NTNU
Kunnskap for en bedre verden

Department of Electronic Systems

TFE4580 - Electronic Systems Design and Innovation, Specialization Project

Machine learning for Human Activity Recognition

Author:
Marcus Notø

December, 2021

Abstract

Human Activity Recognition (HAR) has received a vast amount of research contributions after the development of wearable technologies. Big technological companies such as Apple, Google and Huawei has invested heavily in wearable technologies and the adaptation of Artificial Intelligence software and hardware to enhance activity recognition abilities. Wearable sensors can benefit their users by accumulating activity data and give feedback of abnormalities, falling detection, predicting human behaviour, general health status or as a supporting tool for other assistive technologies. Activity recognition is said to play a vital role in reducing costs and workloads currently being enforced on health services and eldercare. This paper introduces a HAR system using Deep Learning (DL) classifiers and compares their performance with classical Machine Learning (ML) classifiers. The proposed DL classifier was a Recurrent Neural Network (RNN) using a Long Short-Term Memory (LSTM) unit and a bidirectional LSTM (BLSTM) unit. The classifiers were trained on accelerometer data gathered from a smartphones IMU on 6 basic activities (Jogging, walking, walking up/down-stairs, sitting and standing). Random forest was the best performing classical classifier with a accuracy of 89 % against 95 % for the proposed DL classifier. Furthermore, the RNN based activity recognition system was compared against other state-of-the-art solutions. The system performed 3.3% better then a Convolutional Neural Network (CNN) system on the same dataset. It was proposed that a form of data augmentation could enhance the systems performance further because of the unbalance between the number of samples for each activity. The proposed HAR system should be implemented on other datasets and activities to further validate its robustness for activity recognition before adopting the system to help understanding people's behaviour and increase their quality of life.

Table of Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence. i

ANN Artificial Neural Network. 15, 16

BLSTM Bidirectional Long Short-Term Memory. i, 30, 33, 35, 36, 41

CART classification and regression tree. 13, 14

CNN Convolutional Neural Network. i, 39–41

DL Deep Learning. i, v, 2, 5, 15, 16, 18, 22, 26, 30–33, 35, 39, 41, 42

DNN Deep Neural Networks. 16, 30, 33, 35, 36, 39–41

DT Decision Tree. 13–15, 30, 35, 36, 41

FN False Negative. 8

FP False Positive. 8

GIGO Garbage In, Garbage Out. 26

HAR Human Activity Recognition. i, 1–3, 25, 41, 42

ICT Information and Communication Technology. 1

IG Information Gain. 14

IMU Inertial measurement units. i, 1, 2

IoT Internet of Things. 1

LR Logistic Regression. 8, 30, 41

LSTM Long Short-Term Memory. i, 20, 30, 33, 35, 36, 39–42

MAP Maximum a posteriori Estimator. 4

ML Machine Learning. i, v, 2–7, 13, 15, 22, 26, 27, 30, 31, 35, 41

MLE Maximum Likelihood Estimator. 4, 5, 17

MLP Multilayer Perceptron. 16

MSE Mean Squared Error. 4

NaN Not a Number. 26, 27

NLP Natural Language Processing. 18, 41

NN Neural Network. 1, 6, 15–18, 31, 33, 39

PCA Principal Component Analysis. 28

ReLU Rectified Linear Unit. 17, 20

RF Random Forest. 30, 35, 41

RNN Recurrent Neural Network. i, iv, 18–21, 30, 33, 35, 36, 39, 41

SGD Stochastic Gradient Descent. 5, 6, 10, 13

SMOTE Synthetic Minority Over-sampling Technique. 40

SVC Support Vector Classifier. 30

SVM Support Vector Machine. 11, 13, 32, 41

TN True Negative. 8

TP True Positive. 8, 36

WISDM Wireless Sensor Data Mining. iv, 22, 25, 26, 40

1 Introduction

1.1 Background and motivation

Automatic detection of human activity is postulated to have a significant role in the future of healthcare and eldercare in the form of a assistive technology to other technologies like Internet of Things (IoT). The advances in Information and Communication Technology (ICT) combined with IoT breakthroughs can give healthcare personnel better information, enabling more home-centric healthcare and eldercare and sustaining active and healthy ageing [1]. Research within Human Activity Recognition (HAR) is essential to give healthcare personnel better information about their patients. The research field has had a spring of popularity after a growth in the availability of wearable sensors in the consumer market, which has introduced more efficient hardware and classification algorithms for HAR specific tasks.

The definition of HAR is classifying a sequence of sensor data as well-defined human movements. The data can be collected from wearable sensors as Inertial measurement units (IMU) or through video frames or images. Not only is HAR beneficial for health personnel, but it can also be used as a preventive tool and enhance aspects of peoples everyday life. By gaining more knowledge on recognising activities, one can use this information to help people make better decisions and impose preventive measures to reduce inactivity or strenuous activities, which leads to a range of health problems and musculoskeletal pain. Performing regular physical activity is associated with a positive impact on people's physical and mental health. In addition, it is considered as a primary indicator to determine the quality of life [2].

Activity recognition is a complex problem to handle, and one has to consider several aspects. When considering different classification techniques, one must consider the trade-off between time complexity and accuracy. Suppose a classification model has less computational complexity and somewhat poorer accuracy than a model with greater complexity and higher accuracy. In cases of limited computing resources, it will be more acceptable to choose the one with lower accuracy [3]. In other scenarios, the priority can be on the best possible accuracy regardless of the computing complexity. This trade-off needs to be considered when implementing a classification system.

Traditionally solving HAR task was done using engineered features obtained by heuristic processes [4]. Selecting features with the lowest possible correlation and giving a good image of the original input data can be difficult and time-consuming. Therefore, new research has focused on using Neural Network (NN), enabling automated feature extraction from raw sensor inputs. This classification method could simplify the HAR process and increase accuracy.

1.2 Related Work

There exist a vast amount of research papers and literature describing Human Activity Recognition. This include papers on pre-processing techniques, feature extraction, classification techniques and HAR definition and usage [2][5][6]. The problem, however, is that much of the literature uses pre-processed datasets, multiple sensors or combining multiple inertial sensors (such as combining accelerometer, gyroscope and magnetometer data)[5][7]. Therefore, creating classification models which handle raw IMU data from a single unit emulating a smartphone or smartwatch would make the whole process more efficient, less costly and less invasive for the user. Other research papers focus mainly on feature extraction and classical machine learning methods. Therefore, it would be relevant to focus on Deep Learning (DL) methods and compare their performance with classical techniques[2][8].

1.3 Current scope

The scope of this project is to:

1. Understand the process of a HAR system, including accelerometer time series quality control and pre-processing.
2. Develop and test HAR algorithms on accelerometer data from participants performing basic activities.
3. Focus on Deep Learning classifiers and how they compared with classical Machine Learning techniques and other state-of-the-art solutions.

This paper is a joint project of SINTEF and NTNU and is a specialisation project for a master thesis.

1.4 Outline

Section 2 explains the fundamental theory of Machine Learning (ML) and Deep Learning (DL) and introduces methods for multiclass classification tasks. Section 3 describes how to implement a physical activity recognition system, which dataset is used in this paper and how classical classifiers differ in implementation compared with DL classifiers. In addition, it includes the methodology used for data collection, pre-processing, feature extraction and the training process for the different classifiers. Section 4 presents the experimental results and a critical discussion of the obtained results. Finally, section 5 summarise the conclusion and presents possible further work.

2 Theory overview

This section will introduce the fundamental theory used when creating classification algorithms for HAR. This section is based on elements from signal processing, linear algebra and machine learning.

2.1 Data Pre-processing: Feature Scaling

To simplify the mapping from input variables to output variables for ML methods one can use *Feature Scaling*. Feature scaling is a method used to simplify the mapping from input variables to output variables for machine learning methods. Many ML classifiers calculate the Euclidean distance between two points. Such methods can struggle if the range of the input values varies widely. Therefore to circumvent this problem, one should normalise the data or features to a more suitable range that is more approximately proportional to the final distance. *Feature Scaling* is also known as data normalisation in signal processing and is a common data pre-processing step. Another reason to use data normalisation is that the gradient descent can converge much faster if the data is normalised. To normalise the data between two arbitrary values [a,b], one can use the formula [9]:

$$x = (b-a)\frac{x - min(x)}{max(x) - min(x)} + a \tag{1}$$

2.2 Machine Learning Basics

Machine learning is data-driven learning. The algorithm uses data to understand a scenario. To specify, an algorithm that can improve on a task T from a performance measure P with gained experience E on said task T [10]. The experience E can be expressed as a dataset $\mathbb{D}$ with x_i being a vector of m features and y_i being the target value:

$$\mathbb{D} = \{x_i, y_i\}_{i=1}^{n} = (X, y) \tag{2}$$

ML algorithms are implementing applied statistics to solve computational tasks, e.g. classification, regression, estimation and anomaly detection. For instance, the learning task of classification is to specify which of K categories a data point x belongs to. The learning algorithm produces a function $f : \mathbb{R} \rightarrow 1, ...k$, which assigns x to a class $y = f(x)$ [10]. We categorise ML algorithms predominantly as unsupervised or supervised learning based on which data (experience E) is introduced as input during the learning process. In unsupervised learning, the algorithms try to learn

useful properties of the structure of an unlabelled dataset and model the entire distribution $p(X)$ of that dataset. For classification tasks, it can perform clustering, which divides the datasets into clusters of similar examples. For supervised learning, the algorithms try to classify the data x_i into the associated labelled target y_i from a labelled dataset. There also exist other more nuanced learning categories like semi-supervised learning, reinforcement learning and more[11][12]. When designing ML algorithms, one needs to consider the problem at hand. Depending on the learning task, one must consider the *model class*, *cost function* and *optimisation method* before implementation. The model must map features to target variables while considering aspects such as degree of linearisation, data size, accuracy, training time and interpretability of the output. Considering these factors can lead to a good ML method that can ensure generalisation (performing well on new, previously unseen examples).

2.2.1 Optimisation Method

A learning algorithm learns from a performance P, but often one cannot optimise this directly. Therefore ML is often formulated as a minimisation problem of a cost function:

$$\underbrace{J(\theta)}_{cost} = \underbrace{L(\theta)}_{Loss/error} + \underbrace{R(\theta)}_{Regularisation} \tag{3}$$

The cost function consists of a sum of loss functions over the dataset and a model complexity penalty (regularisation). The regularisation term $R(\theta)$ penalises large weights by two common means: Manhattan distance or Euclidean distance, L1 or L2 regularisation, respectively. The terms are given by

$$\begin{aligned} \text{L1: } R(\theta) &= \frac{\lambda}{2}\sum_i |\theta_i| \\ \text{L2: } R(\theta) &= \frac{\lambda}{2}\sum_i |\theta_i|^2 \end{aligned} \tag{4}$$

λ is the hyperparameter that controls the impact of the regularisation term and can be proclaimed with $C = \frac{1}{\lambda}$. Loss functions express the variation between the prediction of the model and the actual data points. It can be challenging to choose a suitable loss function since it must capture the properties of the problem and include essential aspects of the project. To find the optimal statistical estimator, there exist probability frameworks such as Mean Squared Error (MSE), Maximum Likelihood Estimator (MLE) or Maximum a posteriori Estimator (MAP). For instance, for a multiclass classification problem, one wants to estimate the probabilities for different

outcomes (class predictions) which can be done by implementing a MLE and finding the outcome that maximises the likelihood for the class predictions. Another solution could be to minimise the cross-entropy between the training data and the model's predictions as the cost function (Logarithmic loss). Further, the cost function must be maximised or minimised by a form of optimisation algorithm. In general, for ML problems, one can derive it as a finite sum optimisation:

$$\min_{\theta} J(\theta), \text{ where } J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J_i(\theta) \tag{5}$$

Just like there are different flavours of cost functions for each specific problem, there are various ways to optimise. One of the most popular optimisation algorithms used in general ML and DL is Stochastic Gradient Descent (SGD). In gradient descent, one updates the parameters θ using the gradient:

$$\left[\frac{\partial J}{\partial \theta_1}, \ldots, \frac{\partial J}{\partial \theta_k}\right]^T = \nabla_\theta J \tag{6}$$

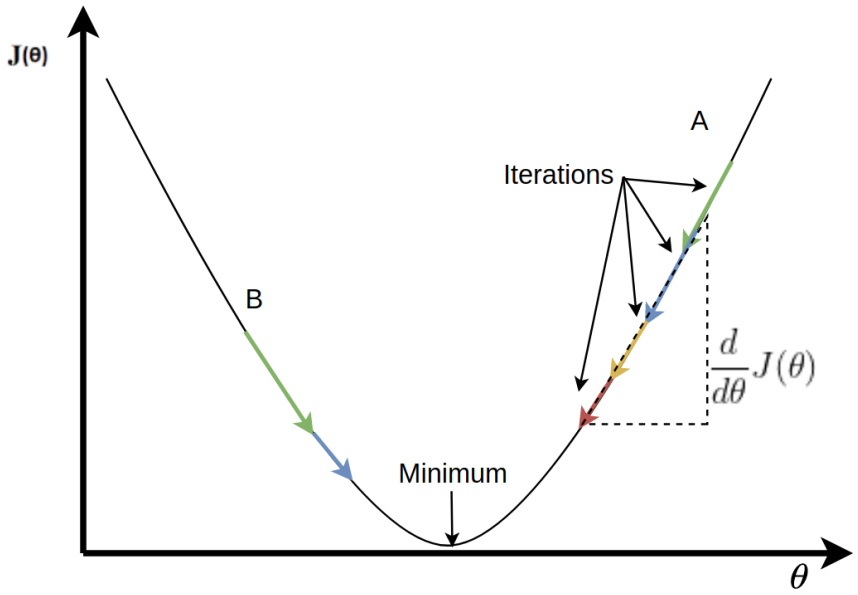


Figure 1: Illustration of two examples of gradient descent. The arrows represent the derivative term at any point. The value of θ decreases in magnitude, as shown by the size of the arrow.

In gradient descent, one searches for the minimum of $J : \mathbb{R}^p \to \mathbb{R}$ by moving in the direction of the negative gradient:

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta J(\theta_k) \tag{7}$$

Figure 1 illustrates how the gradient moves depending on two different starting points which is randomly initialised. The algorithm ensures that parameters updates in the right direction towards the minimum. The learning rate α is an important parameter that indicates how big steps towards the minimum the optimiser is taking. This value has to be tuned so that it is not too high or too low. If the value is to high it will learn faster but it might never converge against the optimal minimum, and if its to low training is not only slower, but may become permanently stuck in a sub-optimal solution. When the slope is zero, the model stops learning. The basic gradient descent method is often called *batch gradient descent*. It calculates the error for each example within the training dataset by going through the entire dataset for each run. This procedure requires much memory. In contrast, the SGD updates the parameters for each training example one by one. The frequency update is more computationally expensive, but it can be faster than the *batch gradient* approach depending on the problem. To balance the robustness of SGD and *batch gradient descent*, one can use the *mini-batch gradient descent*, the go-to method for Neural Network (NN). The method splits the training data into small batches and performs an update for each batch. There exist challenges with gradient descent, such as multiple local minimums and ill-conditioning, due to the non-convex cost function, which can be challenging to uncover. Therefore there exist other improvements to SGD which seek to ensure better performance. One of the most notable improvements is the "adaptive moments (Adam) which combines momentum and adaptive learning rates. Adam ensures that the hyperparameters become more robust, even though the learning rate must still be tuned[13].

2.2.2 Training Process

A simplified machine learning process, illustrated in figure 2, shows how the dataset $\mathbb{D}$ is split into a training D_{train}, validation D_{val} and test D_{test} set. The standard in ML is to split the data randomly to avoid biases and dependencies, where the majority (70-80%) of data points are in the training data and minority (10-15% per set) in the validation and test set. D_{test} is kept outside of the training process to ensure an unbiased and generalisable model. Generalisation is only possible if there are independent, identically distributed (i.i.d.) data, i.e. the samples D_{test} and D_{train} are identically distributed. Therefore, datasets must be balanced and should be drawn from the same distribution $p(x, y)$. Generalisation error (E_{gen}) in it self cannot be computed, instead the (E_{gen}) is estimated from the calculated error from the loss functions for each dataset (E_{train}, E_{val} and E_{test}). From this an expected postulation is given in Eq.8. The loss-gap between training, validation and test should be as small as possible to estimate a good generalisation.

$$E_{train} \leq E_{val} \leq E_{test} \approx E_{gen} \tag{8}$$

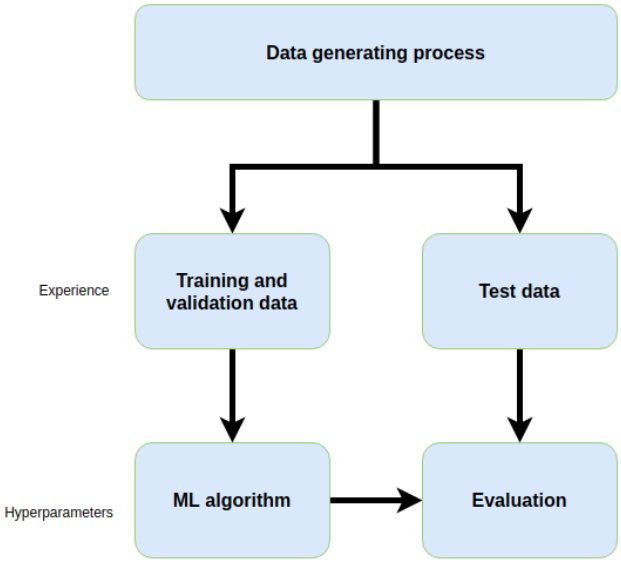


Figure 2: Simplified machine learning process.

2.2.3 Evaluating Performance

For evaluating the performance P of the ML model, one has to choose the correct metrics given task T. For a given classification task, we want to distinguish how well a ML model can recognise the correct class from the samples. A confusion matrix is often used to visualise the performance of such tasks. Figure 3 shows a confusion matrix for a binary classification task.

For classification tasks, the most popular metrics are[14]:

$$\text{Accuracy:} \quad = \frac{TP + TN}{TP + TN + FP + FN} \tag{9}$$

$$\text{Recall:} \quad = \frac{TP}{TP + FN} \tag{10}$$

$$\text{Precision:} \quad = \frac{TP}{TP + FP} \tag{11}$$

$$\text{F-score:} \quad = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{12}$$

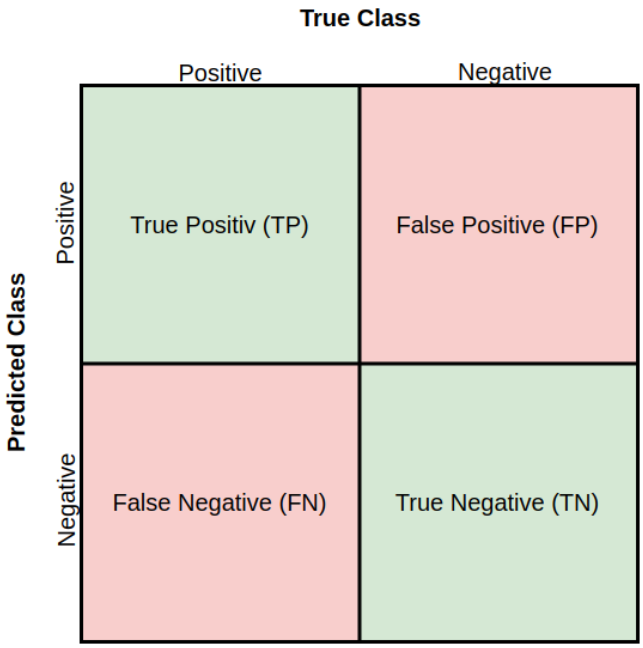


Figure 3: Confusion matrix for a binary classification task.

Accuracy considers the sum of True Positive (TP) and True Negative (TN) at the numerator and the sum of all elements at the denominator. TP and TN are the elements that are correctly classified by the model and go along the main diagonal of the confusion matrix. Recall measures how well the model finds the positive elements in the dataset by taking TP over the models' positive values (TP and False Negative (FN). Precision tells us how much we can trust the model by telling us how many positively predicted labels are positive, which is done by taking the TP over the TP and False Positive (FP) (elements that the models say are positive but really are negative). The F-Score is the weighted average between precision and recall. It finds the best trade-off between the two quantities. The highest value is 1, which indicates perfect precision and recall.

2.3 Classical Machine Learning Methods

2.3.1 Multiclass Logistic Regression

Logistic Regression (LR) is a statistical method using a logistic function to model different possible outcomes given a set of independent variables. This is done by adding parameters in the linear regression algorithm by adding multiple sigmoid

functions (logistic functions) together. The probability that x belongs to a class is $p(y = 1 \mid x) = q(x)$, and if it does not belong to a class the probability is $p(y = 0 \mid x) = 1 - q(x)$. This can be expressed with a Bernoulli distribution:

$$p(y \mid x) = q(x)^y (1 - q(x))^{1-y} \tag{13}$$

The sigmoid function squeezes the output to a linear equation between 0 and 1 by using the logistic function of fitting a straight line or hyperplane, $q : \mathbb{R} \to (0, 1)$. Figure 4 demonstrates how the sigmoid function given in Eq. 14 works.

$$\sigma(x) = sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{14}$$

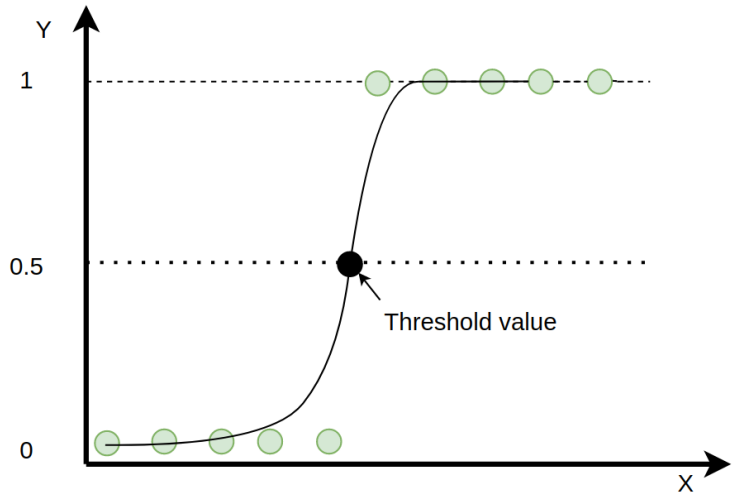


Figure 4: Sigmoid function for logistic regression. If a data point is higher than the threshold value at 0.5, it will be assigned to the class and if lower it will not be assigned to a class.

The Sigmoid function maps the input into the correct domain and the linear regression case becomes [15]:

$$\begin{aligned} q(x; \theta) = \sigma(w^T x + b) = \frac{1}{1 + e^{-w^T x + b}} \\ q(x; \theta) : \mathbb{R} \to (0, 1) \end{aligned} \tag{15}$$

Where $w = [a_0, a_1, ..]$ is the weighted vector determined from regression analysis and b is the bias.

The likelihood of a given model is given by the likelihood function, which is given by accumulating the Bernoulli distributions given in Eq. 13. From this, we can derive

the negative log likelihood function[15]:

$$\begin{aligned} L(\theta) &= \prod_{i=1}^{N} p(y_i \mid x_i, \theta) = q(x_i : \theta)^{y_i}(1 - q(x_i : \theta))^{1-y_i} \\ L(w) &= -\sum_{i=1}^{N} y_i \log(\sigma(w^T x_i)) + (1 - y_i) \log(1 - \sigma(w^T x_i)) \end{aligned} \tag{16}$$

The cost function can then be derived from the likelihood function on a dataset and the target probability of the output (Bernoulli distribution). To find the best model one has to solve the maximum likelihood estimator for logistic regression with optimisation techniques such as SGD:

$$\begin{aligned} \hat{\theta} &= \arg\max_{\theta} L(\theta) = \arg\min_{\theta}(-\log L(\theta)) \\ &\vdots \\ \hat{\theta} &= \arg\min_{\theta} \sum_{i=1}^{n} \log(1 + e^{w^T x_i + b}) - y_i \log(w^T x_i + b) \end{aligned} \tag{17}$$

This cost function is called the binary cross-entropy (BCE) loss. To avoid over/underfitting one can formulated the optimisation problem as a loss function and regularisation term:

$$\hat{\theta} = \arg\max_{\theta} \underbrace{\sum_{i=1}^{N} \log(p(y_i \mid x_i, \theta)}_{\text{Loss function}} + \underbrace{R(\theta)}_{\text{Regularisation}} \tag{18}$$

For a multiclass problem with K classes the logistic function (activation function) has to be substituted with the Softmax function:

$$q(x; \theta) = \sigma(w^T x_i + b) = \frac{e^{w^T x_i + b}}{\sum_{i=1}^{K} e^{w^T x_i + b}} \tag{19}$$

The softmax forces all values to be positive and then normalises to sum up to 1. The model will then have K outputs instead of 1. To compute the likelihood function, we take the product over all classes and find the maximum by implementing an optimisation method as in Eq 17:

$$L(\theta) = \prod_{i=1}^{N} \prod_{k=1}^{K} p_k(y_i \mid x_i, \theta) \tag{20}$$

2.3.2 Support Vector Machine

A Support Vector Machine (SVM) is used in supervised learning for regression and classification analysis. The goal of the SVM is to separate the input data points to their potential classes by creating a hyperplane in n-dimensional space[15]. To achieve this, one wants to maximise the distance to the data points and create a *support vector* with the data points with the smallest distance to the hyperplane. Figure 5 shows a support vector with 4 points (two circles and two triangles) laying on the hyperplanes. It shows the maximum-margin hyperplane for a linear SVM trained with samples from two classes. The hyperplanes can be written by the equation $y = w^T x - b$. Anything above $y = 1$ is in one class, and anything below $y = -1$ is in the other class. To maximise the distance $\frac{2}{\|w\|}$ between the hyperplanes, one wants to minimise the $\| w \|$. Where $ww^T = \| w \|^2$ denotes the normal vector. This is equal to solving the optimisation problem:

$$\arg\min_{w,b} \frac{1}{2} \| w \|^2 \tag{21}$$

with constraints that all data points must satisfy:

$$y = w^T x - b \geq 1 \tag{22}$$

To elevate the classification method to higher-dimensional space SVMs utilise a kernel function, $k(x, x') = \phi(x)^T \phi(x')$. The input space X consists of x and x'. The most used kernel functions are linear, polynomial, radial basis function (RBF) and sigmoid. The goal of the kernel function is to reduce computational complexity by preventing the need for complex transformations. It calculates a distance score between data points. The score is higher for closer data points and lower for points further apart. Then transform the data points to a higher dimensional mapping, making them easier to separate and label. The different kernel functions create different shapes of the hyperplane through the cloud of data points[16].

When there are overlapping class distributions, the SVM must allow some of the training points to be misclassified. Therefore, one introduces ξ_i for each data point, representing a penalty outside the correct margin. $\xi_i = 0$ if the data points are on or inside the correct margin boundary and $\xi_i > 1$ if misclassified. The optimisation problem that the SVM must handle then becomes [15]:

$$\arg\min_{w,b,\xi} \frac{1}{2} w^T w + C \sum_{i=1}^{N} \xi_i \tag{23}$$

Where $C > 0$ denotes the regularisation parameter, which decides how many data

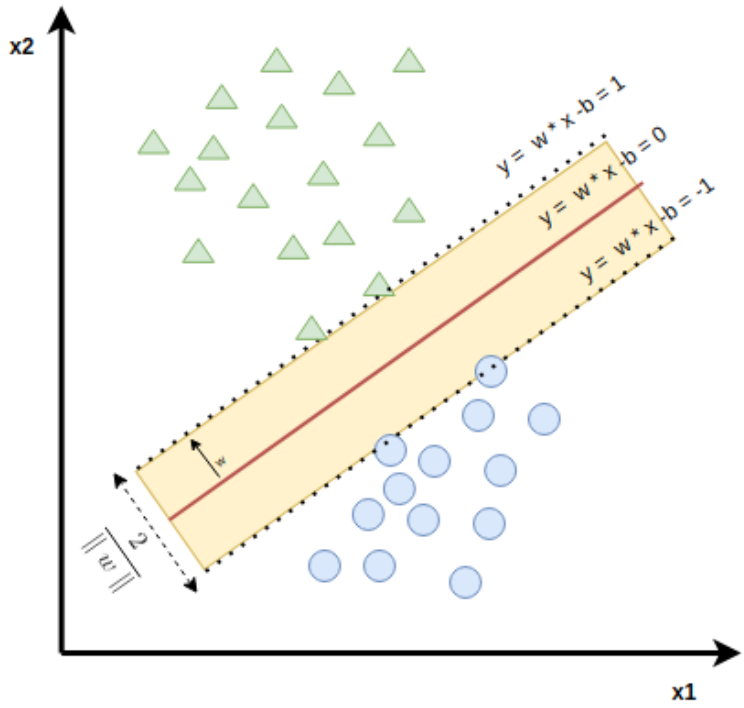


Figure 5: Hyperplane and margins for a linear SVM trained with samples from two classes

points can be falsely assigned. Introducing the kernel function $\phi(x_i)$ which transforms the input space, the classification constraint becomes

$$y_i(w^T\phi(x_i) + b) \geq 1 - \xi_i \tag{24}$$

In order to solve an optimisation problem with a given constraint one can use Lagrange multipliers and solve the Lagrange function[15]:

$$J(w, b, a) = \frac{1}{2} \parallel w \parallel^2 -C[\sum_{n=1}^{N} a_n[y_n(w^T\phi(x_n) + b) - 1]] \tag{25}$$

This is called the *dual problem* since one tries to minimise the parameters w, b while maximising the margins. Taking the derivatives with respect to w and b equal to zero one can eliminate w and b from $J(w, b, a)$:

$$\begin{aligned} w &= \sum_{n=1}^{N} a_n y_n \phi(x_i) \\ \sum_{n=1}^{N} a_n y_n &= 0 \end{aligned} \tag{26}$$

$$J(a) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m y_n y_m k(x_n, x_m) \tag{27}$$

With respect to the constraints $0 \leq a_n \geq C$ and $\sum_{i=1}^{N} a_n y_n = 0$. The cost function must then be implemented in a similar manor as in Eq. 17 introducing a regularisation term and optimisation with a technique as SGD.

For a multiclass classification with K classes, the problem is broken down into a total of $\frac{K(K-1)}{2}$ binary classification SVMs, and then the data points is classified according to the class with the highest score. The approach is called one-vs-one, but there exist other methods such as one-vs-rest[17].

2.3.3 Decision tree classifier

Decision Tree (DT) is a non-parametric supervised ML algorithm used for classification and regression. The selection process can be described as a sequence of binary selections corresponding to the traversal of a tree structure [15]. It is an iterative process where the data is split into partitions and further into separate branches (binary recursive partitioning). Figure 6 shows a DT with multiple classes.

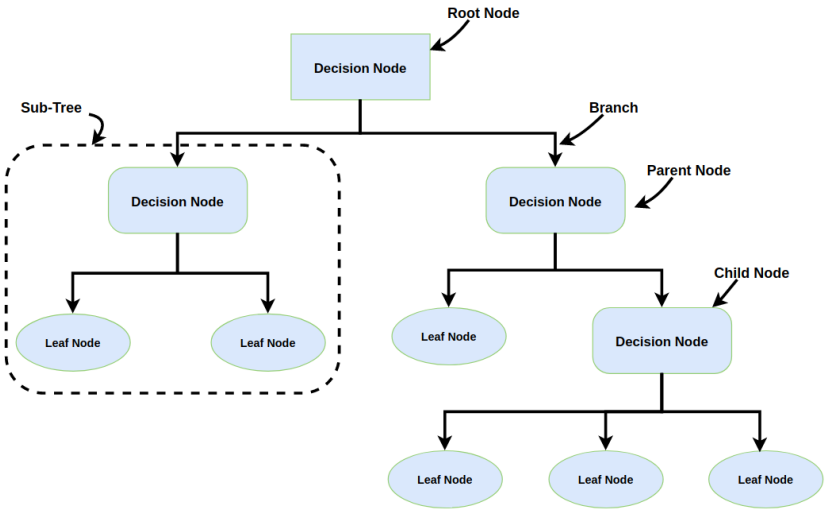


Figure 6: Illustration of Decision Tree with multiple classes.

Each node represents features, branch represents decision rule and leaves represent the models outcome, which can be any of the K classes concerned. The subnodes contain the best-fit attributes from the dataset. There exist a variety of algorithms used to create decision trees. Popular algorithms are Iterative Dichotomiser (ID) 3 and 4.5, CHAID and classification and regression tree (CART).

The CART algorithm has become quite popular and can be used for both regression and classification. The cost functions used in these algorithms are primarily Entropy and Gini index. The idea is to construct the binary tree using features and thresholds that yield the highest Information Gain (IG) at each node. Entropy is a measure of uncertainty or complexity in the subset. The Gini index is a measure of inequality or impurity in the sample at a given node. Mathematically these measures are provided by

$$\begin{aligned} \text{Entropy: } H(Q_m) &= -\sum_{i=1}^{K} p_{i,m} \cdot \log(p_{i,m}) \\ \text{Gini index: } H(Q_m) &= 1 - \sum_{i=1}^{K} p_{i,m}^2 \end{aligned} \tag{28}$$

Where $p_{i,m}$ is the probability that class i is in node m with samples N_m. To find the attributes that return the best IG, one has to calculate the entropy or Gini before and after the break into a new subtree:

$$IG(T, \theta) = H(T) - H(T \mid \theta) = H(T) - \sum_{i \in K} p_{i,\theta} H(i) \tag{29}$$

Where T is the sample space and $\theta = (f, t_m)$ consists of a feature f and threshold t_m. The idea is that $H(T)$ is a measure of uncertainty of a random variable T and by extracting the feature θ the uncertainty about T is reduced ($IG(T, \theta) > 0$). On the other hand, if T is independent of θ the Information Gain is zero ($H(T, \theta) = H(T)$). The cost function $H(T)$ is the impurity criterion attribute (entropy or Gini index) which help us compare the gain of each feature. To find the optimal impurity one chooses the parameter such as the IG is maximised:

$$\hat{\theta} = \arg\max_{\theta} IG(T, \theta) \tag{30}$$

This procedure continues through all the nodes for each level until the maximum depth is reached. If the tree keeps going until all the training data is classified this can lead to over-fitting which does not generalise well. It is therefore important to have a hyperparameter that decides the maximum depth. Other regularisation mechanisms can be minimal cost-complexity pruning or setting the minimum number of samples required at a leaf node [18].

2.3.4 Random Forest Classifier

A Random Forest classifier is a type of bootstrap aggregated Decision Tree by ensemble learning. The overall method is to build multiple decision trees at training

time and then output the class that most trees predict. The performance of random forest is often better than simple DT because it corrects its habit of overfitting by minimising the variance by considering multiple DT trained on different subsets of the same training data. Figure 7 illustrates a topology for a Random Forest algorithm with two classes and B trees.

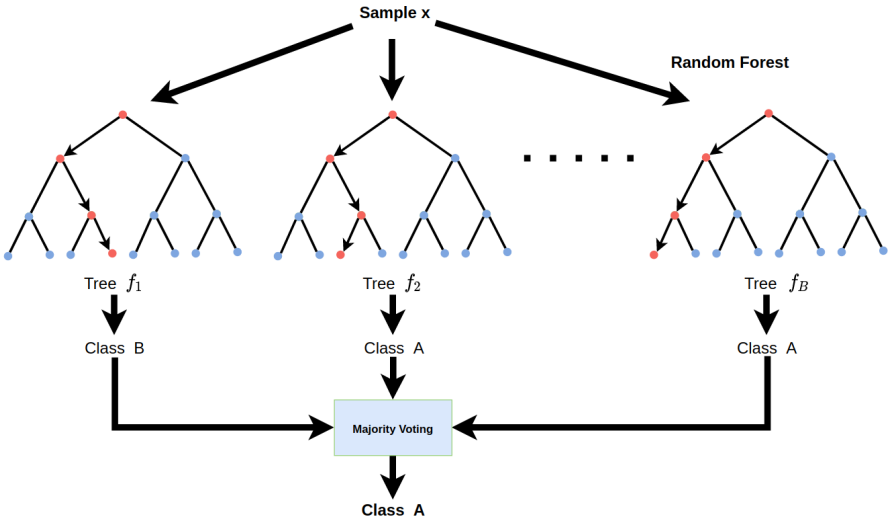


Figure 7: Random Forest with two classes and B Decision trees

The training algorithm takes random subsets of the training set $X = x_1, ...x_n$ and its' corresponding labels Y a total of B times ($b = 1, ...B$), and fits a random tree (f_b) to these samples X_b, Y_b. The prediction of a random sample is done by majority voting, where the most voted class predicted overall models f_b: will be given as the final prediction result. Random Forest uses both row sampling and column sampling (feature bagging) with replacement to decrease variance for the forest estimator. A subset of M features is taken randomly from the training set, and the features which give the best split is used to split the nodes iteratively. When choosing hyperparameters, one focuses on the maximum number of features and the number of trees in the forest. The size of the forest should be as large as possible, but more trees mean higher computational complexity. Additionally, after a critical point, the model will stop improving. The fewer features per tree will also give more reduction in variance, but also a higher bias.

2.4 Deep Learning

Deep Learning (DL) is a subset of Machine Learning, which is dominating the research field at the moment because of its general applicability and high performance. The intention of DL is to make computers think more like human brains by using structured Artificial Neural Network (ANN), also called feedforward Neural Network

(NN) or Multilayer Perceptron (MLP). The goal of ANNs is to approximate some function $f^*(x;\theta)$ by learning a mapping $y = f(x;\theta)$. A feedforward algorithm is implemented to map the data, which passes the input x to a function $f(\cdot)$ with θ (weights w and bias b), and subsequently, by training, determine the output y [10].

For a simple dense NN with one hidden layer the equation for the output becomes:

$$Y = W^{(1)^T} \cdot f(W^{(0)^T} X + B) \tag{31}$$

with weight matrix $W^{(l)}$, bias vector B and activation $f(\cdot)$. Where $W^{(l)} = \sum_j \sum_k w_{jk}^{(l)}$ is the weights between layer $l-1$ with j nodes and layer l with k nodes. Figure 8 illustrates a NN with two layers and two neurons($M = 2$).

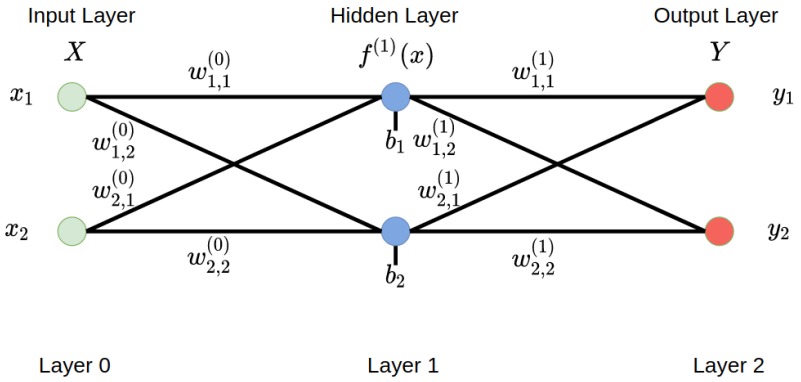


Figure 8: A simple dense Neural Network with M=2 neurons in its hidden layer.

2.4.1 Deep Neural Network

For a Deep Neural Networks (DNN) a network consists of several layers with functions $f(x) = f^{(L)} \cdot \ldots f^{(2)} \cdot f^{(1)}$. Where function $f^{(l)}$ is in layer l in a network of depth L. The dimensionality of each layer $f^{(l)}$ is determine by the M number of neurons, which computes its own activation value. In the hidden layers, $L-1$, the DL algorithm extracts features or building blocks and gives us a new representation of input $X = x_1, ...x_N$ which is subsequently used of classification at the output layer $f^{(L)}$. In Figure 9 a DNN is illustrated with $L-1$ hidden layers and $M = 4$ neurons in the hidden layers. The number of neurons is here fixed, but for other architectures it can vary throughout the hidden layers.

The representation learning is done through nonlinear activation functions $h(\cdot)$, to

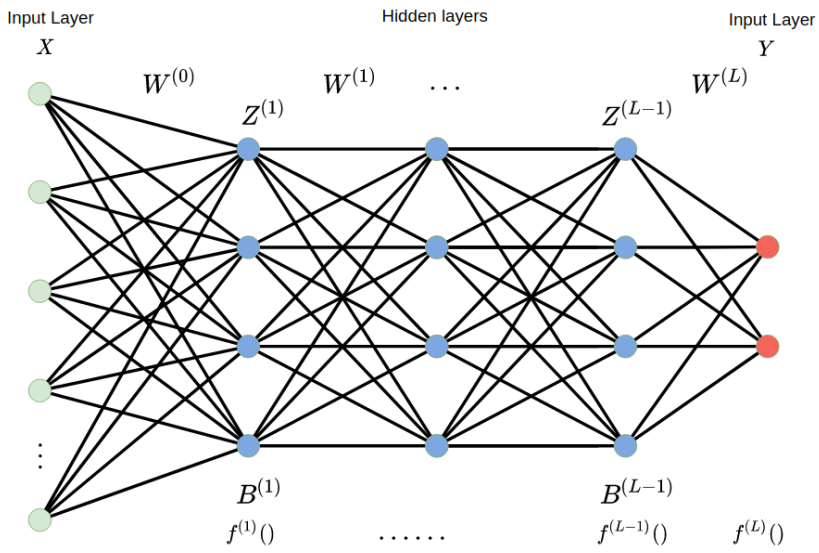


Figure 9: A DNN with $L-1$ hidden layers and $M=4$ neurons. The output of the model is donated with $\hat{y}_i$.

give an output vector for each layer (activation vector):

$$Z^{(l)} = h(W^{(l)^T} \cdot Z^{(l-1)} + B^{(l)}) \tag{32}$$

Where $Z^{(l)}$ is a matrix of size $j = 1, \ldots, M$ which corresponds to the dimension or width of layer l. Activation functions are chosen so that its derivative is simple and can be computed analytically [10]. Common activation functions are, among others, Rectified Linear Unit (ReLU), logistic sigmoid, hyperbolic tangent and softplus.

Finally, the output vector of the model $\hat{Y}$ can be computed using an appropriate activation function for the output of the final layer $Z^{(L)}$, depending on the nature of the data and desired output. For multiple classification problems each activation vector is transformed using a logistic Softmax function, as for Multiple Logistic Regression in Eq. 19:

$$\begin{aligned} \hat{Y} &= softmax(Z^{(L)}) \\ \hat{y}_i = softmax(z_i) &= \frac{e^{z_i}}{\sum_{j=1}^{M} e^{z_j}} \end{aligned} \tag{33}$$

To train the topology introduced above most modern NN classifiers are trained using the maximum likelihood principle (MLE), which, as mentioned before, is the same as minimising the cross-entropy. This shows how far the model is from the "ideal"

solution by minimising the cross-entropy between the data distribution $p_{data}(x, y)$ and model distribution $\hat{p}(x, y)$. The cross-entropy loss is given by [10] :

$$L(Y, \hat{Y}) = \sum_{k=1}^{K} -y_k \log(\hat{y_k}) \tag{34}$$

Here Y is a vector of the number of K classes that the model is trying to classify from the dataset. Where the cost function is found by taking an average over the training data:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{L} L(y, f(x_i; \theta)) \tag{35}$$

The goal of the training algorithm becomes to find the network parameters $\theta = (W, B)$ which minimises the cost function:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{L} \sum_{k=1}^{K} -y_k^{(i)} \log(\hat{y_k}^{(i)}) \tag{36}$$

The solution to this optimisation problem is using the aforementioned stochastic gradient descent method which estimates the error gradient for the current state of the model and then updates the weights of the mode by using backpropagation. Backpropagation allows the information from the cost function to travel backwards through the network to compute the gradients. This is done by implementing the chain rule concerning θ (weight and bias). Then the gradient is calculated one layer at a time backwards to avoid redundant calculations, and the learning rate decides how much each weights is updated.

Before training, it is essential to implement regularisation to ensure good generalisation for new inputs. There exist many forms of regularisation for DL. Most regularisation strategies are based on regularising estimators by trading increased bias for reduced variance. A typical approach is limiting the capacity of models by adding a norm penalty (L^1 or L^2) to the optimisation problem. The regularisation term is added to the cost function as shown in equation 3. Other strategies to avoid overfitting is data augmentation, early stopping, parameter sharing, ensemble methods, dropout and more [10].

2.4.2 Recurrent Neural Networks

Recurrent Neural Network (RNN) is a NN architecture that is well-suited for processing sequential data. These architectures have given good results in solving problems like Natural Language Processing (NLP), stock market prediction, spam filters, and more. The power of RNN comes from their ability to share parameters across

different parts of the model and unrolling. This gives the model the ability to remember what it has learned in previous time-steps and apply it in future predictions by sharing weights across several time steps. To achieve this the model uses a hidden internal state $h_t = f_w(h_{t-1}, x_t)$ that gets updated every time t it receives an input. The hidden internal state is then fed back into the model, as shown in Figure 10.

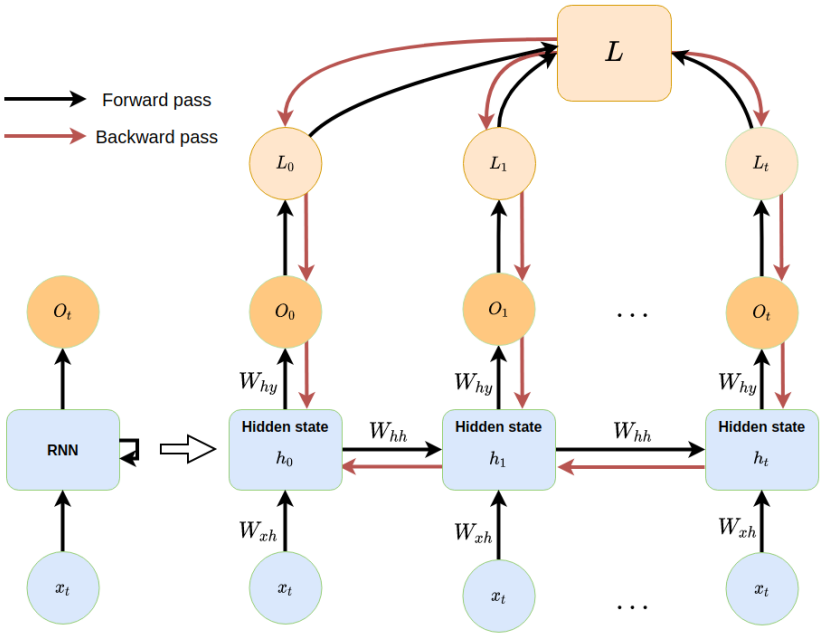


Figure 10: RNN with time steps t. The black arrow shows the forward propagation, and the red arrow shows the backward propagation.

The function that is used to update the hidden state is given by a hyperbolic tangent function *tanh*. This activation function bounds the states and tries to prevent the gradient from exploding. The equation for the hidden state and it's output is given in Eq. 37.

$$\begin{aligned} a_{(t)} &= W_{hh}^T h_{t-1} + W_{xh}^T x_t + B \\ h_t &= tanh(a_{(t)}) \\ O_{(t)} &= C + W_{hy}^T h_{(t)} \end{aligned} \tag{37}$$

The weights for connections input-to-hidden, hidden-to-output and hidden-to-hidden is defined as W_{xh}, W_{hy} and W_{hh} respectively, with bias B and C [10]. To be able to train the model over all time steps, it has to sum all the losses from each time step:

$$L(\{x_1, \ldots, x_t\}, \{y_1, \ldots, y_t\}) = \sum_t L_t \tag{38}$$

Computing the gradient concerning an initial hidden state h_0 involves many factors of W_{hh} in addition to repeated gradient computations. This can lead to exploding or vanishing gradients. The effect can be minimised by choosing a suitable activation function (ReLU), initialising weights as identity matrices, or using a more complex recurrent unit with gates to control what information is passed through.

2.4.3 Bidirectional RNN

Bidirectional RNN (BRNN) is a type of architectures which enables comparable look-ahead ability for RNN. This is done by running an additional layer that passes the information in a backward direction to more flexible process a dataset. This gives additional information for the classifier which in some cases can increase its performance. Figure 11 illustrates the additional hidden state. The forward and backward hidden state updates as follows:

$$\begin{aligned} \overrightarrow{a}_t &= W_{hh}^T \overrightarrow{h}_{(t-1)} + W_{xh}^T x_t + B \\ \overleftarrow{a}_t &= W_{hh}^T \overleftarrow{h}_{(t-1)} + W_{xh}^T x_t + B \end{aligned} \tag{39}$$

Then the forward and backward hidden states are concatenated to obtain the output layer $O_{(t)}$ as in Eq. 37, and the training process is the same as for classical RNN but with a extra backward hidden states.

2.4.4 Long Short Term Memory Network

Long Short-Term Memory (LSTM) is a RNN that uses a more complex recurrent unit or hidden cell to avoid the problem with exploding and vanishing gradient. The model network relies on gated cells to track information throughout many time steps to give additional memory in the decision process and less computational heavy to remember long sequences. Figure 12 shows the different components in the LSTM cell.

The first process in LSTM is the forget gate. The gate takes the current timestamp input x_t, and the previous cell state h_{t-1} as input and multiplies this with the relevant weight matrices before adding bias. The result is then passed along to an activation function σ. The activation function is a sigmoid function and it decides if the information is passed along or not depending on its binary value. For all 1's, the information is kept, and for all 0's, the information is forgotten. The next step is the input gate which stores relevant new information into the cell state. It is the same input as the previous step but multiplying with different weights to decide which values will be updated. Next, a *tanh* layer creates a vector of new candidates for the new candidate values in C_t. Then the LSTM cell selectively updates the cell state values by combining with the forget cell and input cell to form a new cell state

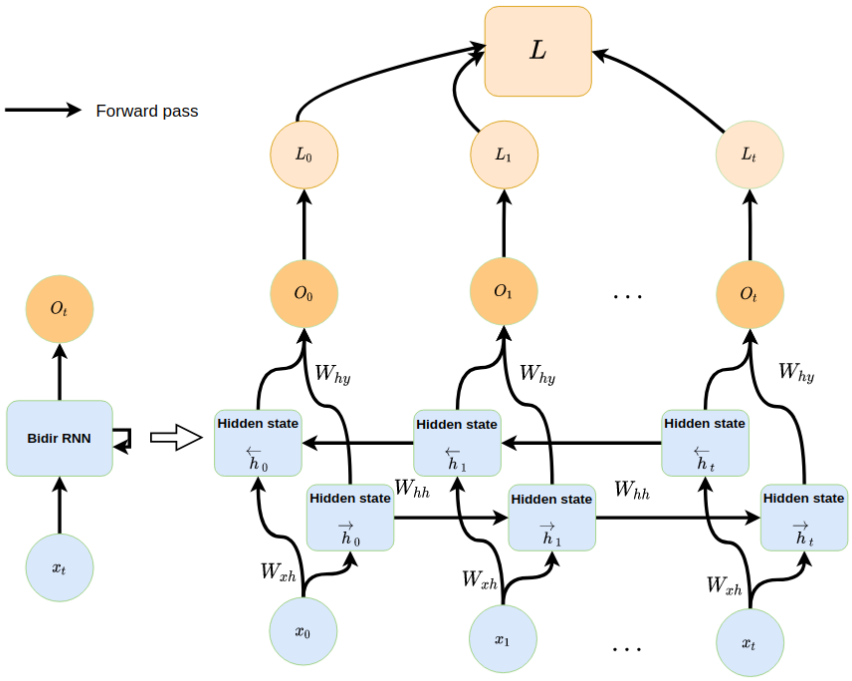


Figure 11: Bidirectional RNN has an additional backward hidden state $\overleftarrow{h}_t$ which gives the network additional memory.

C_t. Finally, the output gate controls what information h_t is sent to the next time step by sending it through a *tanh* function with previously used information h_{t-1}. Mathematically the cell memory and output at time step t can be expressed:

$$\begin{aligned} \text{Forget Gate: } f_t &= \sigma(X_t U_f + H_{t-1} W_f) \\ \text{Input Gate: } i_t &= \sigma(X_t U_i + H_{t-1} W_i) \\ \hat{c}_t &= \tanh(X_t U_c + H_{t-1} W_c) \\ I_t &= i_t \cdot \hat{c}_i \\ \text{Output Gate: } o_t &= \sigma(X_t U_o + H_{t-1} W_o) \end{aligned} \tag{40}$$

$$\begin{aligned} \text{Current Cell Memory: } C_t &= f_t \cdot C_{t_1} + I_t \\ \text{Current Cell Output: } H_t &= o_t \cdot \tanh(C_t) \end{aligned} \tag{41}$$

Where U and W are the corresponding weights that need to be optimised with the left out bias. Training this model happens in the same process as the classical RNN but includes the different cell states. First doing the forward propagation to create a cost function before doing backpropagation to calculate the optimal learning parameters:

$$f_t(\theta) \rightleftharpoons C_t(\theta) \rightleftharpoons H_t(\theta) \rightleftharpoons J(\theta) \tag{42}$$

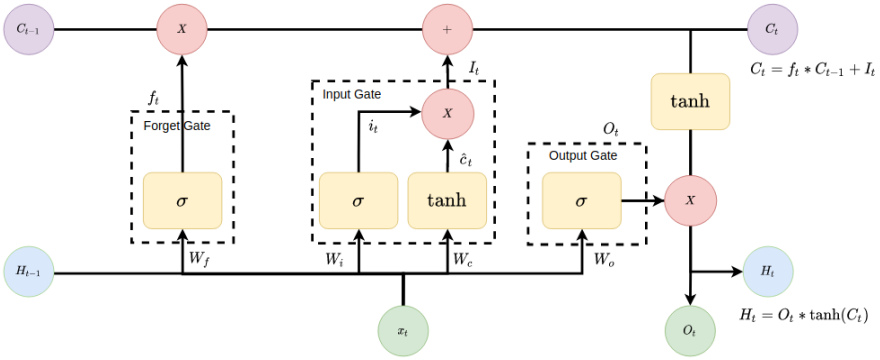


Figure 12: The hidden cell of an LSTM with the forget gate, input gate and output gate.

3 Description and implementation

This section presents the dataset that is used, problem framing and how the classification system was implemented and trained. It also compares the differences in implementation between classical ML and DL classifiers.

3.1 Dataset WISDM

The data collection used for this paper was gathered from the Wireless Sensor Data Mining (WISDM) lab at the Department of Computer and Information Science of Fordham University. The WISDM dataset was collected by using a cell phones accelerometer through controlled laboratory conditions[19]. There are other relevant datasets, e.g. from the University of California Irvine (UCI) [8]. Still, the WISDM dataset was chosen because of its lack of pre-processing and transparent data gathering. The WISDM dataset consists of 1 098 207 samples from 36 participants (users) stored in time (ns), acceleration values (ms^{-2}) in x, y and z-axis, as well as activity label. The mean sampling rate is 20 Hz (1 sample every 50ms), and the acceleration values range between -20 to 20. In this dataset, six activity classes were chosen: Walking, Jogging, Walking Upstairs, Walking Downstairs, Sitting and Standing. These activities were selected since they represent some of the most common activities performed in humans daily lives. The majority of samples are done while participants are walking or jogging, and the minority while sitting or standing. The distribution of samples is shown in figure 13. The activities distribution in number of samples is given as follows. Walking consists of 424 400 (38.6%), jogging consists of 342 177 (31.2%), walking upstairs 122 869 (11.2%), walking downstairs consists of 100 427 (9.1%), sitting consist of 59 939 (5.5%) and standing consists of 48395 (4.4%). The data has a disadvantage in it's distribution which can lead to a

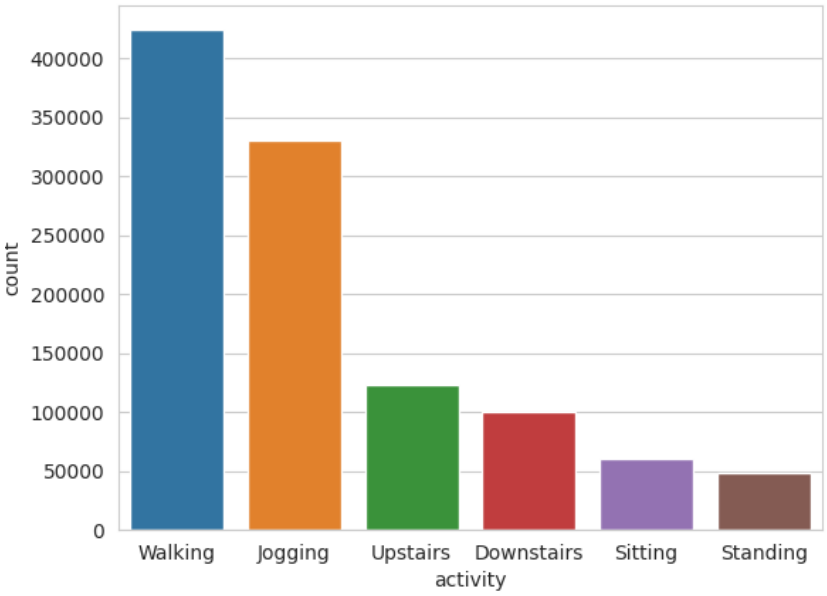


Figure 13: Total number for samples for each activity.

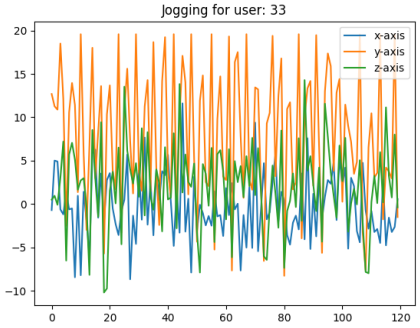
bias for the machine learning methods.

Figure 14 shows the raw signal of all the different activities given by participant number 33. The blue, orange and green lines represent the acceleration values in the x-axis, y-axis and z-axis, respectively. A value of $10 = 1\text{g} = 9.81\ \text{m/s}^2$ and a value of 0 means no gravitational power in that direction. The acceleration is relative to the gravitational acceleration towards the centre of the Earth, so when the phone is on a flat surface, the vertical axes (x-axis and z-axis) will register ± 10 [19].

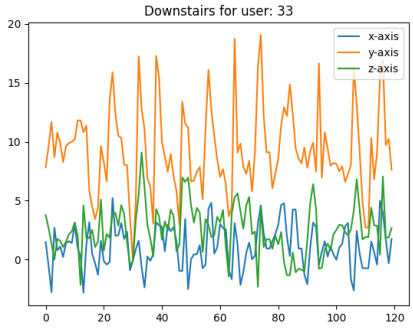
3.2 Problem framing

When analysing a labelled dataset of human activity, one should consider different approaches regarding the goal of the classification algorithm. The overall goal is to classify an activity, but this can, for instance, be done cross-subject or per subject. Per subject classification means splitting the data after subject. Training on some subjects and testing on others. Cross subject means randomising the data and train and test on all subject data. Cross subject analysing might be more desirable but harder to do if the data is unbalanced or the activities are challenging to separate. Per Subject might be easier to model and adapt to new applications. Other considerations can be how to frame the data during modelling. The two main approaches

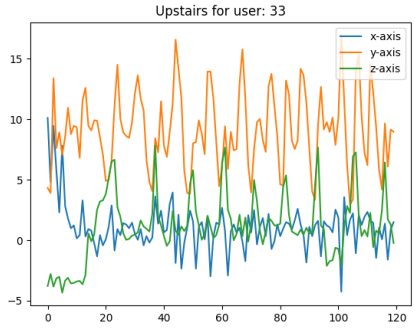
(a) 6 Seconds accelerometer data where the subject is walking.



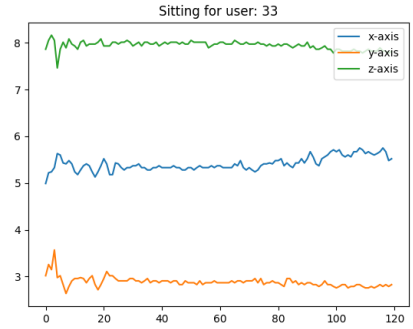
(b) 6 Seconds of accelerometer data where the subject is Jogging.



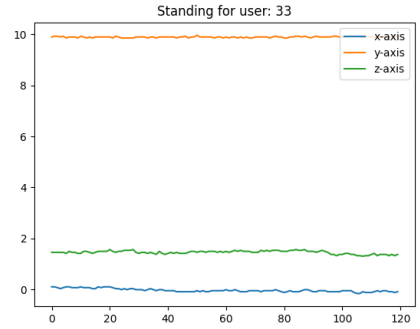
(c) 6 Seconds accelerometer data where the subject is walking downstairs.



(d) 6 Seconds of accelerometer data where the subject is walking upstairs.



(e) 6 Seconds accelerometer data where the subject is sitting.



(f) 6 Seconds of accelerometer data where the subject is standing.

Figure 14: Raw data from each activity for user_id 33

include segmentation of activities and sliding window. When segmenting the activities one has to pre-segment data to each activity. Extract features and train the model on each activity. Sliding Windows is done by splitting the data into windows, with or without overlap, and windows activity is labelled as the most frequent activity. Then one has to extract features and train model over each window. Which approach is chosen depends on the datasets layout and data gathering. For this task we have split the data cross-subject to avoid introducing any bias. Also, the data is extracted by the use of a sliding window since it is easier to implement and is the more popular approach for similar problems.

3.3 Implementation overview

In this paper, the HAR system waswas implemented using cross-subject with a sliding window for the WISDM dataset. Figure 15 show the entire process that is implemented from the data source to the final classified activity.

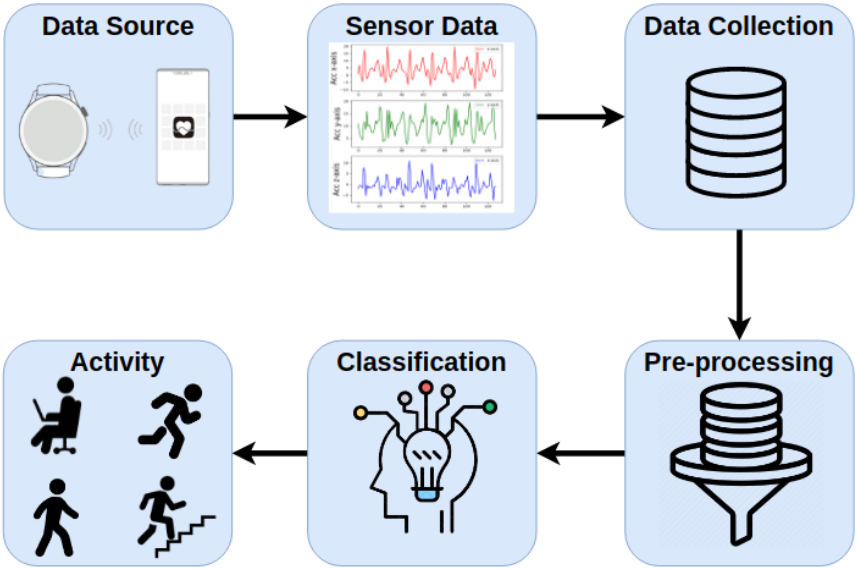


Figure 15: Block diagram of the proposed methodology for activity recognition.Data gathering from a wearable device, accelerometer data is stored and pre-processed before used to train a classifier.

The accelerometer data from WISDM dataset is stored in a suitable data collection. There are many formats to choose from when collecting the data. The most common format is the CSV format. This format has its limitations regarding slow query time and difficulty storing large data files efficiently. There exists alternatives to the CSV format, e.g., SQLite, HDFs, or Parquet, but these also have their own limitations.

Despite its drawbacks, the CSV format is the preferred format because of its simple and intuitive access to data and its broad adoption in Machine Learning. Table 1 shows an example of two random rows from the data collection in CSV format.

Table 1: CSV-file format

Index	user_id	activity	timestamp	x-axis	y-axis	z-axis
1	1	Walking	4991972333000	6.85	7.44	-0.50
99995	4	Sitting	3679282298000	-7.35	-3.91	-5.13

The next step is pre-processing which remove unnecessary components from the data to make it more suitable for classification. The process implemented for the system used in this paper is introduced in Section 3.4. Afterwards the different classification algorithms are implemented. The procedure for implementation is presented in Section 3.6. After classification the final step in Figure 15 is training the algorithm and classifying activity.

The methodology used for the different classifiers is quite similar, but there is a clear distinction between the classical ML and DL in the feature extraction. For DL classifiers its the algorithm who extracts the features itself. While for the classical ML classifiers the features must be extract before the data is introduced to the algorithm. Therefore for classical ML one has to use additional resources to find and optimise features to give the best possible representation of the data. The feature extraction process is introduced in Section 3.5.

3.4 Pre-processing

Pre-processing is essential before introducing the data to classification algorithms to ensure or enhance performance. The phrase Garbage In, Garbage Out (GIGO) explains the reason well. If we submit bad data to the algorithm, we can expect a bad result. Therefore we have to pre-process the data. This can be done using simple methods such as removing irrelevant or redundant information, i.e. out of range values, impossible combinations of data or Not a Number (NaN). Other methods can be more complex such as filtering, which removes noise or unwanted data [20]. In figure 16, the steps taken in the pre-processing is illustrated.

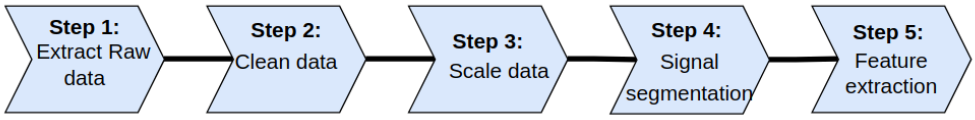


Figure 16: Pre-processing steps done on the WISDM dataset.

The first step is to extract the dataset into a wanted format. As mentioned above the CSV format is implemented. The next step is to clean the data of irrelevant information such as NaN values and unsuitable timestamp values. These steps is necessary to avoid errors throughout the dataset and ensure continuous and relevant values. The third step is one of the most common forms of pre-processing, normalising the data to ensure performance. Here normalisation is refereed to as ensuring that all the values are within the same scale of [-1,1], which is a simple rescaling of the input variables as presented in section 2.1. It can be discussed that this step is irrelevant since the accelerometer values already range between [-20, 20]. Still, it's done as a cautionary step since differences across input variables may increase the difficulty of the problem being modeled in some machine learning models [6]. After normalising the data the next step is signal segmentation. Signal segmentation is a technique to introduce a sliding window to the dataset. It divides the sensor data into smaller time window segments which is more compatible with the classification algorithm. The sliding window is essential to create suitable input values and to easily extract features for the classical ML methods. The sliding window process is illustrated in Figure 17. Choosing an optimal window length can be difficult since the length of each activity varies without dependence on the user or activity. The Figure 18 shows the distribution of each activities length in the dataset. It shows that the smallest duration of an activity is about 110 samples, which is 5.5 seconds with a sampling frequency of 20 Hz. A window duration bigger than this can lead to a worse performing classifier. The chosen window duration was 2.5 second time window (20 Hz $\cdot$ 5 sec = 50 samples) with 50% overlap to ensure that all activities were well represented. In other studies a longer window of up to 10 seconds has been used [2][8][21].

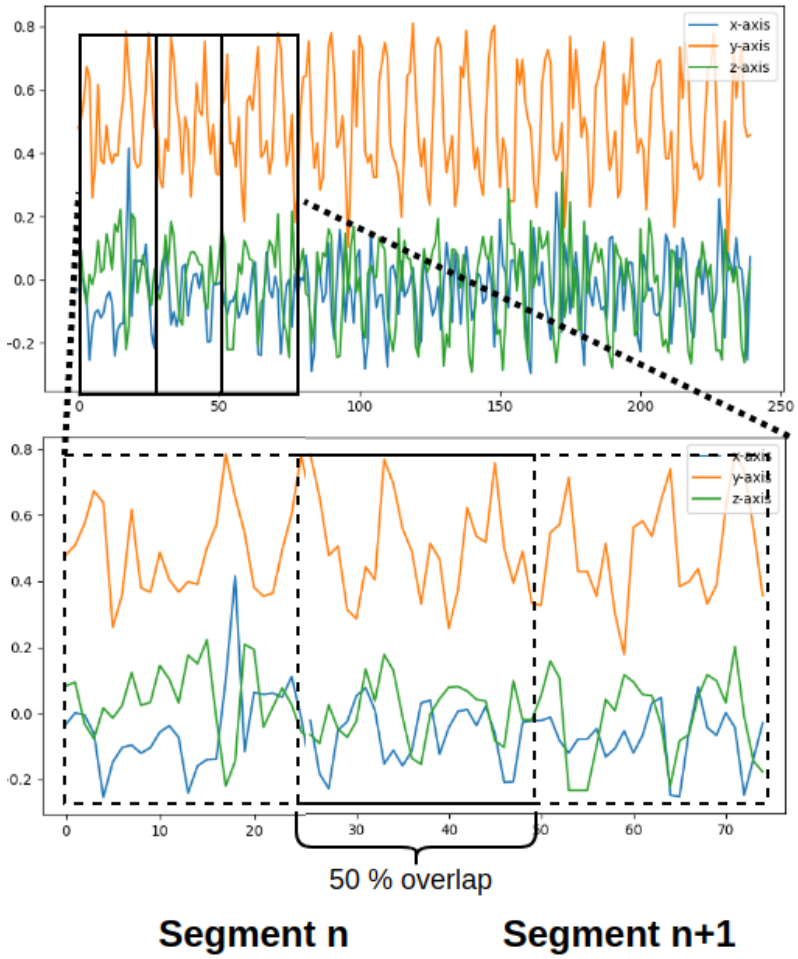


Figure 17: Signal segmentation (windowing) with an overlap of 50% and 50 samples.

3.5 Feature extraction

Feature extraction is to derive relevant values (features) from the signal segmentation. These features are typical signal characteristics from the raw sensory data from each window. From each window, a feature vector is obtained by calculating variables from the time and frequency domain. The features selected are expected to be representative of the information contained in the extensive input data. To reduce the dimensionality but still capture the essence of the input information one can utilise Principal Component Analysis (PCA). The method reduces the input dimensionality by removing correlated features by maximising the variance of the features. Since time series feature engineering can be quite time-consuming and challenging, we have in this paper relied on endorsed features from other papers [2][8]. There also exist helpful Python packages such as TSFRESH, which can sim-

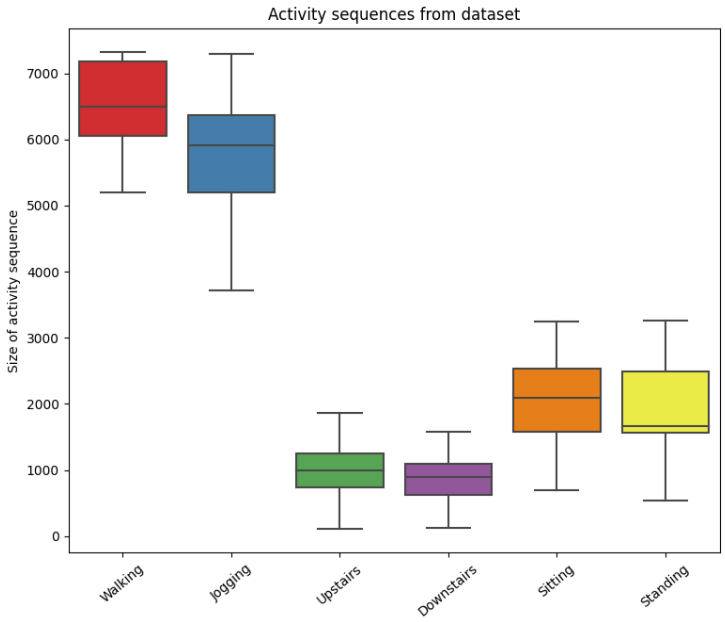


Figure 18: Box plot of duration for each activity in the dataset. The smallest duration was found to be 108 samples for the activity walking upstairs

plify the process further[22]. Table 2 shows the different features extracted from each window to use in the classical machine learning methods.

Table 2: Selected features

Domain	Selected Features
Time-domian	Mean, median, standard deviation, min, max, distance between max-min, average absolute distance from mean and median, values above mean, interquartile range, negative and positive count, average resultant (SVM), signal magnitude area (SMA), max and min indexes of given window, tilt angle and correlation
Frequency domain	Signal entropy, signal energy, peaks count, skewness, kurtosis

There was a total of 90 features extracted from the 3-axis of accelerometer data. In Appendix C.4 the feature extracting is demonstrated using Python code.

3.6 Implementing classification algorithms

The problem at hand is to solve a supervised classification problem with six classes. For such problems, there exist state-of-the-art classification methods. Table 3 shows the classification models implemented for our solution. These methods are well

documented to work for supervised learning and activity recognition tasks [2][4]. The classical classifiers are Logistic Regression (LR), linear Support Vector Classifier (SVC), Decision Tree (DT) and Random Forest (RF). For the DL classifiers there was implemented a four layer Deep Neural Networks (DNN) and two Recurrent Neural Network (RNN)s, a two layer Long Short-Term Memory (LSTM) and one layer Bidirectional Long Short-Term Memory (BLSTM).

Table 3: Classification algorithms

Type	**Classification method**
Classical machine learning	LR, SVC, DT and RF.
Deep Learning	DNN, LSTM and BLSTM

The classical ML and DL code differ mainly in which algorithms are implemented and the input data (features or only accelerometer data). The implementation process can be summarised into 5 steps:

1. Analysing and pre-processing of the data
2. Separate data in training, validation and test sets
3. Implement classification models
4. Train model on training data and validate on validation data
5. Evaluate model against test data

The analysing of the data was introduced in section 3.1 and the pre-processing steps was introduced in Section 3.4. The next step is to separate the data in training, validation and test set. The data must be selected randomly not to present any strange dependencies since the data is unbalanced between users and activities. This is avoided by using a cross-subject splitting with a random separation ratio of 80 % training, 10 % testing and 10 % validation data. There exists other methods of splitting and training the data, such as Leave-one-Subject-out (LOSO) cross-validation [21], but since the dataset is quite large we use a randomised cross-subject splitting without introducing any bias to the algorithms. Step 3 is implementing the classification models. The classification algorithms are written in Python and depend primarily on Scikit-Learn and Keras running on top of Tensorflow [23][24]. The implementation of the classification models can be found in Appendix C.1 and C.2, for classical ML and for DL, respectively.

The fourth step in the process is to train the model on the training and validation data. The training process is demonstrated in Figure 19.

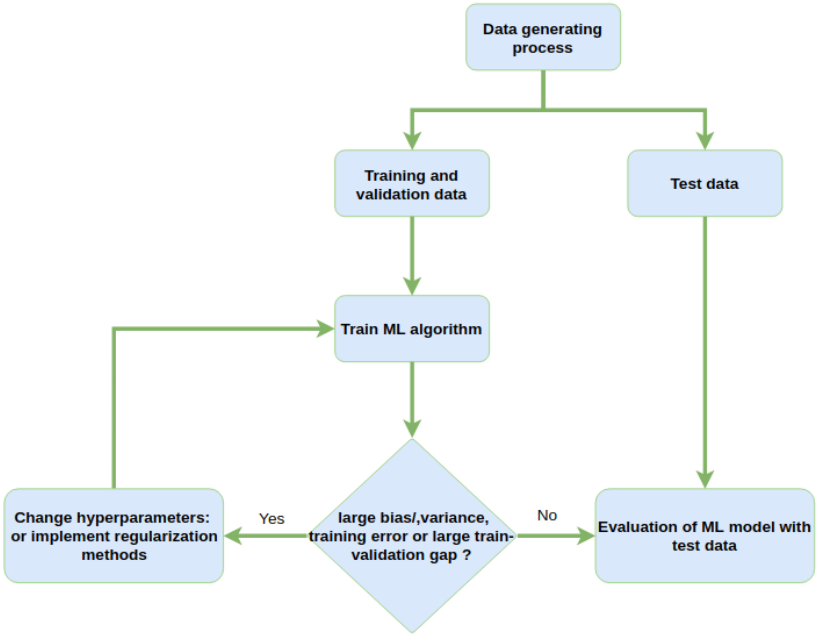


Figure 19: This figure illustrates the implementation and training process for ML algorithms. The model must be tuned if there is a significant training error, a sizeable train-validation gap, or a significant bias or variance.

The goal during the training process is to optimise the accuracy on the validation set while having a low bias/variance and keep the training-validation gap at a minimum. Choosing hyperparameters when implementing classification algorithms is essential to ensure good performance. Self-tuning can be a cumbersome and time-consuming task, therefore hyperparameter optimisation is implemented to find the best suitable parameters for each machine learning method. Both the classical classifiers and the NN classifiers uses a for of Bayesian optimisation. This was accomplished using the using the KerasTuner API[25]. There is also implemented other regularisation measures such as early stopping which stops the learning process after 5 epochs if there is no significant increase in the validation accuracy. In addition there is implemented dropout layers for DL classifiers to avoid overcapacity in the models. The chosen optimisation algorithm was Adam which incorporates adaptive learning rates and momentum for efficient learning.

After finding a satisfactory performance and training-validation gap we can move to the final step of evaluation. In the evaluation step, we test the ML models on the test data which we have left out of the training process to ensure that the results are unbiased.

4 Results and Discussion

In this section the results are introduced and discussed. Firstly the results from the hyperparameter tuning will be introduced for each model and then discussed. Afterwards the result on the test set for each classification algorithm will be presented and compared against each other. Lastly the preferred classifier will be compared to other state-of-the-art solutions.

4.1 Hyperparameter Tuning

During step 4 in the implementation process the classification algorithms are subjected to a hyperparameter optimisation to find the best suitable model parameters. Table 4 and Table 5 demonstrates which hyperparameters that where tested and their corresponding optimal parameter. Hyperparameter optimisation was used to ensure that the best regularisation parameter was chosen efficiently and effectively. It's important to chose good hyperparameters to ensure satisfactory regularisation. There are, as mentioned in section 2.2.1 and 2.4, other methods of regularisation than the once implemented here, but here we have focused on L1 and L2 regularisation, learning rate, dropout techniques, and early stopping. To avoid over/under-capacity and increased bias of the model there could also be introduced some form of dataset augmentation to balance the dataset, ensemble methods or adversarial training.

For logistic regression and SVM the prioritisation was the norm regularisation terms (L1 and L2) which penalises large weights and biases. For the tree-based models the hyperparameter tuning is done with respect to the maximum depth of each decision tree and the total number of trees in the forest. For further optimisation it can be useful to test with other regularisation terms, such as tolerance for the stopping criteria or different learning rates. However, when considering the scope of the project and possible gain, this was not implemented.

Table 4: Hyperparameter optimisation for Classical ML

ML model	**Hyperparameter Tested**	**Optimal Parameters**
Logistic Regression	Penality: {L1, L2} C: {0.01,0.1, 1, 10, 20, 30}	Penalty: L1 C: 20
Linear SVC	Penality: {L1, L2} C: {0.125, 0.5, 1, 2, 8, 16, 30}	Penalty: L1 C: 8
Decision Tree	Max depth: 1-50 with steps of 1	Max depth: 7
Random Forest	Max depth: 1-50 with steps of 1 Max Trees: 1-201 with steps of 25	Max depth: 21 Max Trees: 176

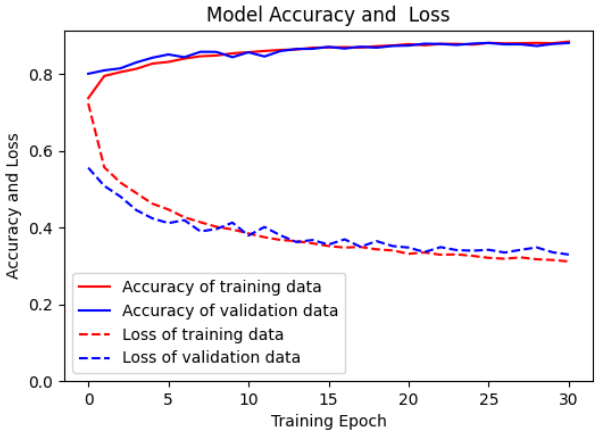
For DL models a hyperparameter optimisation was done moderately with respect

to the regularisation term of the cost function, learning rate and width of the layers (number of neurons). The number of layers was found by experimenting with the neural networks after the hyperparameter optimisation. Increasing the layers beyond the chosen values had little to no impact on the performance, only on the training time. Appendix A illustrates the model architecture for the NN. In Table 5 the hyperparameters are given for the different DL classifiers.

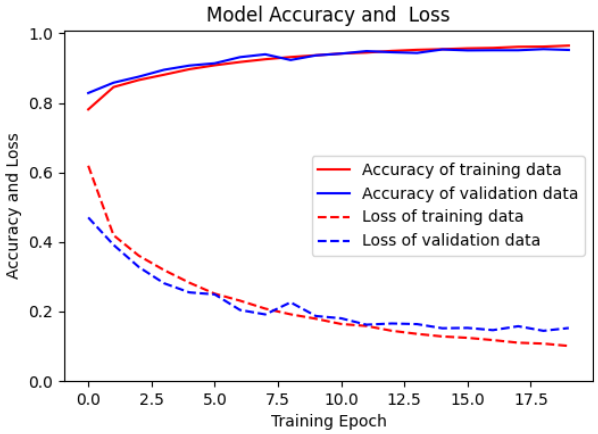
Table 5: Hyperparameter optimisation for Neural Networks

ML model	**Hyperparameter Tested**	**Optimal Parameters**
DNN	Penalty: {L1,L2} C: {0.01,0.1, 1, 10, 20, 30} Learning rate: {3e-5, 3e-4,3e-2, 0.001,0.01} Neurons:{10, 20, 30, 40, 50, 60, 70}	Penalty: L2 C : 0.01 Learning rate: 0.001 Neurons: 50
LSTM	Penalty: {L1,L2} C: {0.01,0.1, 1, 10, 20, 30} Learning rate: {3e-5, 3e-4,3e-2, 0.001,0.01} Neurons:{10, 20, 30, 40, 50, 60, 70}	Penalty: L2 C : 0.01 Learning rate: 3e-4 Neurons: 60
BLSTM	Penalty: {L1,L2} C: {0.01,0.1, 1, 10, 20, 30} Learning rate: {3e-5, 3e-4,3e-2, 0.001,0.01} Neurons:{10, 20, 30, 40, 50, 60, 70}	Penalty: L2 C : 0.01 Learning rate: 3e-4 Neurons: 60

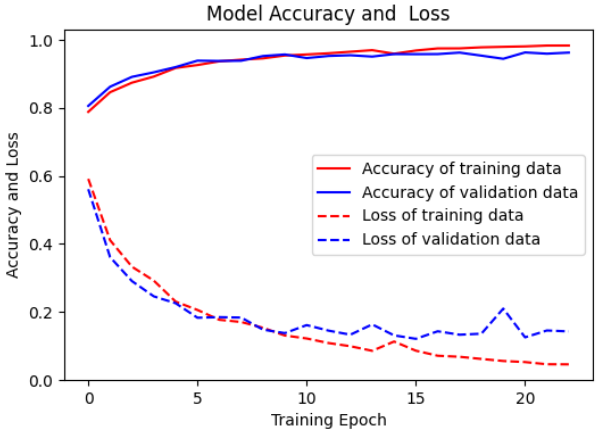
The preferred penalty was L2 which is the most common to use for NN [26]. Choosing the width of a networks should be done in correspondence with the input and output layer. Each NN model was trained with a window of 2.5 seconds and 50% overlap which gives a total input matrix of size 3 dimensions with 50 samples per window with a total of 43413 windows. That gives a input layer of 50 nodes and the output layer consists of 6 nodes for each of the classes. It's therefore not surprising that the optimal width of the models was found to be 50 for the DNN and 60 for LSTM and BLSTM. Finding the optimal learning rate can be seen as redundant since we have implemented *Adam* with adaptive learning rate and momentum. The adaptive algorithm self-tunes the learning rate during training which reduces the need for tuning, since the default value is sufficient for some problems. Nevertheless, there can be situations where tuning off the learning rate can lead to enhanced performance. Neural Network (NN) are complex and are prone to overfitting, local minima and long-term dependencies such as exploding or vanishing gradients. It is therefore important optimise vital hyperparameters. From Table 5 we see that for the DNN the default value of 0.001 was sufficient, but for the RNN models the optimal learning rate was found to be $3 \cdot 10^{-4}$. One can argue that the hyperparameters for the momentum also should be tuned, but it is uncommon tune this parameter since it's consider to be rather robust.



(a) Training history of DNN model with Training accuracy: 87%, Validation accuracy: 86%, training loss: 0.338 and validation loss: 0.361



(b) Training history of LSTM model with Training accuracy: 97%, Validation accuracy:95%, training loss: 0.084 and validation loss: 0.168



(c) Training history of BLSTM model with Training accuracy: 98%, Validation accuracy: 95%, training loss: 0.054 and validation loss:0.17

Figure 20: Training history for the deep learning classifiers.

When the hyperparameter tuning is complete one has to verify that the models has achieved a low error/variance as well as a low generalisation gap (training-validation gap). In Figure 20 an example of the training history for DL classifiers are presented. From these figures we see that the training-validation gap is relatively low and that the accuracy is sufficient. The DNN has the highest loss and the lowest accuracy, but the gap is quite small so there is no indication of overfitting. For the RNN models their performance is quite similar, but there is a higher gap between the training and validation loss for the BLSTM model. This can indicate that there is some overcapacity for this model compared to LSTM model. Still, the gap is relative low compared with the model performance and its suitable to be tested on the test dataset.

4.2 Results on test data

Table 6 and Table 7 presents the results of the activity recognition approaches chosen in this paper. The tables indicate that the Deep Learning (DL) algorithms are performing higher or the same as the classical classifiers on almost all evaluation metrics and recognition rates. The classical ML model with the highest accuracy is the Random Forest (RF) classifier with an average accuracy of 89%. The classifier with the lowest accuracy is the Decision Tree (DT) classifier with an average accuracy of 80%. For the DL classifiers the highest accuracy is given by the RNN models with an average accuracy of 95% and the lowest average accuracy of 87% for the DNN model. The DNN model performs similar to the best performing classical classifiers.

Table 6: Comparison of evaluation metrics with respect to different classifiers.

Classification Model	Precision	Recall	F1-score	Accuracy	Error
Linear Regression	88%	86%	86%	86%	14%
Linear SVM	87%	86%	86%	86%	14%
Decision Tree	78%	80%	78%	80%	20%
Random Forest	89%	89%	89%	89%	11%
DNN	86%	87%	86%	87%	13%
LSTM	95%	95%	95%	95%	5%
BLSTM	95%	95%	95%	95%	5%
Total Average:	88.3%	88.1%	87.7%	88.1%	11.9%

The RNN models have the highest score on every evaluation metric, but surprisingly the BLSTM does not perform better than the LSTM. They both use parameter sharing as an additional regularisation term, but the intention of the bidirectional RNN layer was to give additional information and henceforth increased accuracy. By increasing the number of layers the performance increased negligibly compared

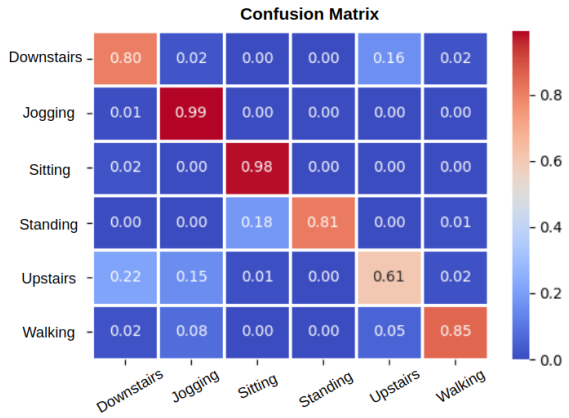
to the immense increased training time. There could be multiple reason for the similar performance of the LSTM and BLSTM. One reason could be the size of training-validation gap for the BLSTM ($0.17 - 0.054 = 0.116$) compared to the LSTM ($0.1683 - 0.0840 = 0.0843$). The difference is rather small so it is believed it is insignificant. Another possibilities could be that because of the unbalanced dataset the RNN algorithms have reached an "upper limit" that can only be solved with data augmentation and balancing the dataset.

Table 7: Comparing the recognition rate for different classifiers with respect to each activity.

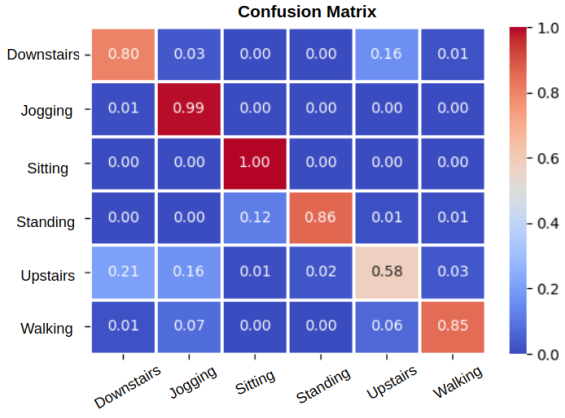
Activity	LR	SVM	DT	RF	DNN	LSTM	BLSTM
Downstairs	80%	80%	60%	77%	48%	83%	85%
Jogging	99%	99%	99%	99%	97%	99%	99%
Sitting	98%	100%	90%	99%	98%	99%	99%
Standing	81%	86%	95%	98%	99%	100%	99%
Upstairs	61%	58%	17%	63%	50%	83%	83%
Walking	85%	85%	85%	90%	95%	98%	98%

Table 7 demonstrates the percentage of True Positive for each activity given the specific classifier. The table indicates that out of the 6 activities there are two activities, walking up and down stairs, are most difficult to recognise. The highest recognition rate for downstairs is 85% by the BLSTM model and the lowest is 48% by DNN. For upstairs the highest rate is for both the RNN models with 83% and the worse is 17% by the DT classifier, which is as bad as random picking. To get an indication of why the models are underperforming on these activities we can look at the characteristics of their raw output in Figure 14. Figure 14 shows that walking up and down stairs have similar activity pattern and can in some instances look similar to jogging or walking. In Figure 13 we saw that the dataset was unbalanced in the favour of walking and jogging, which can have an effect on the bias of simpler classification algorithms such as DT. This is supported by the confusion matrices of the classification models given in Figure 21 and 22.

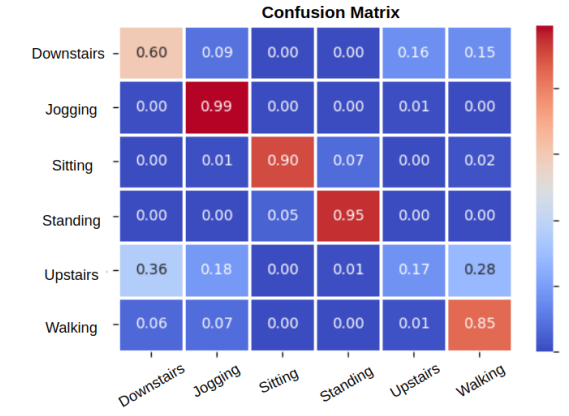
In Figure 22a we see that for the DNN model the misclassification of walking up and down stairs are mostly classified as walking (33%-29%) or by each other(19%-10%). The DT classifier has the reversed patter of the DNN, and has problem distinguishing the difference of walking up or down the stairs. The other classical classifiers have the same misclassification pattern, but some in favour of jogging instead of walking, see Figure 21. The RNN models also misclassify these two activities but to a lesser extent. It's seems that the additional information that the RNN gains works better for similar activities. The performance of the classical classifiers might be enhanced by adding additional features which provides more information about the segments. This was not explored in the papers [2] and [27] where the features in Appendix



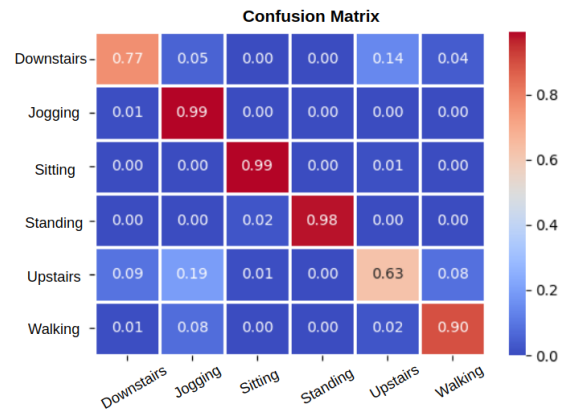
(a) Linear Regression classifier with average accuracy of 86% on test dataset.



(b) Linear Support Vector classifier with average accuracy of 86% on test dataset.

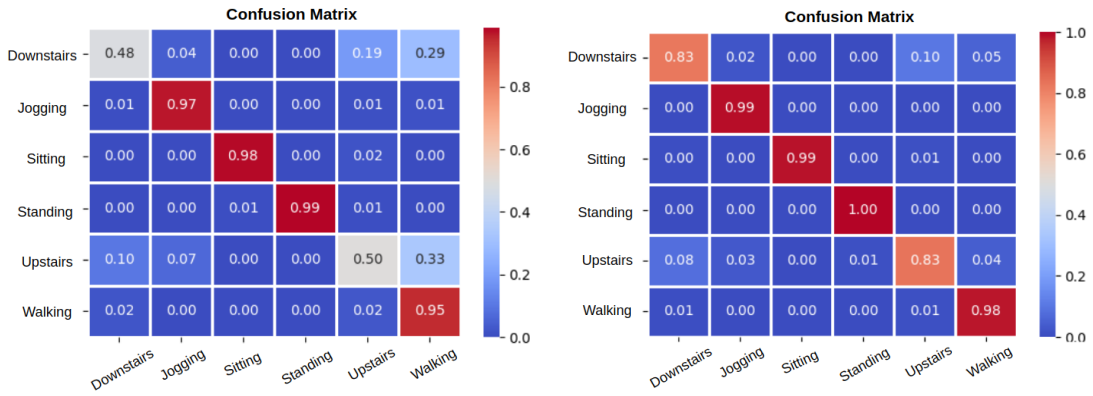


(c) Decision Tree classifier with average accuracy of 78% on test dataset.



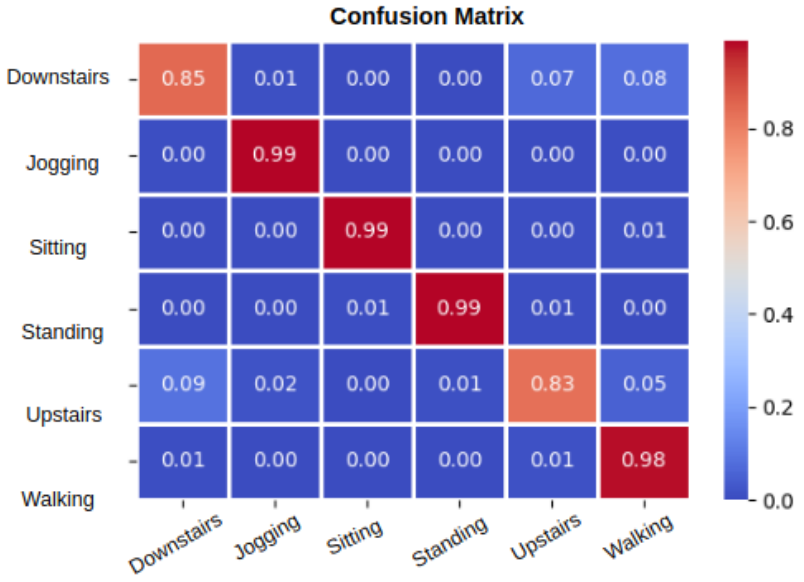
(d) Random Forest classifier with average accuracy of 89% on test dataset.

Figure 21: Confusion matrix for the classical machine learning methods.



(a) DNN model with average accuracy of 87% on the test dataset.

(b) LSTM model with average accuracy of 95% on the test dataset.



(c) BLSTM model with average accuracy of 95% on test dataset.

Figure 22: Confusion matrix for the deep learning methods.

C.4 where derived from. Another important aspect of pre-processing is filtering which can be used to remove undesired components or features from the signal. This has not been discussed since implementing a Butterworth Lowpass filter and median filter did not have any impact on the classifiers performance in this case. For some classifiers it even reduced the performance. A different approach to increase performance could be to adjust the window size during signal segmentation. By increasing the window to 5.5 seconds (instead of 2.5 seconds) the performance of some of the classical classifiers seem to perform better by up to 3-5%, but for others diminish. For the DL classifiers the increasing window size had a negative effect. Increasing the window size to much and the performance for all classifiers started to deteriorate. The window of 2.5 seconds was seen as a middle ground for the different types of classifiers, and similar windows have been used for other solutions [5][28].

4.3 Comparative Analysis

This section briefly present the result obtained in our approach in comparison with other state-of-the-art solutions: Paper 1 [5], paper 2 [21] and paper 3 [28]. For a fair comparison we have analysed it against two papers using the same dataset, paper 2 and paper 3. Paper 2 uses an DNN approach and implements data augmentation and analysis the impact of using only two axis of accelerometer data, while paper 3 uses a Convolutional Neural Network (CNN) model. Paper 1 uses the same model as our proposed approach (LSTM), but on a different, more balanced, dataset with 10 participants and multiple sensors (3 IMU and one ECG sensor). Table 8 presents the recognition rate for the different papers and two of the NN models tested in our approach (DNN and LSTM).

Table 8: Comparing results of DL classifiers with state-of-the-art approaches: Paper 1 [5], paper 2 [21] and paper 3[28].

Activity	[5]	[21]: y-z axis	[21]: x-y-z axis	[28]	**DNN**	**LSTM**
Downstairs	94%	89.1%	71.5%	76.7%	48%	83%
Jogging	91%	95.5%	94.0%	97.6%	97%	99%
Sitting	92%	90.3%	92%	82.3%	98%	99%
Standing	100%	93.2%	93.7%	95.7%	99%	100%
Upstairs	94%	82.8%	79.3%	62.9%	50%	83%
Walking	100%	95.9%	92.5%	97.1%	95.2%	98%
Average	95%	91.1 %	87.2%	90.4%	81.2%	93.7%

Compared with the CNN model in paper 3 our LSTM model performs about 3.3% better. The difference in jogging, walking and standing are minor compared to sitting with difference of 16.7%, downstairs with 6.3% and upstairs with 20.1%. This results back up the assumption that RNN models performs better on time-

series data than CNN models which are more frequently used for image processing. Comparing our LSTM with the LSTM in paper 1 the average performance is rather similar. The multimodal system performs on average only 1.3% better than our single accelerometer system. Our model performs slightly better for jogging (8%) and sitting (7%), but especially for downstairs and upstairs the model in paper 1 performs better 11%. This is believed to have some correlation with the fact that the dataset is more balanced compared with WISDM dataset, but the enhanced performance is also most likely correlated with the additional IMUs and ECG sensor. It is also difficult to compare these models when they are implemented on different datasets. The dataset from paper 1 has much fewer participants compared to the WISDM dataset and this can have a impact on performance.

Paper 2 uses a DNN model and Synthetic Minority Over-sampling Technique (SMOTE) for data augmentation [29]. By comparing with our DNN we see that the our proposed approach perform slightly better on activities such as jogging, sitting and standing but much worse on the activities downstairs (23.5%) and upstairs (29.3%). It therefore seems that the data augmentation has had a relative large impact on the most misclassified activities of the DNN model. In addition, in paper 2 there was an additional improvements by using only y-z axis of the accelerometer. It was shown that by eliminating the x-axis the averaged performance was raised by 3.9%. Paper 2 has therefore two additional aspects that can be implemented for our proposed approach and can increase the performance further.

5 Conclusion and Future work

This paper aims to understand the process of a Human Activity Recognition system with a focus on Deep Learning classifiers and how they compared with classical Machine Learning techniques and other state-of-the-art solutions. To reach this goal, a HAR system for daily activities, based on a publicly available accelerometer dataset using a single sensor, was implemented in Python.

The paper has focused on the accuracy and precision rate of each activity for different classifiers and what advancements can be made to increase the performance further. The classical ML methods implemented was Logistic Regression, Support Vector Machine, Decision Tree and Random Forest, where Random Forest attained the highest average accuracy of 89%. The DL classifiers where Deep Neural Networks and two Recurrent Neural Network (LSTM and BLSTM). The two RNNs performed considerable better then the classical classifiers with an accuracy of 95%. When comparing our RNN models performance with the performance of a state-of-the-art CNN solution [28], it was seen that the performance of the RNN models were 3.3% higher. This supports the RNN model as a viable solution for HAR. There are some disadvantages with the RNN models concerning their long training time and vanishing and exploding gradients. Therefore, it would be interesting to compare the performance with a Transformer model, which has gained a lot of interest in the last couple of years in NLP [30]. This model avoids the problem with the gradients and reduces the training time since the model can be trained in parallel.

Additionally, our results demonstrated that most of the classifiers performed well on distinct activity patterns such as walking, jogging, sitting and standing, but had more difficulties distinguishing similar activities such as walking downstairs and upstairs. The unbalance in the datasets can have an effect on the misclassification since it gives a bias towards the more frequent activities walking and jogging. The activities sitting and standing had a high accuracy even though they had the lowest number of samples. The reason why these activities were not misclassified is most likely because of their distinct and monotone accelerometer pattern seen in figure 14. Upstairs and downstairs have similar patterns with each other and to some degree walking and jogging. Therefore, to improve our classification system it could be beneficial to balance our dataset by introducing a form of data augmentation. This is supported by paper 2 [21] in the comparative analysis, which achieved higher accuracy for activities downstairs and upstairs with data augmentation. Another improvement that this paper introduced was the increased performance of the DNN by only using data from the y-z axis. Therefore, it would be interesting to see if this alteration would boost the performance of the proposed RNN models.

To investigate how our proposed approach performs against a HAR system with multiple sensors, our model was compared with a similar LSTM model from paper

3 [5]. Our LSTM model only performed slightly worse (1.3%) than the multimodal LSTM, which seemingly supports the usage of only a single accelerometer for recognising physical activity. The comparison is problematic since the two models are using different datasets. The proposed model should therefor be tested on the same datasets , but using only one sensor, to verify the usage of a single accelerometer sensor for activity recognition. Nevertheless, the comparison indicates that the proposed approach has promising results compared to other solutions and supports the use of a single accelerometer sensor for Deep Learning classifiers in the field of Human Activity Recognition.

In addition to the improvements mentioned above, the list below presents some possible future work:

- Adding more activities to the recognition system would give a better information base for the user. These activities can, for instance, be different types of sporting activities or regular daily activities important for monitoring people in their homes or at work.
- Implementing a semi-supervised Human Activity Recognition system which uses a minor labelled dataset and a larger labelled dataset to train the model. The algorithm would then have a more extensive database to train on, perhaps leading to a higher performance.
- Implement the proposed HAR system for a real-life application.
    - Developing a form of position tracking for home monitoring in eldercare to help people live longer in their homes.
    - Address the problem with privacy that smart monitoring systems provide. Addressing that non-camera based approaches can be more acceptable for home monitoring for eldercare.

Bibliography

[1] S. Stavrotheodoros, N. Kaklanis, K. Votis and D. Tzovaras, 'A smart-home iot infrastructure for the support of independent living of older adults', in *Artificial Intelligence Applications and Innovations*, L. Iliadis, I. Maglogiannis and V. Plagianakos, Eds., Cham: Springer International Publishing, 2018, pp. 238–249, ISBN: 978-3-319-92016-0.

[2] A. S. A. Sukor, A. Zakaria and N. A. Rahim, 'Activity recognition using accelerometer sensor and machine learning classifiers', in *2018 IEEE 14th International Colloquium on Signal Processing Its Applications (CSPA)*, 2018, pp. 233–238. DOI: 10.1109/CSPA.2018.8368718.

[3] N. D. Charmi Jobanputraa Jatna Bavishib, 'Human activity recognition: A survey', *Elsevier B.V*, 2019. DOI: https://doi.org/10.1016/j.procs.2019.08.100.

[4] F. Ordóñez and D. Roggen, 'Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition', *Sensors*, vol. 16, p. 115, Jan. 2016. DOI: 10.3390/s16010115.

[5] M. Z. Uddin and A. Soylu, 'Human activity recognition using wearable sensors, discriminant analysis, and long short-term memory-based neural structured learning', *Scientific Reports*, vol. 11, Aug. 2021. DOI: 10.1038/s41598-021-95947-y.

[6] D. Figo, P. C. Diniz, D. R. Ferreira and J. M. P. Cardoso, 'Preprocessing techniques for context recognition from accelerometer data', *Pers. Ubiquitous Comput.*, vol. 14, no. 7, pp. 645–662, 2010. DOI: 10.1007/s00779-010-0293-9. [Online]. Available: https://doi.org/10.1007/s00779-010-0293-9.

[7] S. Mohammed, W. Gomaa, M. Abdu-Aguye and N. El-borae, 'Improved imu-based human activity recognition using hierarchical hmm dissimilarity', Jan. 2020, pp. 702–709. DOI: 10.5220/0009886607020709.

[8] D. Anguita, A. Ghio, L. Oneto, X. Parra and J. L. Reyes-Ortiz, 'A public domain dataset for human activity recognition using smartphones', in *ESANN*, 2013.

[9] J. Han, J. Pei and M. Kamber, *Data Mining: Concepts and Techniques*, ser. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2011, pp. 111–118, ISBN: 9780123814807. [Online]. Available: https://books.google.no/books?id=pQws07tdpjoC.

[10] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[11] O. Chapelle, B. Scholkopf and A. Zien Eds., '¡emphasis emphasistype="bold"¿semi-supervised learning¡/emphasis¿ (chapelle, o. et al., eds.; 2006) [book reviews]', *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 542–542, 2009. DOI: 10.1109/TNN.2009.2015974.

[12] C. Finn, T. Yu, J. Fu, P. Abbeel and S. Levine, *Generalizing skills with semi-supervised reinforcement learning*, 2017. arXiv: 1612.00429 [cs.LG].

[13] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[14] M. Grandini, E. Bagli and G. Visani, 'Metrics for multi-class classification: An overview', *ArXiv*, vol. abs/2008.05756, 2020.

[15] C. M. Bishop, *Neural Networks for Pattern Recognition*. USA: Oxford University Press, Inc., 1995, ISBN: 0198538642.

[16] L. Szymanski and B. McCane, 'Visualising kernel spaces', 2011.

[17] M. Pal, 'Multiclass approaches for support vector machine based land cover classification', Mar. 2008.

[18] L. Breiman, J. H. Friedman, R. A. Olshen and C. J. Stone, 'Classification and regression trees', 1983.

[19] J. R. Kwapisz, G. M. Weiss and S. A. Moore, 'Activity recognition using cell phone accelerometers', *SIGKDD Explor. Newsl.*, vol. 12, no. 2, pp. 74–82, Mar. 2011, ISSN: 1931-0145. DOI: 10.1145/1964897.1964918. [Online]. Available: https://doi.org/10.1145/1964897.1964918.

[20] S. Kotsiantis, D. Kanellopoulos and P. Pintelas, 'Data preprocessing for supervised learning', *International Journal of Computer Science*, vol. 1, pp. 111–117, Jan. 2006.

[21] A. R. Javed, U. Sarwar, S. Khan, C. Iwendi, M. Mittal and N. Kumar, 'Analyzing the effectiveness and contribution of each axis of tri-axial accelerometer sensor for accurate activity recognition', *Sensors*, vol. 20, p. 2216, Apr. 2020. DOI: 10.3390/s20082216.

[22] M. Christ, N. Braun, J. Neuffer and A. W. Kempa-Liehr, 'Time series feature extraction on basis of scalable hypothesis tests (tsfresh – a python package)', *Neurocomputing*, vol. 307, pp. 72–77, 2018, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2018.03.067. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231218304843.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay, 'Scikit-learn: Machine learning in Python', *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[24] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[25] F. Chollet *et al.* (2015). 'Keras', [Online]. Available: https://github.com/fchollet/keras.

[26] S. Mc Loone and G. Irwin, 'Improving neural network training solutions using regularisation', *Neurocomputing*, vol. 37, no. 1, pp. 71–90, 2001, ISSN: 0925-2312. DOI: https://doi.org/10.1016/S0925-2312(00)00314-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231200003143.

[27] Z. Chen, Q. Zhu, Y. C. Soh and L. Zhang, 'Robust human activity recognition using smartphone sensors via ct-pca and online svm', *IEEE Transactions on Industrial Informatics*, vol. 13, no. 6, pp. 3070–3080, 2017. DOI: 10.1109/TII.2017.2712746.

[28] A. Ignatov, 'Real-time human activity recognition from accelerometer data using convolutional neural networks', *Applied Soft Computing*, vol. 62, pp. 915–922, 2018, ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2017.09.027. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494617305665.

[29] K. W. Bowyer, N. V. Chawla, L. O. Hall and W. P. Kegelmeyer, 'SMOTE: synthetic minority over-sampling technique', *CoRR*, vol. abs/1106.1813, 2011. arXiv: 1106.1813. [Online]. Available: http://arxiv.org/abs/1106.1813.

[30] N. Wu, B. Green, X. Ben and S. O'Banion, *Deep transformer models for time series forecasting: The influenza prevalence case*, Jan. 2020.

Appendix

A Neural Networks Model Architecture

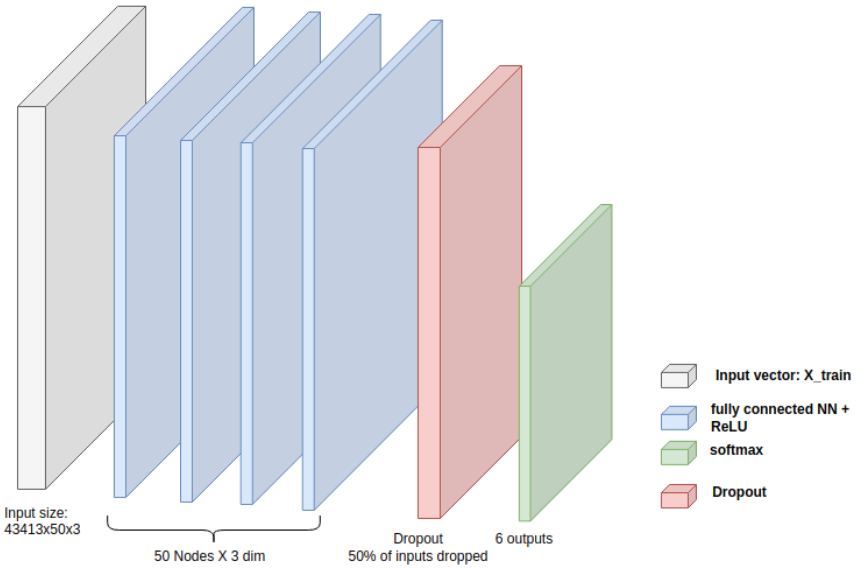


Figure 23: Model architecture for DNN containing 4 hidden layers and one dropout layer.

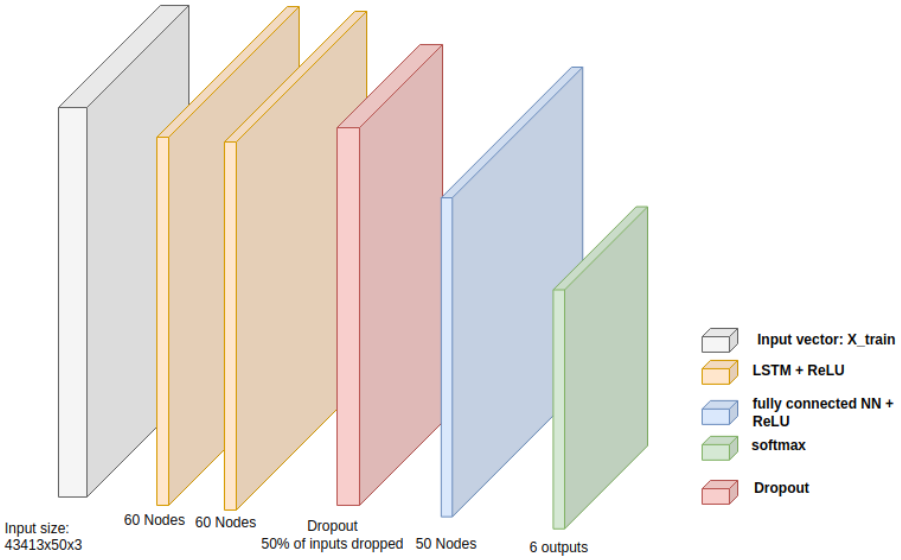


Figure 24: Model architecture for LSTM containing 2 LSTM layers and one dropout layer.

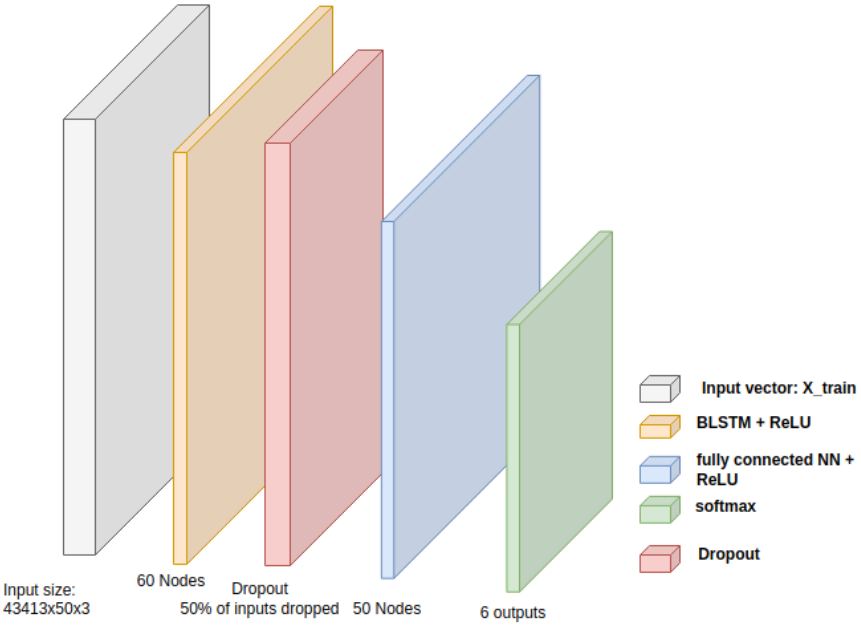


Figure 25: Model architecture for BLSTM containing one BLSTM layer and one dropout layer.

B Classification Report

Table 9: Linear Regression: Classification report

Activity	Precision	Recall	f1-score	Accuracy
Downstairs	0.76	0.80	0.78	241
Jogging	0.83	0.99	0.91	503
Sitting	0.85	0.98	0.91	114
Standing	1.00	0.81	0.90	95
Upstairs	0.58	0.61	0.59	195
Walking	0.99	0.85	0.92	838
Total:				
Accuracy			0.86	1986
Macro avg	0.84	0.84	0.83	1986
Weighted avg	0.88	0.86	0.86	1986

Table 10: Linear SVM: Classification report

Activity	Precision	Recall	f1-score	Accuracy
Downstairs	0.78	0.80	0.79	241
Jogging	0.83	0.99	0.90	503
Sitting	0.90	1.00	0.95	114
Standing	0.95	0.86	0.91	95
Upstairs	0.55	0.58	0.56	195
Walking	0.99	0.85	0.91	838
Total:				
Accuracy			0.86	1986
Macro avg	0.84	0.85	0.84	1986
Weighted avg	0.87	0.86	0.86	1986

Table 11: Decision Tree: Classification report

Activity	Precision	Recall	f1-score	Accuracy
Downstairs	0.54	0.60	0.57	241
Jogging	0.81	0.99	0.89	503
Sitting	0.94	0.90	0.92	114
Standing	0.91	0.95	0.93	95
Upstairs	0.38	0.17	0.23	195
Walking	0.88	0.85	0.87	838
Total:				
Accuracy			0.80	1986
Macro avg	0.74	0.74	0.73	1986
Weighted avg	0.78	0.80	0.78	1986

Table 12: Random Forest: Classification report

Activity	Precision	Recall	f1-score	Accuracy
Downstairs	0.88	0.77	0.82	241
Jogging	0.81	0.99	0.89	503
Sitting	0.97	0.99	0.98	114
Standing	1.00	0.98	0.99	95
Upstairs	0.70	0.63	0.66	195
Walking	0.97	0.90	0.93	838
Total:				
Accuracy			0.89	1986
Macro avg	0.89	0.88	0.88	1986
Weighted avg	0.89	0.89	0.89	1986

Table 13: DNN: Classification report

Activity	Precision	Recall	f1-score	Accuracy
Downstairs	0.68	0.48	0.56	387
Jogging	0.96	0.97	0.97	1307
Sitting	1.00	0.98	0.99	233
Standing	0.99	0.99	0.99	190
Upstairs	0.65	0.50	0.57	490
Walking	0.85	0.95	0.90	1735
Total:				
Accuracy			0.87	4342
Macro avg	0.85	0.81	0.83	4342
Weighted avg	0.86	0.87	0.86	4342

Table 14: LSTM: Classification report

Classification report for test data:				
Activity	Precision	Recall	F1-score	Support
Downstairs	0.86	0.83	0.84	387
Jogging	0.98	0.99	0.99	1307
Sitting	0.99	0.99	0.99	233
Standing	0.96	1.00	0.98	190
Upstairs	0.86	0.83	0.85	490
Walking	0.97	0.98	0.98	1735
Total:				
Accuracy			0.95	4342
Macro avg	0.94	0.94	0.94	4342
Weighted avg	0.95	0.95	0.95	4342

Table 15: BLSTM: Classification report

Classification report for test data:				
Activity	Precision	Recall	F1-score	Support
Downstairs	0.85	0.85	0.85	387
Jogging	0.99	0.99	0.99	1307
Sitting	0.99	0.99	0.99	233
Standing	0.96	0.99	0.98	190
Upstairs	0.88	0.83	0.86	490
Walking	0.96	0.98	0.97	1735
Total:				
Accuracy			0.95	4342
Macro avg	0.94	0.94	0.94	4342
Weighted avg	0.95	0.95	0.95	4342

C Python code

C.1 Classical Machine Learning Classifiers

```
######################################################################
#                   1) Pre-processing the data
######################################################################
# Extract data
data = txt_to_pd_WISDM()
feature_df = feature_extraction(data, sec=5.5, overlap_prosent=50)
# Normalize data [-1,1]: across subjects or
data = normalize_data(feature_df)
######################################################################
#           Step 2) split data into training and test sets:
######################################################################
random_seed =42
X_train, X_test, X_val, y_train, y_test, y_val,class_labels = \
split_train_test_data(feature_df, ratio=0.8)
#  conduct one-hot-encoding of our labels:
y_train_hot = pd.get_dummies(y_train).to_numpy()
y_test_hot = pd.get_dummies(y_test).to_numpy()
y_val_hot = pd.get_dummies(y_train).to_numpy()
######################################################################
#       Step 3/4) Create and train model:
#   Perform grid search to find best hyperparameters
```

```
#   Function model_tuning performs grid search and model training
#   returns a performance dictionary with training time, test time,
#   predicted values, accuracy, classification report and the model
#######################################################################
## Logistic regression:
log_reg = linear_model.LogisticRegression(solver='liblinear')
parameters = {'C':[0.01, 0.1, 1, 10, 20, 30],
                'penalty':['l2','l1']}
log_reg_res = model_tuning(log_reg,class_labels, X_train, y_train, X_val,
            y_val, parameters, n_jobs=-1, cv=3, verbose=1)
## Linear support vector classification
parameters = {'C':[0.125, 0.5, 1, 2, 8, 16]}
lr_svc = LinearSVC(tol=0.00005, dual=False, 30)
lr_svc_res = model_tuning(lr_svc,class_labels, X_train, y_train, X_val,
            y_val, parameters, n_jobs=-1, verbose=1)
## decision tree classifier:
parameters = {'max_depth':[6]} # np.arange(1,10,1)}
dtc = DecisionTreeClassifier()
dtc_res = model_tuning(dtc,class_labels, X_train, y_train, X_val,
            y_val, parameters, n_jobs=-1,)
# # Random Forest Classifier
params = {'n_estimators': [151], #np.arange(1,181,25),
           'max_depth': [19]} #np.arange(1,25,2)}
rfc = RandomForestClassifier()
rfc_res = model_tuning(rfc,class_labels, X_train, y_train, X_val,
            y_val, params, n_jobs=-1,)

#######################################################################
#       Step 5) Validation and evaluation of model
#######################################################################
#Performance on validation set::
model_res ={'Logistic Regression':log_reg_res,
           'Linear SVC ': lr_svc_res,
           ' Decision tree classifier' :dtc_res,
           'Random Forest classifier':rfc_res}

t = PrettyTable(['ML model', 'Accuracy', 'Error'])
for key in model_res:
```

```
    accuracy = str(round(model_res[key]['accuracy']*100, 2))+'%'
    error = str(round(100-(model_res[key]['accuracy']*100), 2)) +'%'
    t.add_row([key, accuracy, error])
print(t)

# Performance on test set:

for key in model_res:
    print('model: ', key)
    model = model_res[key]['model']
    y_pred = model.predict(X_test)
    # Show model performance stats:
    accuracy = metrics.accuracy_score(y_true=y_test, y_pred=y_pred)
    cm = metrics.confusion_matrix(y_test, y_pred)
    print('********Classification Report********')
    classification_report = metrics.classification_report(y_test,
    ↪  y_pred)
    print(classification_report)
```

C.2 Deep learning Classifiers

```
#####################################################################
#                   1) Pre-processing the data
#####################################################################
# Extract data
data = txt_to_pd_WISDM()
# Normalize data [-1,1]: across subjects or
data = normalize_data(data)
segment_array, segment_labels,LABELS = extract_windows(data,
↪  sec=2.5, overlap_prosent=50)
#####################################################################
#           Step 2) split data into training and test sets:
#####################################################################
X_train, X_test, y_train, y_test = train_test_split(segment_array,
        segment_labels, test_size = 0.2, random_state = random_seed)

X_test, X_val, y_test, y_val
```

```
    = train_test_split(X_test, y_test, test_size=0.5,
    ↪ random_state=1)
# Conduct one-hot-encoding of labels:
y_train_hot = pd.get_dummies(y_train).to_numpy()
y_test_hot = pd.get_dummies(y_test).to_numpy()
y_val_hot = pd.get_dummies(y_val).to_numpy()


######################################################################
#       Step 3/4) Create and train model
# Create the  DNN and choose the hyperparameters for the NN
######################################################################
# Initializing parameters
parameters = {"n_hidden": [1, 2, 3, 4],
              "penalty": ['L1','L2']
                'C': [0.01,0.1, 1, 10, 20, 30],
                  "n_neurons": np.arange(10, 70, 10),
                  "learning_rate":[3e-4, 3e-2],
                  }
# Initialise models:
DNN = DNN_model(epochs, N_LAYERS, COSTUME, n_classes)
LSTM= LSTM_model(epochs, n_classes, timesteps, N_FEATURES=input_dim)
BLSTM = bidir_LSTM_model(epochs, n_classes, timesteps,
↪ N_FEATURES=input_dim)
nn_model = [DNN, LSTM, BLSTM]
# Hyperparameter tuning:
tuner = RandomSearch(nn_model,
                        objective='val_accuracy',
                         max_trials=3,
                         executions_per_trial=2,
                         overwrite=True)

tuner.search(X_train, y_train, epochs=40, validation_data=(X_val,
↪ y_val))
models = tuner.get_best_models(num_models=2)
# Train model:
train_model(models, X_train, y_train_hot)

######################################################################
#       Step 5) Validation and evaluation of model
######################################################################
#test models:
```

```
for model in models:
    # validation data performance
    y_pred_val = model.predict(X_val)

    best_class_val = np.argmax(y_pred_val, axis=1)

    print('Classification report for training data:')
    print(classification_report(y_val, best_class_val))
    # Training data performance
    y_pred_test = model.predict(X_test)
    best_class_pred_test = np.argmax(y_pred_test, axis=1)
    best_class_test = np.argmax(y_test_hot, axis=1)

    print('Classification report for test data')
    print(classification_report(best_class_test,
    ↪ best_class_pred_test))
    confusion_matrix(best_class_test, best_class_pred_test, LABELS,
    ↪ normalize=True)

    #Evaluation score: categorical cross-entropy and accuracy
    score = model.evaluate(X_test, y_test_hot)
```

C.3 Data Normalisation

```
def normalize_data(data):
    drop_columns =['timestamp', 'user_id', 'activity']
    for column in data.drop(columns=drop_columns):
        numerator = (data[column]-data[column].min())
        denominator = (data[column].max()-data[column].min())
        data[column] = 2*(numerator/denominator)-1

    return data

```

C.4 Feature Extraction

```
#########################################################################
#                   tilt_angle ()
#   Extracting the tilt angles from the accelerometer data
#########################################################################

def tilt_angle(data):
    Ax = np.arctan2(data['x-axis'],
         np.sqrt(data['y-axis']**2+data['z-axis']**2))
    Ay = np.arctan2(data['y-axis'],
          np.sqrt(data['x-axis']**2+data['z-axis']**2))
    df = pd.DataFrame({'Ax': Ax, 'Ay': Ay})
    data = pd.merge(data, df, left_index=True, right_index=True)
    return data

#########################################################################
#                   tilt_angle ()
#   Extracting the tilt angles from the accelerometer data
#########################################################################

def feature_extraction(data, sec=10, overlap_prosent=50):
     x_values, y_values, z_values =([] for i in range(3))
    ax_values, ay_values, activity_labels, id_values = ([] for i in
    ↪  range(4))
    xy_corr, xz_corr, yz_corr =([] for i in range(3))

    # Extracty the wanted features from segment
    time_steps = int(sec*20)
    overlap = time_steps-int(time_steps*(overlap_prosent/100))
    if overlap_prosent>100 or overlap_prosent<0:
        print("Invalid Input Entered")
        print("Overlap_prosent value must be between [0,100]")
        print("And the total overlap can't be zero, must at least be
        ↪  one")
        exit(0)
    # 20Hz (1 sample every 50ms) for WISDM
    if overlap <=0:
        print('Overlap is sat to 1 sample since it was set to 0
        ↪  which is not possible')
        print('One is the smallest possible overlap')
```

```
        overlap =1
    cutoff = 6/20
    data_high = filter_butter(data, cutoff, type='high')
    data_low = filter_butter(data, cutoff, type='low')

    data = (tilt_angle(data))
    # windowing function:
    for i in range(0, len(data)- time_steps, overlap):
        xs = data['x-axis'].values[i: i + time_steps]
        ys = data['y-axis'].values[i: i + time_steps]
        zs = data['z-axis'].values[i: i + time_steps]

        xs_low = data_low['x-axis'].values[i: i + time_steps]
        ys_low = data_low['y-axis'].values[i: i + time_steps]
        zs_low = data_low['z-axis'].values[i: i + time_steps]

        xs_high = data_high['x-axis'].values[i: i + time_steps]
        ys_high = data_high['y-axis'].values[i: i + time_steps]
        zs_high = data_high['z-axis'].values[i: i + time_steps]
        ax = data['Ax']. values[i: i + time_steps]
        ay = data['Ax']. values[i: i + time_steps]


        activity = data['activity'][i: i + time_steps].mode()[0]
        id = data['user_id'][i: i + time_steps].mode()[0]

        x_values.append(xs)
        y_values.append(ys)
        z_values.append(zs)

        x_values_low.append(xs_low)
        y_values_low.append(ys_low)
        z_values_low.append(zs_low)
        x_values_high.append(xs_high)
        y_values_high.append(ys_high)
        z_values_high.append(zs_high)

        ax_values.append(ax)

        ay_values.append(ay)
        activity_labels.append(activity)
        id_values.append(id)
```

```
79
80          xy_corr.append(pearsonr(xs,ys)[0])
81          xz_corr.append(pearsonr(xs,zs)[0])
82          yz_corr.append(pearsonr(zs,ys)[0])
83
84      # Statistical Features on raw x, y and z in time domain
85      feature_df = pd.DataFrame()
86      feature_df['user_id'] = id_values
87      feature_df['activity'] = activity_labels
88      #fast fourier transform:
89      # converting the signals from time domain to frequency domain
        ↪  using FFT
90      x_values_fft = pd.Series(x_values).apply(lambda x:
        ↪  np.abs(np.fft.fft(x))[1:int(time_steps/2+1)])
91      y_values_fft = pd.Series(y_values).apply(lambda x:
        ↪  np.abs(np.fft.fft(x))[1:int(time_steps/2+1)])
92      z_values_fft = pd.Series(z_values).apply(lambda x:
        ↪  np.abs(np.fft.fft(x))[1:int(time_steps/2+1)])
93
94      # mean
95      feature_df['x_mean'] = pd.Series(x_values).apply(lambda x:
        ↪  x.mean())
96      feature_df['y_mean'] = pd.Series(y_values).apply(lambda x:
        ↪  x.mean())
97      feature_df['z_mean'] = pd.Series(z_values).apply(lambda x:
        ↪  x.mean())
98      feature_df['ax_mean'] = pd.Series(ax_values).apply(lambda x:
        ↪  x.mean())
99      feature_df['ay_mean'] = pd.Series(ay_values).apply(lambda x:
        ↪  x.mean())
100
101     # std dev
102     feature_df['x_std'] = pd.Series(x_values).apply(lambda x:
        ↪  x.std())
103     feature_df['y_std'] = pd.Series(y_values).apply(lambda x:
        ↪  x.std())
104     feature_df['z_std'] = pd.Series(z_values).apply(lambda x:
        ↪  x.std())
105
106     # avg absolute diff
107     feature_df['x_aad'] = pd.Series(x_values).apply(lambda x:
        ↪  np.mean(np.absolute(x - np.mean(x))))
```

```
108     feature_df['y_aad'] = pd.Series(y_values).apply(lambda x:
        ↪  np.mean(np.absolute(x - np.mean(x))))
109     feature_df['z_aad'] = pd.Series(z_values).apply(lambda x:
        ↪  np.mean(np.absolute(x - np.mean(x))))
110
111     # min
112     feature_df['x_min'] = pd.Series(x_values).apply(lambda x:
        ↪  x.min())
113     feature_df['y_min'] = pd.Series(y_values).apply(lambda x:
        ↪  x.min())
114     feature_df['z_min'] = pd.Series(z_values).apply(lambda x:
        ↪  x.min())
115
116     # max
117     feature_df['x_max'] = pd.Series(x_values).apply(lambda x:
        ↪  x.max())
118     feature_df['y_max'] = pd.Series(y_values).apply(lambda x:
        ↪  x.max())
119     feature_df['z_max'] = pd.Series(z_values).apply(lambda x:
        ↪  x.max())
120
121     # max-min diff
122     feature_df['x_maxmin_diff'] = feature_df['x_max'] -
        ↪  feature_df['x_min']
123     feature_df['y_maxmin_diff'] = feature_df['y_max'] -
        ↪  feature_df['y_min']
124     feature_df['z_maxmin_diff'] = feature_df['z_max'] -
        ↪  feature_df['z_min']
125
126     # median
127     feature_df['x_median'] = pd.Series(x_values).apply(lambda x:
        ↪  np.median(x))
128     feature_df['y_median'] = pd.Series(y_values).apply(lambda x:
        ↪  np.median(x))
129     feature_df['z_median'] = pd.Series(z_values).apply(lambda x:
        ↪  np.median(x))
130
131     # median abs dev
132     feature_df['x_mad'] = pd.Series(x_values).apply(lambda x:
        ↪  np.median(np.absolute(x - np.median(x))))
133     feature_df['y_mad'] = pd.Series(y_values).apply(lambda x:
        ↪  np.median(np.absolute(x - np.median(x))))
```

```
    feature_df['z_mad'] = pd.Series(z_values).apply(lambda x:
    ↪  np.median(np.absolute(x - np.median(x))))

    # interquartile range
    feature_df['x_IQR'] = pd.Series(x_values).apply(lambda x:
    ↪  np.percentile(x, 75) - np.percentile(x, 25))
    feature_df['y_IQR'] = pd.Series(y_values).apply(lambda x:
    ↪  np.percentile(x, 75) - np.percentile(x, 25))
    feature_df['z_IQR'] = pd.Series(z_values).apply(lambda x:
    ↪  np.percentile(x, 75) - np.percentile(x, 25))

    # negtive count
    feature_df['x_neg_count'] = pd.Series(x_values).apply(lambda x:
    ↪  np.sum(x < 0))
    feature_df['y_neg_count'] = pd.Series(y_values).apply(lambda x:
    ↪  np.sum(x < 0))
    feature_df['z_neg_count'] = pd.Series(z_values).apply(lambda x:
    ↪  np.sum(x < 0))

    # positive count
    feature_df['x_pos_count'] = pd.Series(x_values).apply(lambda x:
    ↪  np.sum(x > 0))
    feature_df['y_pos_count'] = pd.Series(y_values).apply(lambda x:
    ↪  np.sum(x > 0))
    feature_df['z_pos_count'] = pd.Series(z_values).apply(lambda x:
    ↪  np.sum(x > 0))

    # values above mean
    feature_df['x_above_mean'] = pd.Series(x_values).apply(lambda x:
    ↪  np.sum(x > x.mean()))
    feature_df['y_above_mean'] = pd.Series(y_values).apply(lambda x:
    ↪  np.sum(x > x.mean()))
    feature_df['z_above_mean'] = pd.Series(z_values).apply(lambda x:
    ↪  np.sum(x > x.mean()))

    # number of peaks
    feature_df['x_peak_count'] = pd.Series(x_values).apply(lambda x:
    ↪  len(find_peaks(x)[0]))
    feature_df['y_peak_count'] = pd.Series(y_values).apply(lambda x:
    ↪  len(find_peaks(x)[0]))
    feature_df['z_peak_count'] = pd.Series(z_values).apply(lambda x:
    ↪  len(find_peaks(x)[0]))
```

```
160     #FFT number of peaks
161     feature_df['x_peak_count_fft'] =
        ↪  pd.Series(x_values_fft).apply(lambda x:
        ↪  len(find_peaks(x)[0]))
162     feature_df['y_peak_count_fft'] =
        ↪  pd.Series(y_values_fft).apply(lambda x:
        ↪  len(find_peaks(x)[0]))
163     feature_df['z_peak_count_fft'] =
        ↪  pd.Series(z_values_fft).apply(lambda x:
        ↪  len(find_peaks(x)[0]))
164
165     # skewness
166     feature_df['x_skewness'] = pd.Series(x_values).apply(lambda x:
        ↪  stats.skew(x))
167     feature_df['y_skewness'] = pd.Series(y_values).apply(lambda x:
        ↪  stats.skew(x))
168     feature_df['z_skewness'] = pd.Series(z_values).apply(lambda x:
        ↪  stats.skew(x))
169     # FFT skewness
170     feature_df['x_skewness_fft'] =
        ↪  pd.Series(x_values_fft).apply(lambda x: stats.skew(x))
171     feature_df['y_skewness_fft'] =
        ↪  pd.Series(y_values_fft).apply(lambda x: stats.skew(x))
172     feature_df['z_skewness_fft'] =
        ↪  pd.Series(z_values_fft).apply(lambda x: stats.skew(x))
173     # kurtosis
174     feature_df['x_kurtosis'] = pd.Series(x_values).apply(lambda x:
        ↪  stats.kurtosis(x))
175     feature_df['y_kurtosis'] = pd.Series(y_values).apply(lambda x:
        ↪  stats.kurtosis(x))
176     feature_df['z_kurtosis'] = pd.Series(z_values).apply(lambda x:
        ↪  stats.kurtosis(x))
177     # FFT kurtosis
178     feature_df['x_kurtosis_fft'] =
        ↪  pd.Series(x_values_fft).apply(lambda x: stats.kurtosis(x))
179     feature_df['y_kurtosis_fft'] =
        ↪  pd.Series(y_values_fft).apply(lambda x: stats.kurtosis(x))
180     feature_df['z_kurtosis_fft'] =
        ↪  pd.Series(z_values_fft).apply(lambda x: stats.kurtosis(x))
181
182     # Energy: expect different energy for low and highpassed
        ↪  signal
```

```
183     feature_df['x_energy'] = pd.Series(x_values).apply(lambda x:
        ↪  np.sum(x**2)/100)
184     feature_df['y_energy'] = pd.Series(y_values).apply(lambda x:
        ↪  np.sum(x**2)/100)
185     feature_df['z_energy'] = pd.Series(z_values).apply(lambda x:
        ↪  np.sum(x**2/100))
186
187     # FFT  Spectral energy
188     feature_df['x_energy_fft'] =
        ↪  pd.Series(x_values_fft).apply(lambda x: np.sum(x**2)/50)
189     feature_df['y_energy_fft'] =
        ↪  pd.Series(y_values_fft).apply(lambda x: np.sum(x**2)/50)
190     feature_df['z_energy_fft'] =
        ↪  pd.Series(z_values_fft).apply(lambda x: np.sum(x**2/50))
191
192     # FFT entropy:
193     feature_df['x_entropi_fft'] =
        ↪  pd.Series(x_values_fft).apply(lambda x: entropy(x))
194     feature_df['y_entropi_fft'] =
        ↪  pd.Series(y_values_fft).apply(lambda x: entropy(x))
195     feature_df['z_entropi_fft'] =
        ↪  pd.Series(z_values_fft).apply(lambda x: entropy(x))
196
197     # avg resultant
198     feature_df['avg_result_accl'] = [i.mean() for i in
        ↪  ((pd.Series(x_values)**2 + pd.Series(y_values)**2 +
        ↪  pd.Series(z_values)**2)**0.5)]
199
200     # signal magnitude area
201     feature_df['sma'] =    pd.Series(x_values).apply(lambda x:
        ↪  np.sum(abs(x)/100)) + pd.Series(y_values).apply(lambda x:
        ↪  np.sum(abs(x)/100)) \
202                     + pd.Series(z_values).apply(lambda x:
                        ↪  np.sum(abs(x)/100))
203
204
205    # correlation:
206     feature_df['xy_corr'] =pd.Series(xy_corr)
207     feature_df['xz_corr'] = pd.Series(xz_corr)
208     feature_df['yz_corr'] = pd.Series(yz_corr)
209
210
```

```
211     #Capturing indices
212      # Max Indices and Min indices
213
214      # index of max value in time domain
215      feature_df['x_argmax'] = pd.Series(x_values).apply(lambda x:
          ↪  np.argmax(x))
216      feature_df['y_argmax'] = pd.Series(y_values).apply(lambda x:
          ↪  np.argmax(x))
217      feature_df['z_argmax'] = pd.Series(z_values).apply(lambda x:
          ↪  np.argmax(x))
218
219      # index of min value in time domain
220      feature_df['x_argmin'] = pd.Series(x_values).apply(lambda x:
          ↪  np.argmin(x))
221      feature_df['y_argmin'] = pd.Series(y_values).apply(lambda x:
          ↪  np.argmin(x))
222      feature_df['z_argmin'] = pd.Series(z_values).apply(lambda x:
          ↪  np.argmin(x))
223
224      # absolute difference between above indices
225      feature_df['x_arg_diff'] = abs(feature_df['x_argmax'] -
          ↪  feature_df['x_argmin'])
226      feature_df['y_arg_diff'] = abs(feature_df['y_argmax'] -
          ↪  feature_df['y_argmin'])
227      feature_df['z_arg_diff'] = abs(feature_df['z_argmax'] -
          ↪  feature_df['z_argmin'])
228
229      # index of max value in frequency domain
230      feature_df['x_argmax_fft'] =
          ↪  pd.Series(x_values_fft).apply(lambda x:
          ↪  np.argmax(np.abs(np.fft.fft(x))[1:51]))
231      feature_df['y_argmax_fft'] =
          ↪  pd.Series(y_values_fft).apply(lambda x:
          ↪  np.argmax(np.abs(np.fft.fft(x))[1:51]))
232      feature_df['z_argmax_fft'] =
          ↪  pd.Series(z_values_fft).apply(lambda x:
          ↪  np.argmax(np.abs(np.fft.fft(x))[1:51]))
233
234      # index of min value in frequency domain
235      feature_df['x_argmin_fft'] =
          ↪  pd.Series(x_values_fft).apply(lambda x:
          ↪  np.argmin(np.abs(np.fft.fft(x))[1:51]))
```

```
236     feature_df['y_argmin_fft'] =
        ↪  pd.Series(y_values_fft).apply(lambda x:
        ↪  np.argmin(np.abs(np.fft.fft(x))[1:51]))
237     feature_df['z_argmin_fft'] =
        ↪  pd.Series(z_values_fft).apply(lambda x:
        ↪  np.argmin(np.abs(np.fft.fft(x))[1:51]))
238
239     # absolute difference between above indices
240     feature_df['x_arg_diff_fft'] = abs(feature_df['x_argmax_fft'] -
        ↪  feature_df['x_argmin_fft'])
241     feature_df['y_arg_diff_fft'] = abs(feature_df['y_argmax_fft'] -
        ↪  feature_df['y_argmin_fft'])
242     feature_df['z_arg_diff_fft'] = abs(feature_df['z_argmax_fft'] -
        ↪  feature_df['z_argmin_fft'])
```