cit. on pp. 2, 4).

[6] Marília Barandas and et al. 'TSFEL: Time Series Feature Extraction Library'. In: *SoftwareX* (2020). DOI: https://doi.org/10.1016/j.softx.2020.100456. URL: https://github.com/ElsevierSoftwareX/SOFTX_2020_1 (cit. on p. 2).

[7] Descriptions of extracted features and related code explained at: URL: https://tsfel.readthedocs.io/en/latest/descriptions/feature_list.html (visited on 19/09/2023) (cit. on p. 2).

[8] L. Breiman. 'Random Forests.' In: *Machine Learning 45* (2001), pp. 5–32. DOI: https://doi.org/10.1023/A:1010933404324 (cit. on p. 2).

[9] Description of scipy's Histogram Gradient Boosting Classifier at: URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.HistGradientBoostingClassifier.html (visited on 19/09/2023) (cit. on p. 3).

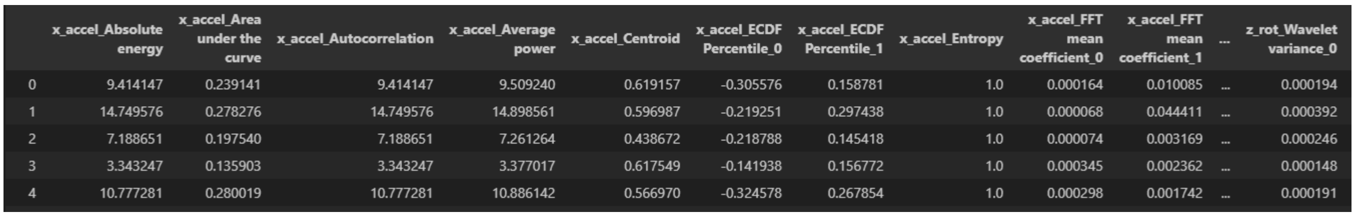
[10] Jerome H. Friedman. 'Stochastic gradient boosting'. In: *Computational Statistics & Data Analysis* 38.4 (2002). Nonlinear Methods and Data Mining, pp. 367–378. ISSN: 0167-9473. DOI: https://doi.org/10.1016/S0167-9473(01)00065-2. URL: https://www.sciencedirect.com/science/article/pii/S0167947301000652 (cit. on p. 3).

[11] Syed Md. Minhaz Hossain and Kaushik Deb. 'Plant Leaf Disease Recognition Using Histogram Based Gradient Boosting Classifier'. In: *Intelligent Computing and Optimization*. Ed. by Pandian Vasant, Ivan Zelinka and Gerhard-Wilhelm Weber. Cham: Springer International Publishing, 2021, pp. 530–545. ISBN: 978-3-030-68154-8 (cit. on p. 3).

[12] Kevin K Dobbin and Richard M Simon. 'Optimally splitting cases for training and testing high dimensional classifiers'. In: *BMC medical genomics* 4.1 (2011), pp. 1–8 (cit. on p. 3).

5 Appendix

5.1 Additional Figures



	x_accel_Absolute energy	x_accel_Area under the curve	x_accel_Autocorrelation	x_accel_Average power	x_accel_Centroid	x_accel_ECDF Percentile_0	x_accel_ECDF Percentile_1	x_accel_Entropy	x_accel_FFT mean coefficient_0	x_accel_FFT mean coefficient_1	...	z_rot_Wavelet variance_0
0	9.414147	0.239141	9.414147	9.509240	0.619157	-0.305576	0.158781	1.0	0.000164	0.010085	...	0.000194
1	14.749576	0.278276	14.749576	14.898561	0.596987	-0.219251	0.297438	1.0	0.000068	0.044411	...	0.000392
2	7.188651	0.197540	7.188651	7.261264	0.438672	-0.218788	0.145418	1.0	0.000074	0.003169	...	0.000246
3	3.343247	0.135903	3.343247	3.377017	0.617549	-0.141938	0.156772	1.0	0.000345	0.002362	...	0.000148
4	10.777281	0.280019	10.777281	10.886142	0.566970	-0.324578	0.267854	1.0	0.000298	0.001742	...	0.000191

Figure 3: Top 5 rows of training features

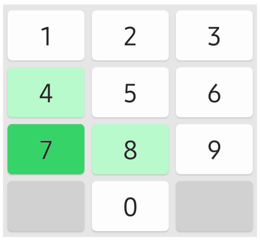
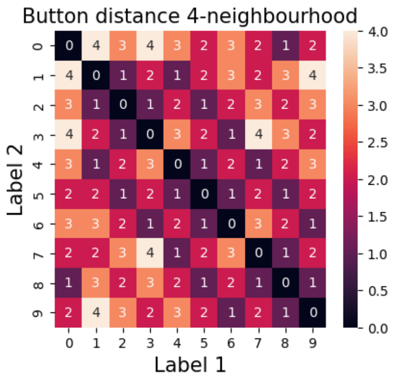


Figure 4: 4-Neighborhood relation of ’7’ on touch keyboard



	0	1	2	3	4	5	6	7	8	9
0	0	4	3	4	3	2	3	2	1	2
1	4	0	1	2	1	2	3	2	3	4
2	3	1	0	1	2	1	2	3	2	3
3	4	2	1	0	3	2	1	4	3	2
4	3	1	2	3	0	1	2	1	2	3
5	2	2	1	2	1	0	1	2	1	2
6	3	3	2	1	2	1	0	3	2	1
7	2	2	3	4	1	2	3	0	1	2
8	1	3	2	3	2	1	2	1	0	1
9	2	4	3	2	3	2	1	2	1	0

Figure 5: 4-Neighborhood relation in reference to confusion matrix

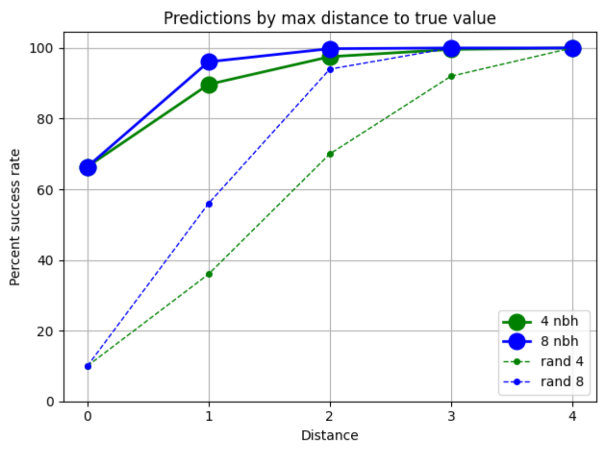


Figure 6: Neighbourhood accuracy compared to random guesses

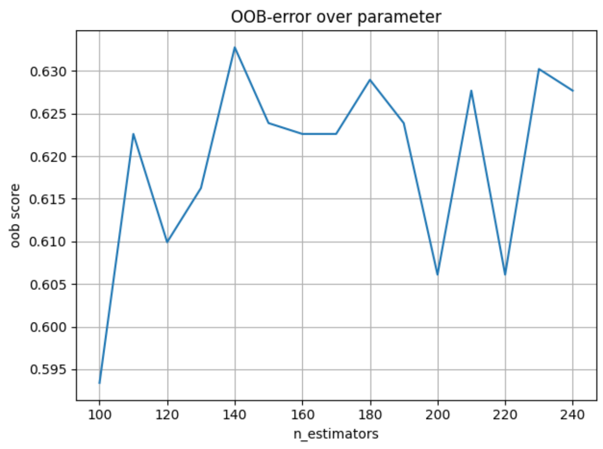


Figure 7: Example for inconclusive results in hyperparameter tuning

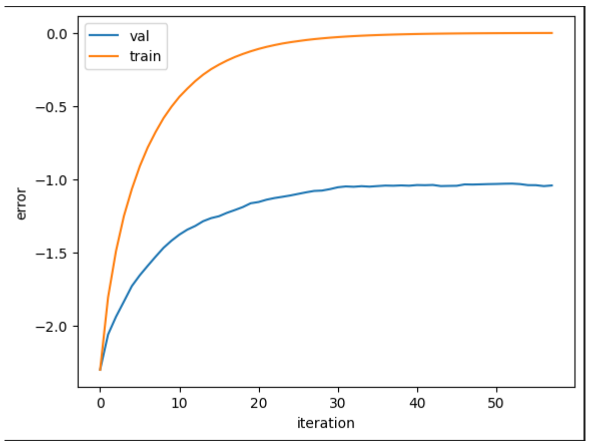


Figure 8: Train & Validation error for HGBC

5.2 Android app

```
data class QueueEntry(
    val timestamp: Long,
    val sensortype: String,
    val valueX: Float,
    val valueY: Float,
    val valueZ: Float
)

class MainActivity : ComponentActivity(), SensorEventListener {

    val queueData: Queue<QueueEntry> = LinkedList()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        var manager: SensorManager = getSystemService(Context.SENSOR_SERVICE)
            as SensorManager;

        manager.registerListener(this@MainActivity,
            manager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION), 0);
        manager.registerListener(this@MainActivity,
            manager.getDefaultSensor(Sensor.TYPE_GYROSCOPE), 0);

        setContent {
            KeyAccelListenerTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Column(
                        modifier = Modifier
                            .fillMaxSize()
                            .padding(16.dp),
                        horizontalAlignment = Alignment.CenterHorizontally,
                        verticalArrangement = Arrangement.spacedBy(16.dp)
                    ) {
                        Greeting(
                            "Welcome to KeyAccelListener",
                            modifier = Modifier.align(Alignment.CenterHorizontally)
                        )
                        Text(
                            text =
                                "Type something in the TextField to start recording sensor data
                            modifier = Modifier.align(Alignment.CenterHorizontally)
                        )
                        TextFieldNum()
                    }
                }
            }
        }
    }

    private fun getStorageDir(): String? {
        return Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_DOCUMENTS).absolutePath
    }

    public override fun onSensorChanged(evt: SensorEvent) {
            try {
                if (evt.sensor.type == Sensor.TYPE_LINEAR_ACCELERATION){
                    queueData.offer(QueueEntry(evt.timestamp, "ACC", evt.values[0],
                    evt.values[1], evt.values[2]))
                }
                if (evt.sensor.type == Sensor.TYPE_GYROSCOPE){
                    queueData.offer(QueueEntry(evt.timestamp, "ROT", evt.values[0],
                    evt.values[1], evt.values[2]))
                }

                // remove old elements
                while(queueData.element().timestamp < evt.timestamp - 1000000000) // timestamp
                {
                    queueData.remove()
                }
            } catch (e: IOException) {
                e.printStackTrace();
            }
        }

    @Override
    public override fun onAccuracyChanged(sensor: Sensor?, i: Int) {}
```

```
    @OptIn(ExperimentalMaterial3Api::class)
    @Composable
    fun TextFieldNum() {
        var text by remember { mutableStateOf(TextFieldValue("")) }
        val scope = rememberCoroutineScope()
        TextField(
            value = text,
            label = { Text(text = "Number Input") },
            keyboardOptions = KeyboardOptions(keyboardType =
                KeyboardType.Number),
                onValueChange = { newString ->
                    text = newString
                    scope.launch {
                        delay(500)
                        storeDataToFile(text.text)
                        text = TextFieldValue("")
                    }

                }
        )
    }

    public fun storeDataToFile(letter: String){
        var writer: FileWriter = FileWriter(File(getStorageDir(),letter + "_"
            + System.currentTimeMillis() + ".csv"));
        writer.write(String.format("timestamp, datatype, x_value, y_value, z_value\n"));
        for(entry in queueData){
            writer.write(String.format("%d, %s, %f, %f, %f\n",
            entry.timestamp, entry.sensortype, entry.valueX, entry.valueY, entry.valueZ));
        }
        writer.close()
    }
}
```

5.3 ML Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import re
import tsfel
import glob

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix  # evaluation metrics
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier

# importing data from .csv files
path = "/data"
csv_filenames = glob.glob(path + "/*.csv")
pattern = r"data/([^/])_"

raw_data_start = 60
raw_data_len = 100
sensors = 6

labels = []
preprocessed_data = np.zeros([1, raw_data_len, sensors])

for filename in csv_filenames:
    data = pd.read_csv(filename, skipinitialspace=True)

    label = re.search(pattern, filename).group(1)

    data_rot = (data[data["datatype"] == "ROT"]).drop(["datatype"], axis=1).to_numpy()
    data_accel = (data[data["datatype"] == "ACC"]).drop(["datatype"], axis=1).to_numpy()

    data_processed = np.ndarray([data_accel.shape[0], 7])

    # interpolate data to synchronise both sensors data-streams together
    data_processed[:, 0:4] = data_accel
    for i in range(4, 7):
        data_processed[:, i] = np.interp(
            data_processed[:, 0], data_rot[:, 0], data_rot[:, i - 3]
        )

    data_processed = data_processed[raw_data_start : raw_data_start + raw_data_len, :]

    labels.append(label)
    preprocessed_data = np.concatenate((preprocessed_data, [data_processed[:, -6:]]))
```

```
labels = np.array(labels)
preprocessed_data = preprocessed_data[1:, :, :]  # remove time

# reformat datax
preprocessed_data_reshaped = np.vstack(preprocessed_data)
preprocessed_data_df = pd.DataFrame(
    preprocessed_data_reshaped,
    columns=["x_accel", "y_accel", "z_accel", "x_rot", "y_rot", "z_rot"],
)

# use tsfel (Time Series Feature Extraction) to extract features
cfg_file = tsfel.get_features_by_domain()
X_tsfel = tsfel.time_series_features_extractor(
    cfg_file, preprocessed_data_df, fs=100, window_size=100
)


# remove colums with no relevant data
def cleanFeatures(df: pd.DataFrame) -> pd.DataFrame:
    mask = df.nunique() == 1
    drop_cols = mask[mask].index
    df_dropped = df.drop(columns=drop_cols)
    return df_dropped


X = cleanFeatures(X_tsfel).copy()

# mean-normalize data
X = (X - X.mean()) / X.std()

# visualise data with two parameter pca

pca_vis2_tsfel = PCA(n_components=2)
pca_vis2_tsfel.fit(X)

fig, ax = plt.subplots(1, 1)
ax.grid(True)
plt.tight_layout()

markers = ["o", "^", "v", ">", "<", "*", "s", "p", "x", "+", "."]
colors = ["b", "g", "r", "k", "c", "m", "y", "lime", "orange", "navy", "pink"]
```

```
plot_labels = ["1", "3", "7", "9"]

for label_i in range(labels.shape[0]):
    if labels[label_i] not in plot_labels:
        continue

    marker = markers[plot_labels.index(labels[label_i])]
    color = colors[plot_labels.index(labels[label_i])]
    data_temp = pca_vis2_tsfel.transform(X.iloc[[label_i]])
    ax.scatter(data_temp[:, 0], data_temp[:, 1], s=10, marker=marker, color=color)


# split data into training and testing data
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.33)

# use sklearn classifier on data
clf = RandomForestClassifier()
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
multi_accuracy = accuracy_score(y_test, y_pred)
c_mat = confusion_matrix(y_pred, y_test)

ax = plt.subplot()
sns.heatmap(c_mat, annot=True, fmt="g", ax=ax)
ax.set_xlabel("Predicted labels", fontsize=15)
ax.set_ylabel("True labels", fontsize=15)
ax.set_title("Confusion Matrix", fontsize=15)

# analyze results

# Define the keyboard layout
keyboard_layout = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [None, 0, None]]

# Create the 10x10 matrix of 4-neighborhood distances
distnace_matrix_4nbh = np.ndarray((10, 10))
distnace_matrix_8nbh = np.ndarray((10, 10))

for key1_i in range(12):
    for key2_i in range(12):
        key1 = keyboard_layout[key1_i // 3][key1_i % 3]
        key2 = keyboard_layout[key2_i // 3][key2_i % 3]
        if key1 is not None and key2 is not None:
            distnace_matrix_4nbh[key1, key2] = abs(key1_i // 3 - key2_i // 3) + abs(
                key1_i % 3 - key2_i % 3
            )
            distnace_matrix_8nbh[key1, key2] = max(
                abs(key1_i // 3 - key2_i // 3), abs(key1_i % 3 - key2_i % 3)
            )

# visualize neighbourhood relations between labels
fig, ax = plt.subplots(1, 2, figsize=(10, 4))
sns.heatmap(distnace_matrix_4nbh, annot=True, fmt="g", ax=ax[0])
sns.heatmap(distnace_matrix_8nbh, annot=True, fmt="g", ax=ax[1])
ax[0].set_title("Button distance 4-neighbourhood")
ax[1].set_title("Button distance 8-neighbourhood")

dist_keys = 5
labled_correct_4nbh = np.ndarray(dist_keys)
labled_correct_8nbh = np.ndarray(dist_keys)
rand_correct_4nbh = np.ndarray(dist_keys)
rand_correct_8nbh = np.ndarray(dist_keys)

sum_samples = np.sum(c_mat)

for index in range(dist_keys):
    labled_correct_4nbh[index] = (
        np.sum(c_mat * np.where(distnace_matrix_4nbh <= index, 1, 0))
        / sum_samples
        * 100
    )
    labled_correct_8nbh[index] = (
        np.sum(c_mat * np.where(distnace_matrix_8nbh <= index, 1, 0))
        / sum_samples
        * 100
    )
    rand_correct_4nbh[index] = np.sum(np.where(distnace_matrix_4nbh <= index, 1, 0))
    rand_correct_8nbh[index] = np.sum(np.where(distnace_matrix_8nbh <= index, 1, 0))

fig, ax = plt.subplots(1, 1)
ax.set_title("Predictions by max distance to true value")
ax.plot(
    np.arange(dist_keys),
    labled_correct_4nbh,
    color="green",
    marker="o",
    linestyle="solid",
    linewidth=2,
    markersize=12,
)
ax.plot(
    np.arange(dist_keys),
    labled_correct_8nbh,
    color="blue",
    marker="o",
    linestyle="solid",
    linewidth=2,
    markersize=12,
)
ax.plot(
    np.arange(dist_keys),
    rand_correct_4nbh,
    color="green",
    marker="o",
    linestyle="dashed",
    linewidth=1,
    markersize=4,
)
ax.plot(
    np.arange(dist_keys),
    rand_correct_8nbh,
    color="blue",
    marker="o",
    linestyle="dashed",
    linewidth=1,
    markersize=4,
)
ax.grid(True)
plt.legend(["4 nbh", "8 nbh", "rand 4", "rand 8"])
plt.xlabel("Distance")
plt.ylabel("Percent success rate")
plt.xticks([0, 1, 2, 3, 4])
plt.yticks(np.arange(0, 101, 20))
plt.tight_layout()
plt.show()


print(
    f"Prediction accuracy distance 0:    4nbh: {labled_correct_4nbh[0]:.2f}%, 8nbh:\
    {labled_correct_8nbh[0]:.2f}%"
)
print(
    f"Prediction accuracy distance >= 1: 4nbh: {labled_correct_4nbh[1]:.2f}%, 8nbh:\
    {labled_correct_8nbh[1]:.2f}%"
)
```