Project Report
Classifying signals from accelerometer type sensors of phones

1. Introduction and Objectives

This project originates from a university lab exam and focuses on applying various **machine learning** models to **classify signals** gathered from accelerometer sensors on four different phones.

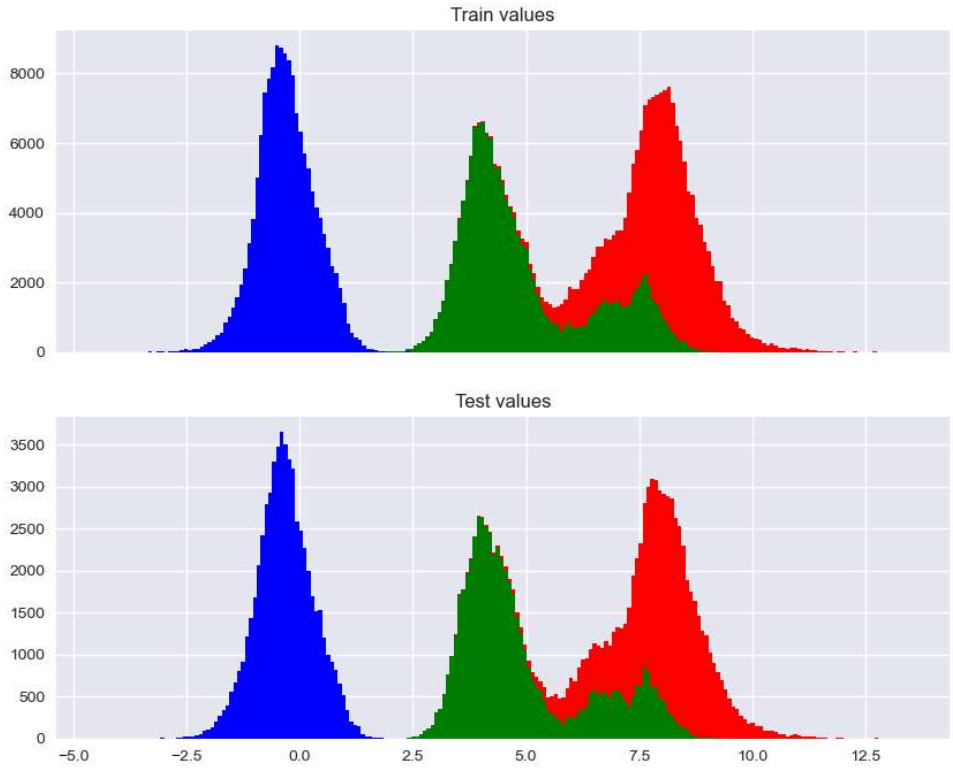
2. Data Insights

The dataset consists of **1000 training** samples and **400 testing** samples, with each signal stored in a separate .txt file. Each file contains three columns representing the **x, y, z axes** of the gravitational force. Each row in a file provides three values corresponding to the **g-force components** along each axis at a specific time. The number of rows in each file depends on the number of values recorded within a **1.5-second timeframe**.

I've calculated the minimum, maximum, average, and standard deviation for both the training and testing sets. The results are the following:

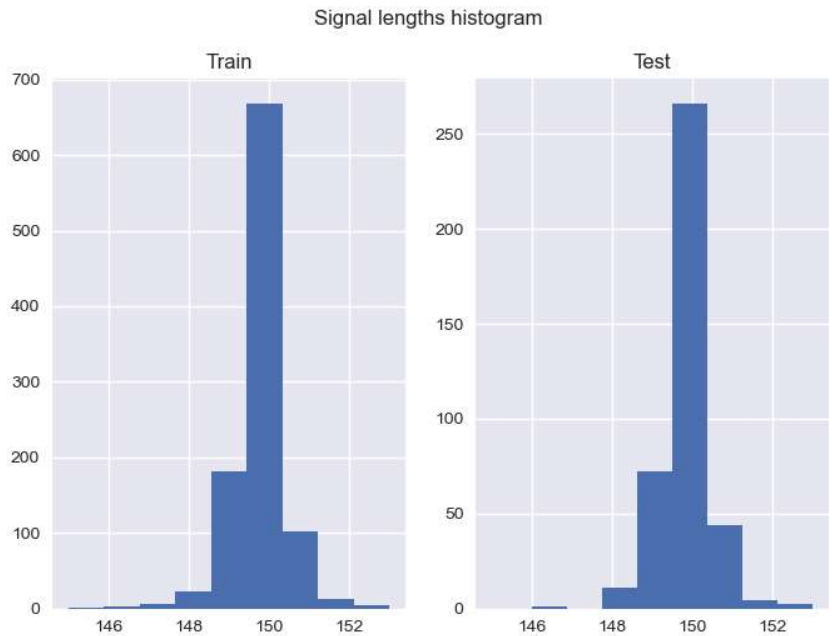
	Training	Testing
Min	-4.487	-3.599
Max	13.014	13.465
Avg	4.195	4.187
Std Dev (σ)	3.585	3.594

The following plot shows **histograms for both sets**. Note that the x-axis values are shown in blue, y-axis values in green, and z-axis values in red.



The code for those calculating the values and plotting the histograms can be found in the *insights.ipynb* notebook.

Furthermore, the signals have **different lengths** because the reporting frequency is different. As we can see below, the signal length varies between **145** and **153** lines, the average for both training and testing set being 149.9.



3. Data Preprocessing

The notebook *length-normalization.ipynb* includes code to **standardize the length** of each signal file. There are two approaches: either **reducing** each file to a specified length or **extending** them by padding with zeros up to a given length. The resulting datasets are saved in the directories *data_short* and *data_long*.

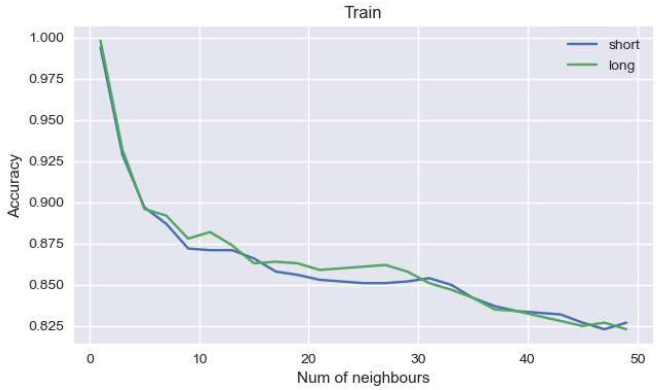
Some of the tests are done by transforming the signal into the associate **Markov transition matrix**. For each axis of the signal the values are discretized in k intervals. For each axis we compute a matrix A of size $k \times k$ where the $A(i, j)$ component is the probability to get from interval i to interval j. This probability is computed as follows: when traversing a discretized signal along a specific axis from the first to the last value, at each step t, increment the corresponding cell in matrix A. To transform the values in A into probabilities, normalize matrix A by dividing each element in a row by the sum of elements in that row. A signal feature vector is obtained by flattening and concatenating the three matrices corresponding to the x, y, and z axes. Every signal should be transformed in a **feature vector of length $3 * k * k$**.

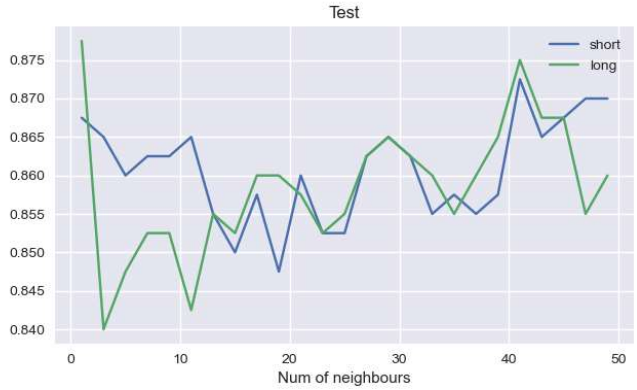
4. Model Selection

Inspired by the original exam, the models used are **k-Nearest Neighbors** (k-NN), **Support Vector Machines** (SVMs) and **Neural Networks**.

4.1 k-NN

The model utilizes **Manhattan distance** to determine the nearest neighbors for each signal in the dataset. It was trained with the feature vectors resulted from the Markov transition matrices. K was varied from 1 to 49, selecting only odd numbers, for both the short and long datasets.





As observed from the above graphs, **low k values** (1–7) yield high accuracy on the training set but lead to overfitting, as seen by fluctuations in test accuracy. **Larger k values** (20–30) seem more suitable for balanced performance on the test set, although they lead to slightly lower accuracy on the training set.

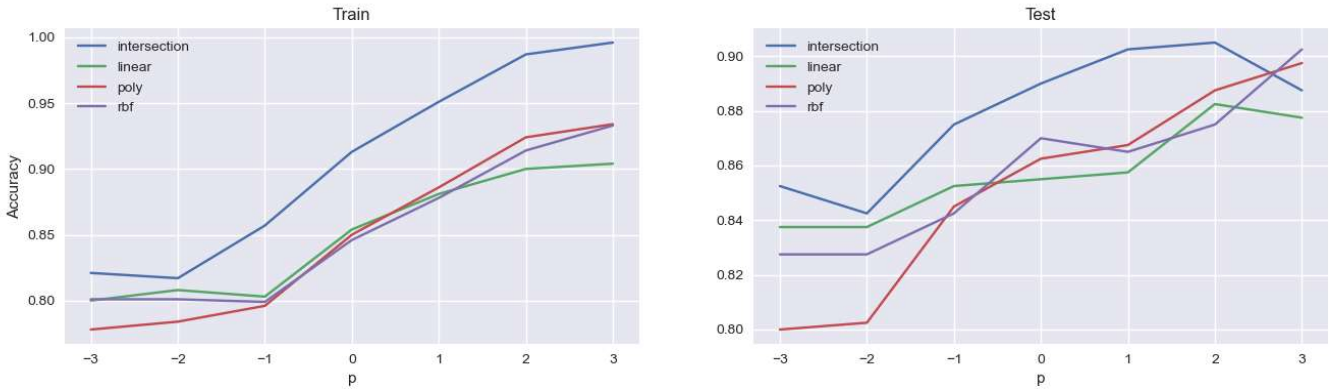
The minor length difference between *short* and *long* do influence performance slightly in the test set, long generally achieving slightly higher accuracy.

The **best k** is 41, as in both the long and short datasets, the accuracy is a little over **0.87**.

4.2 SVM

I experimented with various **kernel functions** for the Support Vector Machine model, including **linear, polynomial**, and radial basis function (**rbf**). The model was trained with the feature vectors resulted from the Markov transition matrices from the long dataset. Additionally, I implemented a **custom intersection kernel** using a precomputed approach. This kernel calculates similarity by summing the minimum values of corresponding elements in each vector for each element i in the a,b vectors:

$$K(a,b) = \sum_{i=1}^{n} min(ai, bi)$$



By **varying the regularization parameter**, $C = 10^p$, we can see how the intersection kernel is a good variant for values of C between 1 and 10 even if it looks to be overfitting after a certain C value.

The code for k-NN and SVM models can be found in the *knn-svm.ipynb* notebook, along with the code generating the plots.

4.3 Neural Networks

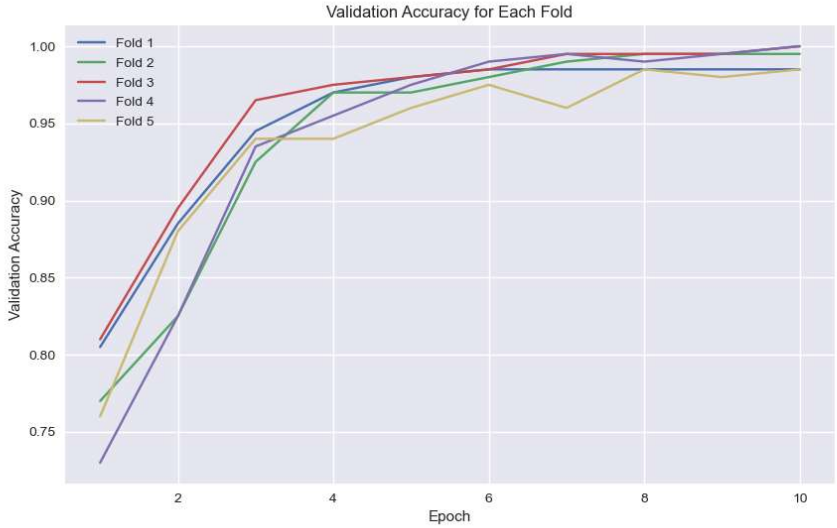
I implemented a **custom PyTorch Dataset** class to prepare and standardize our data for model training and evaluation. By inheriting from the PyTorch Dataset class, we defined the essential methods required for efficient data handling: *__init__*, *__len__*, and *__getitem__*. This approach enables compatibility with PyTorch's DataLoader, facilitating batched data loading. **Standardization** is important, as it ensures that features have a mean of zero and a standard deviation of one. I applied a *StandardScaler* from the *sklearn.preprocessing* library to the training data and passed the fitted scaler as a parameter to the testing data.

The **architecture** I tried is the following:

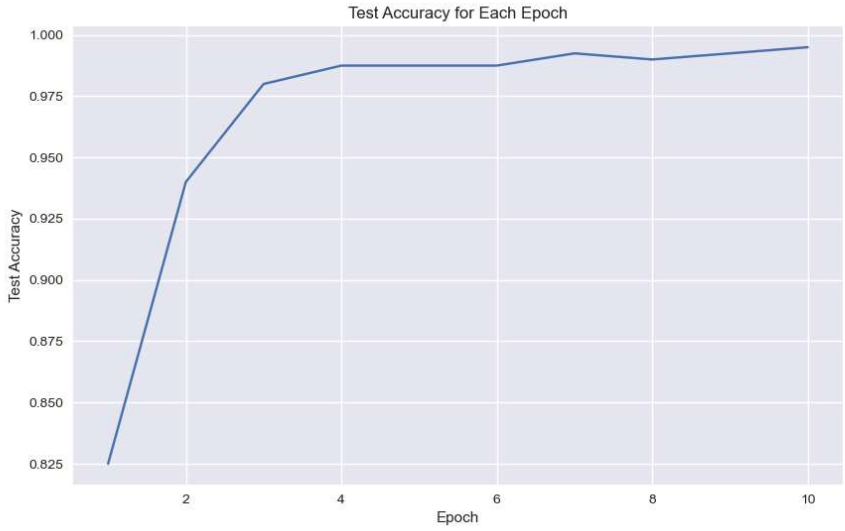
i. ***nn.Flatten()*** flattens the input tensor to a single vector;
ii. ***nn.Linear(3 * 153, 32)*** is a fully connected layer that reduces the 459 features to 32 ;
iii. ***nn.Linear(32, 32)*** another fully connected layer with 32 units that maintains the dimensionality;
iv. ***nn.Linear(32, 4)*** the output layer which produces the score for the 4 classes

The two hidden layers use **ReLU activation** to introduce non-linearity. I used the **Adam optimizer** with a learning rate of 10^{-3} and the loss is calculated using **Cross-entropy loss**.

The training data was split in **5 folds** to properly evaluate the model. The result can be seen below.



Average **validation accuracy** of the folds is **0.99** so the architecture can proceed to be trained on the full dataset and be tested.



After 10 epochs, the **testing accuracy** is **0.995**, the model being able to almost perfectly classify the signals.

The code for the Neural Network and generating the plots can be found in the *nn.ipynb* notebook.