Requirements, preparations and implementation of an exemplary machine learning pipeline for HCI

Master's thesis by

Fabian Kirsch

First Supervisor:

Prof. Dr.-Ing. Matthias Rötting

Second Supervisor:

Sarah-Christin Freytag, M.Sc.

Human Factors

Faculty V

Technical University of Berlin

16.09.2019

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den _____

_____

Unterschrift

Abstract

The goal of this master's thesis is to explore machine learning methods suitable for enhancing human computer interaction (HCI). Fundamental concepts of machine learning are introduced and the general composition of a machine learning pipeline is explained. An entire machine learning pipeline was implemented focusing on methods that require little data-specific adaption and potentially generalize well to a wide range of use cases in the HCI domain. For the modeling layer an LSTM (a recurrent neural network) was chosen and different architectures and configurations were tested and compared. A public human activity recognition (HAR) time series data set was used for testing the pipeline. To enhance understanding for the reader the machine learning pipeline was integrated into the digital version of this thesis and an entirely reproducible script is provided. The best performing model has an accuracy of ~90% on the HAR data set, which already provides strong evidence for the benefits of generalizable machine learning methods in HCI. Specific suggestions for further pipeline enhancements are presented. A key learning from this thesis is that the field of machine learning is vast and understanding even one type of algorithm well requires significant time. Much time was spent on manually finding a suitable architecture and configuration for the LSTM. Therefore, some frameworks that automate this process are recommended given that sufficient computational resources are available.

Zusammenfassung

Das Ziel dieser Masterarbeit ist es Methoden des Machine-Learnings zu untersuchen, die sich zur Verbesserung von Human-Computer-Interfaces (HCI) eignen. Grundlegende Konzepte des Machine-Learnings werden vorgestellt und der Grundaufbau einer Machine-Learning-Pipeline wird erklärt. Zudem wurde eine vollständige Machine-Learning-Pipeline implementiert mit dem Fokus auf Methoden, die wenig manuelle Anpassung für den spezifischen Use Case benötigen und dadurch leicht auf andere Use Cases übertragbar sind. Ein LSTM (eine Form des Recurrent Neural Network) wurde implementiert und verschiedene Architekturen und Konfigurationen wurden getestet und verglichen. Ein öffentlich verfügbarer Human-Activity-Recognition (HAR) Datensatz wurde zum Testen der Pipeline verwendet. Um dem Leser das Verstehen einer Machine-Learning-Pipeline zu erleichtern wurden Text und Software Code in der digitalen Version der Masterarbeit zu einem komplett reproduzierbaren Skript integriert. Die Accuracy des am besten funktionierenden Models auf dem HAR Datensatz ist um die 90%. Dies zeigt eindeutig das Potential von generalisierbaren Machine-Learning Methoden zur Verbesserung von HCI. Zudem werden konkrete Verbesserungsvorschläge für die Pipeline vorgestellt. Eine zentrale Lernerfahrung aus dieser Arbeit ist, dass das Feld des Machine-Learnings gewaltig groß ist und auch nur eine Art von Algorithmus zu verstehen schon sehr zeitaufwändig ist. Viel Zeit wurde investiert um manuell eine gute Architektur und Konfiguration für das LSTM zu finden. Daher werden für den Fall, dass ausreichend Rechenkapazitäten vorhanden sind einige Frameworks empfohlen, die diesen Vorgang automatisieren.

Contents

List of Figures

List of Tables

1 Introduction

In 1820 about 90% of the world population lived in extreme poverty, in 1950 about 60%, in 1980 about 45%, in 2000 about 30% and in 2015 already less than 10% (Bourguinon & Morrison, 2009). Increasing automation since the industrial revolution and the resulting exponential increase in productivity and economic growth continues to move people out of poverty (Bolt & Van Zanden, 2014). People can afford better shelter, food, medical care and education (Lutz, Butz, & Samir, 2017; Peltzman, 2009), while the working hours keep decreasing (Feenstra, Inklaar, & Timmer, 2015). While at first simple mechanical processes in seldom changing environments where automated, today intelligent machines take over more complex processes and human and machine work are more integrated (Noble, 2017). This introduces a new problem, because in this work setting humans and machines need to exchange a lot more information. While machines can efficiently exchange lots of information through digital networks (Dixit, Lannoo, Colle, Pickavet, & Demeester, 2015), humans are very slow at communicating information from their brain to a machine. They can type something into a keyboard, push buttons and more recently they can also tell a machine what to do (Sheridan, 2016). Constantly communicating with a machine interferes with the human's actual task (Cook, Cranmer, Finan, Sapeluk, & Milton, 2017). In addition, communicating unconscious or continuously changing mental states (Finkelstein, 1999) is not even possible. A solution to this is that a machine collects data about the human through sensors and makes sense of this data itself. This way the machine can receive information from the human without the human having to tell the machine. The goal of this master's thesis is to gain more insight into machine learning methods that can help machines recognize human activity and states from sensory data. This knowledge can then help to build machine interfaces that are better adapted for human machine interaction.

2 Theory

2.1 Machine learning in human factors

In human factors system designs are optimized for human well being and general system performance (IEA, 2019). Machine interfaces are designed based on human's perceptual, cognitive and physical abilities and limitations. For example, concerning the perception it is important to know the colors or tones a human can typically distinguish. Regarding the cognitive abilities it might be important to know how many items can be stored in working memory and for how long can they be stored there, how long is the typical attention span or for how long can humans usually perform a specific task until they become tired. The physical abilities determine for example how precisely an object can be grasped or how much weight can be carried for a particular time period without fatigue. And to understand how quickly a human can react to a particular stimulus all three abilities need to be considered (Proctor & Van Zandt, 2018).

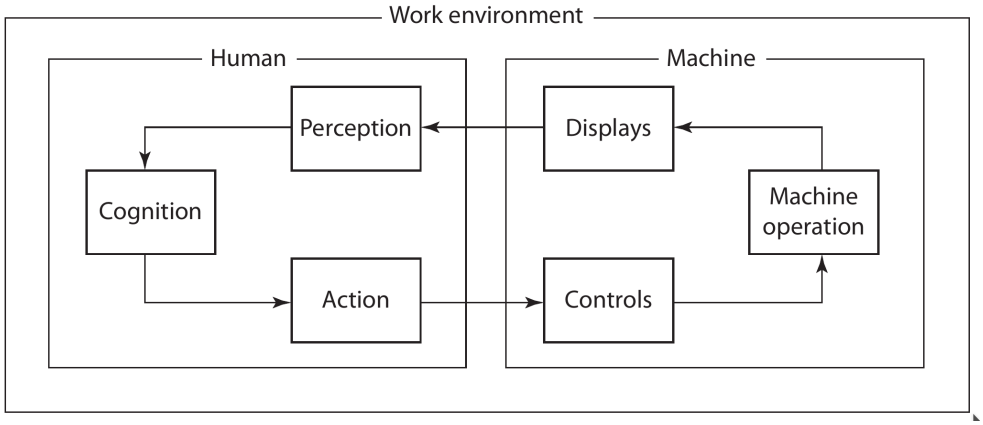

Figure 1: A traditional representation of a human-machine system (Image source: Proctor & Van Zandt (2018), p. 11).

In the traditional human-computer system shown in Figure 1 the machine can only adapt to the human through actions the human performs on the

machine's controls. This limits the overall system performance in several ways. If the human performs tasks other than communicating with the machine those tasks might be interfered with or paused (Cook et al., 2017). If the human wants to communicate a cognitive state to the machine, translating this cognitive state into an action introduces a delay. Also, a human can only communicate states or actions that the human is actually aware of. Finally, the amount of information a human can communicate to a machine is very limited and is a bottleneck in the information transmission from human to machine (Suchman, 1987; Tufte, Goeler, & Benson, 1990).

To increase the amount, timeliness and precision of the information transmission from the human to the machine the 'Control' element in Figure 1 can be replaced or extended with sensors. This way the machine can gather information about the human activities and cognitive states without requiring any 'Action' from the human. Modern cars for example give a warning if the seat belt is not fastened and the weight on the seat crosses a defined threshold (U.S. patent No. 4849733A, 1989). This is a simple example and the conditions for the warning are defined in the software or electrical circuits of the car and will likely work for most humans. However, when using sensors to recognize more complex behaviors or mental states like attention, emotions, preference or fatigue pre-defining the conditions for particular states or activities in software or hardware might be impossible. Also individual differences might be so large that the conditions would need to be adapted for every person. This is where machine learning can facilitate human-computer interaction. In machine learning algorithms learn the relationships from data collected through sensors of particular mental states or activities (Bishop, 2006; Koza, Bennett, Andre, & Keane, 1996). These relationships are represented in a model, which can then be used by the computer to recognize complex activities or mental states (Müller et al., 2008; Rani, Liu, Sarkar, & Vanman, 2006; Rautaray & Agrawal,

2015). Today various sensors and brain-imaging techniques are available to collect data about mental and physical states and activities (Jacob & Karn, 2003; Mukhopadhyay, 2015; Tan & Nijholt, 2010).

2.2 Machine intelligence

In machine learning a computer looks at data and learns the relationships in that data – it builds a model of that data (Bishop, 2006). Like the human brain builds a model of the world from data it receives through the sensory organs. A child in Northern Europe learns that something very light, green and round might be a grape, something heavier and red might be an apple, and something very long and yellow a banana. It learns this by perceiving the shape, color and weight of the fruits through the visual sense and touch combined with a person telling the child the name of the fruit – the label. If the child sees something long and yellow in the future, it will know that this is banana. The same a computer can do if provided with a list of weight, color, height and width of each object as the input data and the name of the object (the label) as the output. Having built a model of these objects the computer will be able to give the correct label of the object when given the weight, color, height and width. Both, the computer and the brain are cognitive systems: they learn from experience – build a model – and thereby can modify their behavior (Hollnagel & Woods, 2005). To understand better how intelligent machines can be integrated into human societies one needs to know what capabilities machines currently have and will likely have in the future.

How intelligent are machines compared to humans and how intelligent will they be? Among many other definitions intelligence is defined as a memory system that can make predictions (Hawkins & Blakeslee, 2007). The human brain stores information and makes predictions, which include classifications

like the identification of a banana – so humans are intelligent. This also applies to a computer that does machine learning – so also computers are intelligent. Humans have general intelligence because their cognitive ability allows them to solve general problems (Simon & Newell, 2017) and to perform a large variety of tasks from navigating their own body, speaking and understanding text to driving a car (Deary, 2001; Gray & Thompson, 2004; Mackintosh & Mackintosh, 2011). To a lesser degree also animals have this general intelligence (Reader, Hager, & Laland, 2011). However, machines until now can only solve specific problems and perform specific tasks like either driving a car or playing chess or speaking and are considered to have narrow intelligence (Legg & Hutter, 2007). The dominant opinion in the machine learning community is that it is only a matter of time until machines will have general intelligence and even super intelligence, i.e. being more intelligent than humans (Bostrom, 1998). Machines with general intelligence will be more flexible in the tasks they can perform and be able to solve general problems.

For machines to develop general intelligence, will they also become or need to become conscious during this process? It seems that the primary function of consciousness is to integrate information that would otherwise be independent (Baars, 2002; Seth, Izhikevich, Reeke, & Edelman, 2006), and might therefore be a key factor in general problem solving. One of the common perceptions of consciousness in humans is 'the experience of self' and 'knowing that one knows' (Farthing, 1992). Related to this is the notion of qualia (Latin for 'what kind of'), which describes subjective individual experiences like the pain of a headache or the green color of a leaf. The materialistic world view holds that matter is the fundamental substance in nature (Novack, 1965). That consciousness emerges from matter seems counter intuitive to many and is called the 'hard problem of consciousness' (Chalmers, 1995). There are multiple scientific theories on consciousness (Crick & Koch, 2003; Dehaene, Changeux,

Naccache, Sackur, & Sergent, 2006; Parvizi & Damasio, 2001; Tononi, 2004). The integrated information theory (IIT) (Tononi, Boly, Massimini, & Koch, 2016) follows a computational approach and is most helpful when trying to understand consciousness in machines. The IIT starts from the conscious experience, regarding it as the only thing that is actually certain – as René Descartes already stated in the 17th century: "I think, therefore I am" (Burns, 2001). Using brain imaging techniques like EEG, fMRI, or PET (Di Perri et al., 2016) one can see what happens in the brain when a human has a conscious experience – the 'neural correlates of consciousness' (Koch, 2004). The IIT describes the underlying physical structures and processes of a consciousness experience in so called 'postulates'. These postulates can then be used to determine if a physical system, including machines, has consciousness based on its structure and processes.

The physical structures on which humans and machines process information currently differ very much. Humans, who miss parts of the thalamocortical system, usually have a reduced conscious experience and reduced general intelligence (Tononi & Edelman, 1998). However, people can miss the entire cerebellum, but have a normal conscious experience. The neurons in the thalamocortical system are highly connected including feedback loops, while the neurons in the cerebellum show only very few connections. The thalamocortical system helps us to solve general problems, while the cerebellum is very specialized and needed for human's fine motor skills (Arshavsky, Gelfand, & Orlovsky, 1983; Granger & Hearn, 2007). Unlike the human thalamocortical sytem, today's machines compute information mostly serially (Von Neumann, 1945), doing one computation after the other and without integrating any of that information or feeding it back into the computation process. This might be an explanation for why machines today have narrow intelligence, but don't yet show behavior that would be considered generally intelligent

or suggesting a conscious experience. On specific tasks, where integration of information is not essential computers already outperform humans by far: the number of computations that an ordinary computer chip can do (~ 3GHz, i.e. 3000000000 computations per second) is many orders of magnitude higher than that of a single neuron (max 0.5 - 1KHz, i.e. 500 - 1000 action potentials per second), which allows them to process information much quicker than a human. Another key difference between brains and computers is that in brains the processing units (neurons), and memory (connections between neurons) are in the same place, while in computers these are separate. The CPU is doing the computations while the data is only temporarily loaded into the CPU cache and is generally stored on a hard drive, known as the von Neumann architecture (Von Neumann, 1945). However, prototypes of brain-inspired architectures for computers are being developed (Boybat et al., 2018). These integrate the processing and memory units, making the processing of data faster and less energy intensive and potentially also pave the way for architectures that allow for integration of information and with that for more generally intelligent and conscious machines.

2.3 Core methods and approaches in machine learning

In this section several methods and approaches will be presented that are important in developing a machine learning pipeline. Fundamental methods and approaches are presented first, more specific ones last.

2.3.1 Uncertainty and iteration

The process of developing a machine learning pipeline is iterative and the outcome is uncertain. In contrast, software development can be planned much better, because with enough experience the developers can tell if something can be built and also give an estimate how much time it will take them. In machine

learning a set of tools is available, but a machine learning engineer never knows what information the data holds and if algorithms will be successful at fitting a model to the data. Finding a well fitting machine learning pipeline is therefore an iterative process, during which different methods and algorithms are combined and tested in different ways.

2.3.2 Algorithms and models

An algorithm is an abstract and clearly defined way to solve a certain problem or take a decision (Rogers Jr, 1987). Algorithms define how a software decides what to do given a certain input. Algorithms can also be simple and mechanical like a thermostat on a radiator. It always behaves the same way given the same input unless reprogrammed or rebuilt. In machine learning algorithms are not used to define specific behavior, but to fit a model to the observed relationships in data (Bishop, 2006). What decision is being taken in the future (in machine learning this is usually called inference) depends on what has been in the past. In this case the particular algorithm defines only what process is applied to learn the structures present in the data and how decisions in the eventual model are represented. During the process of fitting the model the algorithm adapts the parameters of the model so that the model will eventually reflect the relationships in the data and be helpful in decision making. This fitting process can also be initialized with certain parameters, called hyper-parameters. The hyper-parameters define the complexity and architecture of the eventual model as well as the flow of the fitting processes itself.

2.3.3 Learning approaches

In supervised learning a model is built based on finding relationships between inputs and outputs (Russel & Norvig, 2010). The algorithm receives

a data set which contains a defined output for all inputs. The outputs can be either continuous, e.g. the length of a fruit or categorical, e.g. the label "banana". The former case is called regression, the latter classification. Semi-supervised is a variant in which not for all inputs outputs are provided.

In unsupervised learning only inputs are provided to the algorithm, which then tries to find similarities among these and cluster them in some way (Hinton & Salakhutdinov, 2006). For example, providing only width and weight of a set of 100 fruits composed of bananas, apples and grapes, the algorithm should group all data entries in 3 clusters corresponding to the 3 fruits. Another example would be to detect anomalies or outliers in a data set. The result would be a single cluster containing the normal observations with abnormal observations lying outside that cluster.

In reinforcement learning the algorithm is provided with one or multiple outcomes that should be reached in a particular process and with a range of possible input values needed for that process (Sutton & Barto, 1998). The algorithm continuously adapts the inputs based on the outcome – successful strategies are reinforced. This type of algorithms is used when a computer should learn a game like chess or find the most efficient configuration for a power plant that. In chess the outcome for an input (moving a chess piece) is only clear in the end of the game (win or loss). Depending on the type of power plant the efficiency of the power plant will show with a certain delay. Computer simulations of real word processes are a common way in reinforcement learning to gain insights into the optimal configurations more quickly (Silver et al., 2018).

See Section 8.4 in the appendix for which algorithms are contained within each of these learning approaches and how to find a suitable algorithm.

2.3.4 Loss functions, optimizers and performance metrics

Loss functions are used to reflect how well the model is fit to the data (Wald, 1950). For example when trying to fit a straight line (the line reflects the predicted values) to a cloud of points (this cloud reflects the actual values) the loss function could be the mean of all squared distances of all points to the line on a certain dimension, which is called the mean-squared-error loss function (see Table 1 for details and more examples). If the line lies outside the point cloud the loss will be very high and lower if it lies within the point cloud. The minimum of the loss function is to be assumed the best fit for the model. Assuming the model has only two parameters, the loss function can be thought of as mountains. The altitude of the mountains reflects the loss, the north-south dimension reflects one parameter and the east-west dimension the other. One starts the search for the valley in some random location and then starts to move around trying to find the bottom of the lowest valley, which reflects the minimum of the loss and the best model fit (see Figure 2).

Table 1: Loss functions for regression and classification in a supervised learning approach.

Type	Name	Function
Regression	Mean Squared Error (MSE)	$(\sum_{i=1}^{n}(y_i - \hat{y}_i)^2) * \frac{1}{n}$
Regression	Mean Absolute Error (MAE)	$(\sum_{i=1}^{n}\|y_i - \hat{y}_i\|) * \frac{1}{n}$
Classification	Cross Entropy Loss	$-(y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i))$

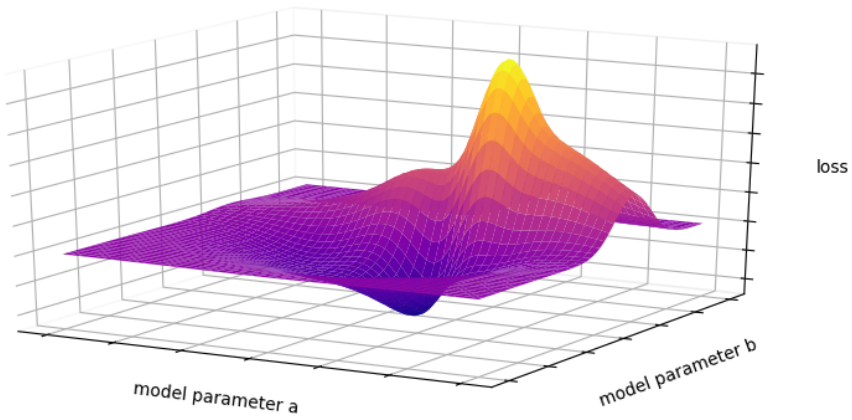

Figure 2: Loss as a function of model parameters a and b. The best model fit is assumed to be at the minimum of the loss (darkest in this figure).

Optimizers are a set of algorithms, that decide how to tune the parameters of the model based on the output from the loss function so that eventually the minimum of the loss function is found. This includes both the direction of change (positive or negative) of the parameter as well as the amount of change. In the mountain analogy the optimizer decides which way to go and how far to go before checking again the altitude. Of course the obvious direction is to go down the mountain to the valley. So a very simple optimizer could just check all the gradients of its current location and move along the steepest negative

gradient until reaching the lowest point where all gradients are positive and there is no way down. But what if just ended up in a small pit and the valley is still a long way to go or if the valley behind the next mountain ridge is even lower than the current valley? In technical terms these are local minima and if the optimizer stops changing the model parameters here, the best model fit will not be found.[1] Optimizers are therefore usually designed in more complex ways, so they also do bigger jumps and don't easily get stuck. In practice models can have millions of parameters to tune, which makes this task a lot more difficult.

Sometimes additional performance metrics are used besides the loss function to evaluate the model after the fitting process. Different loss functions reward or punish different structures in the data like the presence of outliers. So checking the outputs of other loss functions can help in understanding how well the model will be able to solve the problem at hand. More importantly, some metrics (e.g. accuracy) that are intuitive to humans are not smooth and can therefore not be used as loss functions because the optimizing algorithms cannot optimize it. A proxy loss function is therefore used that the algorithms can optimize. The human however will still want to look at the intuitive metric afterwards to understand how well the model is fit to the data.

2.3.5 Over-fitting and generalizability

Optimizing the loss function too much also has a downside. If the algorithm can handle all the complexity in the data there is a danger that the random noise in the training data is also modeled. This means the model can perfectly predict the outputs in the training data even though there is some random variance in the data that is actually not predictable. When applying the model to new data the model will fail, because the random noise in the new data

[1] An interactive visualization of different optimizers: https://emiliendupont.github.io/2018/01/24/optimization-visualization/ (2019-06-06)

will be different then in training data. This is called over-fitting the model (Domingos, 2012). Therefore, a common practice in evaluating the performance of a machine learning pipeline is to use different data for testing the model than for training the model.

The models are trained (i.e. model parameters are adjusted) using a training set. The model's performance is then checked on a test set to make sure that the model did not over-fit on the training data. However, testing different algorithms, algorithm-architectures and hyper-parameters of algorithms the final model might be over-fitted to the test data as well. Therefore a validation set is kept, which will only be used once on the final model. The performance of the model on the validation data will then be an accurate reflection of how the model performs on new data.

To reduce over-fitting during training regularization can be used to keep the parameters within certain limits or do not allow them to change too much (Friedman, Hastie, & Tibshirani, 2001). In the mountains analogy this would mean one wants to reach the valley, but one does not want to descend into some deep crevice as this is too specific.

2.3.6 Feature engineering

It can be beneficial to make certain aspects of the data more salient by manually generating new input variables from the provided input variables (also called features) - this is called feature engineering (Domingos, 2012). This is a manual process and domain knowledge is helpful to know what aspect of the data are important to predict the outcome. For example to highlight interactions between certain variables, these can be multiplied with each other or pair-wise correlations for variables can be computed. Other examples are to re-code a categorical variable that contains more than 2 labels into a single variable for each category, called 'one-hot encoding' or 'dummy variables'.

These new features can then be used along with the provided features for the training of the algorithm. Some algorithms are also capable of extracting these feature themselves not requiring any manual work. This is called representation learning (de Jesus Rubio, Angelov, & Pacheco, 2015).

2.3.7 Scaling and normalization

For many algorithms to perform well it is essential that the features have a similar center and variance. For example if feature A ranges from 0.1 to 0.2 and features B ranges from 10 to 1000 the algorithm might mistakenly interpret feature B as a better predictor. Therefore features are usually scaled to have a similar center and spread. A common method is to scale variables that all values are either within -1 and 1 or 0 and 1. Another method is to normalize the features, i.e. each feature has a center of 0 and a variance of 1 (Friedman et al., 2001).

2.3.8 Dimensionality reduction

In high dimensional spaces (usually more than 30 features) it is harder to find significance because the data becomes sparser. This also referred to as the curse of dimensionality (Verleysen & François, 2005). There are two different approaches to solve this: feature selection and feature projection.

In feature selection the original features are used, but features that do not contribute significantly to a better model fit are removed (Friedman et al., 2001). For example in linear regression, there is forward selection where features are added one after another starting with the strongest ones and each time it is checked if the model fit improved significantly. Once a feature does not improve the model significantly anymore the procedure is stopped and no new features are added. In backward selection all features are added in the beginning and then removed one by one until the model fit drops significantly.

In feature projection the high dimensional feature space is projected into a lower dimensional features space using methods like principal component analysis (PCA), linear discriminant analysis (LDA) or independent component analysis (ICA) (Friedman et al., 2001). These methods condense the information present in the many features of the high dimensional space to fewer features, that are orthogonal to each other, i.e. don't share information. Which of these methods to choose usually requires some data exploration and domain knowledge. There are also fully automated methods like autoencoders (Hinton & Salakhutdinov, 2006), which is a type of neural network.

2.3.9 Offline vs online learning

Once a model has been built with some initial training data inferences can be made on new data. However, in some cases the relationships in the new data will change over time. The initially trained model will not be an accurate presentation anymore of the relationships present in the latest data and the performance of the model will get worse over time. In this case the model needs to be retrained or updated. In offline learning the models gets retrained once in a while and then this retrained model will be used for inference from then on until the another model gets trained including the latest data. In contrast, in online learning for every new incoming data point an inference will be made and the data point will be also be used to update the model immediately. This way the model continuously adapts to the data.

2.3.10 Filtering noise in continuous signals

If the data is a continuous signal from a physical sensor it might include noise. Noise in the signal will make it harder for an algorithm to find the relevant patterns. The noise can be random jitter due to measurement error of the sensor and not hold any information. Noise can also be another signal

that is present in the data, but not relevant for the current problem. Using filters the relevant signal can be emphasized or extracted and the noise can be reduced or removed.

2.3.11 Storage types

If data should rather be stored in files or in a database depends on the use case. Databases allow to quickly read data, so if speed is the top priority a database might the better choice. However, databases also have a computational overhead that requires additional computational resources and they need be setup which takes time. Relational databases also require the data to come in a previously specified schema, which can be cumbersome to setup when dealing with raw data. Files are very flexible and require no set up and are therefore the default choice for doing research and building prototypes.

2.3.12 Batch processing vs online processing

In batch processing new observations are first stored and only passed to the pipeline after a certain number of observations (n) has been collected or after a certain delay (d). If the output of the pipeline is only needed once a day, it can be cheaper with regard to computational resources because the machine running the pipeline can be turned on only once a day.

In online processing new observations are passed immediately to the machine learning pipeline once they are available. This setup is usually chosen if the output of the pipeline is needed as soon as possible. Online processing can be thought of as a variant of batch processing with n=1 and d=0.

2.3.13 Sub-sampling

Data from a set of different sensors might be collected with a sampling rate of 100 Hz. However, if only characteristics, like temperature, which change at

a much slower pace are of interest a sampling rate of 1 Hz might be enough. Through sub-sampling, i.e. taking only every 100th data point the algorithm only needs to processes 1% of the data.

2.3.14 Testing

In software development tests are written which can automatically verify the integrity of the code base. Unit tests are used to verify that single functions work as as expected and integration tests verify the output final output of an application. This way the code can be changed and new code can be added without the need for elaborate and time consuming human testing to ensure the correct functioning of the application. In machine learning testing is even more crucial because the outcome might be very complex and not easily verifiable by a human. Therefore unit tests should be written for essential parts of a machine learning pipeline to guarantee the correct processing of the data.

2.4 General composition of a machine learning pipeline

A machine learning pipeline accepts one or more input variables, called features, and provides one or more output variables. From the collection of data to inference the data moves through three different blocks in a machine learning pipeline: extract-transform-load (ETL), pre-processing and modeling (Figure 3). These blocks are composed of different layers. A layer holds a particular statistical or data processing method, which is applied to the data. Some layers are always in the same block, some can be in different blocks and many of the layers are optional. Also, the order of the layers can be different depending on the use case. Prior to the data entering the pipeline it needs to be generated.

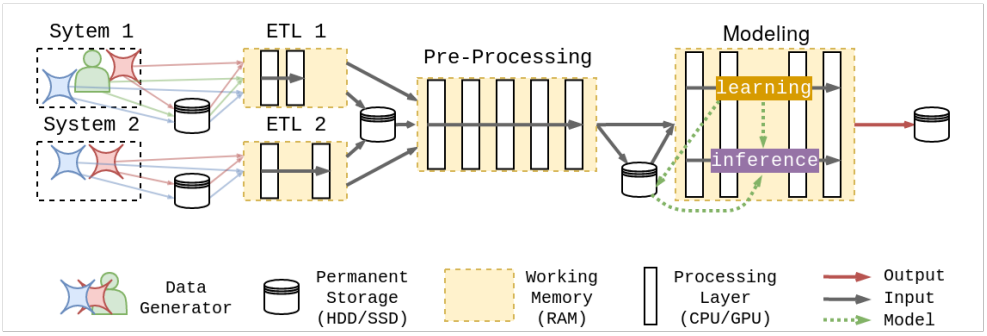

Figure 3: General machine learning pipeline composition

2.4.1 Storage

If data is not currently processed by a processing unit it needs to temporarily be stored somewhere. All the storing within the pipeline happens in-memory. This means the data is held in the machines working memory (RAM). This type of memory allows very quick write and read access and is necessary to interact with the processing units (CPU or GPU). However, working memory has a limited capacity and if the power of the machine is cut all information in working memory is lost.

Permanent storage (SSD or HDD) is slower in reading and writing and therefore not suitable for interacting with the processing units. However, it preserves information even if left without power, which makes it a save place to store data and models. Also, it has usually a much higher capacity than the working memory and can therefore hold all the data that is currently not being processed.

2.4.2 Data generation

Data can be generated through sensors that measure some physical quality or it can be generated virtually by a computer. Examples for physical qualities are light, temperature, pressure or movement - this includes buttons of a computer keyboard. Virtual data generation by a computer could be monitoring

the behavior of a software, logging the behavior of users on a website or even simulating data or randomly generating it. The generated data can then either be passed to the ETL block (Section 2.4.3) or passed to permanent storage.

2.4.3 Extract, transform, load (ETL)

In this block data from different sources is extracted, i.e. selecting the data subset that one is interested in. So not more data than necessary will be processed. Data from different sources is also merged so it can be passed to the pipeline as a single set of observations. The format of the data might need to be changed, e.g. from a dictionary-format to a columnar-format. Also the data types might need to be adapted: during data collection all data points might have been saved as strings even if the data is actually numeric. These data points will then be transformed to integer or float values. Importantly, the actual data is not changed in the ETL block, only its structure.

The ETL block is specific to every data source that should be processed by the pipeline. Therefore multiple ETL blocks can exists for example if data from multiple systems should be processed. The input to this block is received either directly from the data generators or from permanent storage. The output of this block can be passed to permanent storage or to the pre-processing block (Section 2.4.4).

2.4.4 Pre-processing

The pre-processing block can be very small or very large. In rare cases it might not exist at all. It holds all the layers that change the data in some way, but not yet build a model of the data. In this block, the data might be cleaned from noise, observations with missing values be removed, new features (columns) be generated through various feature engineering techniques or features removed by applying dimension reduction techniques. The output

of this block might look very different than the input and it could be that none of the values passed to this block is present in the output. The output of this block can be passed to permanent storage or to the modeling block (Section 2.4.5).

2.4.5 Modeling

The modeling block can contain multiple layers as different machine learning models might be combined and the output of one model serves as the input to the next model (stacked models). Initially the data is used for learning. Once the models are fitted to the data, these models are then used for inference on new data points. The models can be held in working memory or also permanently stored. Next to the models this block produces the actual output of the pipeline, i.e. some classification or prediction.

3 Methods and tools

3.1 Pipeline requirements

The aim of this thesis is to develop a machine learning pipeline for human activity and state recognition that can be applied with only minor changes to different uses cases and data sets. The pipeline should accept raw sensory data as an input and provide the correct labels for the mental or physical state or activity the human is currently performing. The training of the model can happen offline, but online inference, i.e. classifying activities and states should be possible in real-time and happen within less than a second.

3.2 Data

3.2.1 Data set

There are at least two major obstacles to developing a well working machine learning pipeline, which are often confounded. One obstacle is to find a well working implementation for the actual pipeline. The other obstacle is that the data used for developing and testing the pipeline needs to contain the patterns that one actually aims to find. If the data is too noisy or the patterns that one expects actually don't exist, there is no way for the algorithm to learn the patterns and classify the behavior correctly. The focus of this thesis lies on developing a well working pipeline implementation. Therefore, a public machine learning data set was chosen, which has been successfully used for machine learning. This way the second obstacle of bad data is avoided. The chosen data set was published by Reyes-Ortiz, Oneto, Samà, Parra, & Anguita (2016). It can be retrieved from the UCI machine learning repository.[2] The data set contains human activity recognition data from 30 participants. The participants performed different physical activities like standing, walking or

[2]Data set at UCI machine learning repository: http://archive.ics.uci.edu/ml/datasets/Smartphone-Based+Recognition+of+Human+Activities+and+Postural+Transitions (2019-06-06)

sitting. For data collection a smartphone (Samsung Galaxy S2) attached to the waist of the participants was used. The data set contains raw sensory data collected at 50Hz from the accelerometer and gyroscope of the smartphone as well as the labels of the current activity[3] or the current transition between two activities.[4] The labels refer to specific time windows of the sensor data and do not cover the entire data set. That means, there are data points which are not labeled. See Figure 4 for an example of how the data looks like. Besides the raw sensory data the data set also contains pre-processed data, but which is not used in this thesis.

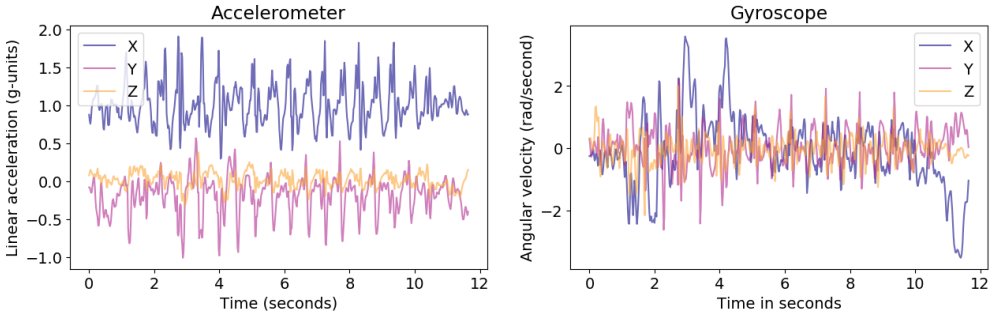

Figure 4: Raw sensory data of gyroscope and accelerometer while walking collected at 50Hz.

3.2.2 Data split

To ensure an accurate performance measurement of the tested models the data is split into a train (60%), test (20%) and validation (20%) set. The data split is based on participant ids, which were randomly assigned once to one of the three groups before the testing of implementations started. The validation set is only used to test the performance of the final model.

[3] Activities: walking, walking_upstairs, walking_downstairs, sitting, standing, laying

[4] Transitions between activities: stand_to_sit, sit_to_stand, sit_to_lie, lie_to_sit, stand_to_lie, lie_to_stand

3.3 Language and packages

The pipeline is implemented in Python 3 using standard data science packages included in the anaconda distribution for python 3.71 on Linux.[5] Additionally, the tensorflow-gpu[6] package is used, a deep learning library that utilizes both CPU and GPU for building machine learning models. Tensorflow provides only a low level API, therefore, the keras package[7] is used on top, which provides a high-level API for implementing neural networks.

3.4 Machine setup used for building pipeline

The models were trained on a local desktop computer with a Nvidia GPU. To use the GPU for training tensorflow models Nvidia's parallel computing platform CUDA[8] and Nvidia's GPU-accelerated library of primitives for deep neural networks cuDNN[9] need to be installed. It is important to install matching versions of GPU drivers, CUDA and cuDNN. The installation process can be quite cumbersome and the documentation[10] should be closely followed. A complete list of hardware specifications, drivers and essential software packages used in this thesiss are listed in Table 2.

Table 2: Hardware and Software specifications of the machine used for training the models.

Hardware/Software	Model/Version
GPU	Nvidia GeForce 970 GTX 4GB VRAM
CPU	AMD FX 8350 8-Cores
Memory	8 GB DDR3 at 1666 MHz
Operating System	Kubuntu 18.04 64-bit, kernel 4.15.0

[5]https://docs.anaconda.com/anaconda/packages/py3.7_linux-64/ (2019-06-06)
[6]https://www.tensorflow.org/ (2019-06-06)
[7]https://keras.io/ (2019-06-06)
[8]https://developer.nvidia.com/cuda-zone (2019-06-06)
[9]https://developer.nvidia.com/cudnn (2019-06-06)
[10]https://docs.nvidia.com/cuda/index.html (2019-06-06)

Hardware/Software	Model/Version
GPU OpenGL driver	Nvidia 415.27
CUDA version	10.0
Cudnn version	7.4.2

3.5 Reproducability and code structure

This entire thesis is fully reproducible including data acquisition, data pre-processing, modeling and plots. The thesis was written in markdown[11] including python code blocks. Reusable code is kept in custom python modules and imported within the python code blocks. The markdown files where first converted to jupyter notebooks,[12] then executed and then converted back to markdown files. The final PDF file was then created using pandoc.[13]

See Section 8.1 in the appendix for a reference where you can find the digital repository of this reproducible thesis. See Section 8.2.1 in the appendix for a guide on how to run parts of this thesis with minimal setup online using Google Colaboratory. See Section 8.2.2 in the appendix for how to setup a local machine and reproduce the entire thesis. For some of the crucial custom ETL and pre-processing modules unit tests were written. See Section 8.3 for how to run them.

[11] https://www.markdownguide.org/ (2019-06-06)
[12] https://jupyter.org/ (2019-06-06)
[13] https://pandoc.org/ (2019-06-06)

4 Implemented pipeline layers

In this section the actual implementation of a machine learning pipeline is described. A machine learning pipeline consists of several layers, which can be grouped into extract-transform-load (ETL), pre-processing and modeling (see Section 2.4). Each subsection describes a different layer in the pipeline. The order of the subsection reflects the order of the layers in the implementation. Some layers are always required, some are optional and some are mutually exclusive, e.g. if two models that cannot be combined. Some of the layers are specific to the data set used, but most layers are applicable to different data sets. A layer might have various configuration options or in the case of the modeling-layers also different architectures.

4.1 ETL layer: loading and splitting

The raw data is loaded from different files in permanent storage into memory and merged into a single data object. The raw sensory data consists of 60 separate data files representing data from different sessions. The labels are again in a separate file. All these files are loaded and merged into one data object. This data object is then split into a train, test and validation set based on participant ids specified in the config. If the data is not in the data folder specified in the config the data is downloaded first from the UCI machine learning repository and extracted. See Table 4 for the shape of the data after this layer.

4.2 ETL layer: sequencing

The output of this pipeline are discrete classifications, while the inputs are streams of observations sampled at 50Hz representing a continuous signal from different sensors. As each data point represents only a 50th of a second it will not provide enough information for an algorithm to classify single data

points. Classifying the entire stream is not possible either, because it contains different behaviors that belong to different classes. It is therefore necessary to slice the stream into separate sequences (or windows), which can then be classified. When sequencing the data, two parameters can be set. The length of the sequence and distance between the starting points of succeeding sequences. The sequence should be long enough to reflect patterns related to the different behaviors to classify. The maxmium length depends on how quickly the behavior changes and which sequence lengths the modeling algorithms can deal with. For example, LSTMs can deal with sequence lengths of up to 300. The stepsize depends on how much overlap of sequences makes sense and how often a classification is needed. See Table 4 for the shape of the data after applying a *sequence length* of *128* and a *sequence stepsize* of *64* (parameters taken from authors of data Reyes-Ortiz et al. (2016)).

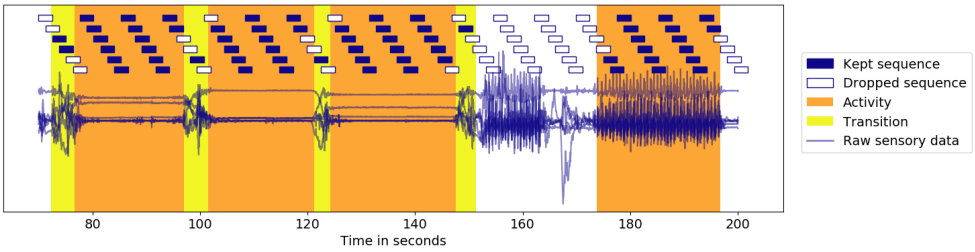

Figure 5: Sequencing of raw data. Sequences are 128 samples long and overlap by 50%, i.e. every sequence shifts 64 samples. Sequences are displayed in multiple rows so they are visually distinct. The background color indicates if the current data point is labeled as an activity or a transition. If the background is not colored no label is available. Only sequences with a distinct label are kept.

4.3 ETL layer: sequence cleaning

Next, sequences that contain unlabeled observations and sequences that contain observations with different labels are dropped (sequences with white fill in Figure 5). Now, each sequence has a distinct label. When applying the model to new data this layer is not applied because no labels will exist.

However, for training it is important that the sequences have a distinct label. See Table 4 for the shape of the data after this layer.

4.4 ETL layer: separating input and output features

The input and output features need to be passed to the models separately, so the data sets (train, test & validation) containing the sequences need to be split to input features (x) and the output feature (y). The output feature sequences now hold a lot of redundant information as they contain a label for each observation in the sequence. However, all of these labels are the same within a sequence since after the cleaning (Section 4.3). Therefore, the length of the output vector within each sequence can be reduced from *128* (sequence length) to *1*. see Table 4 for the shape of the data after this layer.

4.5 ETL layer: label selection

Once all sequences have a unique label, we can filter the labels we want our algorithm to train on, i.e. in this case select particular activities or transitions. See Table 3 for an overview of all activities and transitions present in the data. See Table 4 how the data set size changes after keeping only the activities and dropping the transitions.

Table 3: Overview of activity and transition labels present in the data.

Type	Label	Name
activity	1	walking
activity	2	walking upstairs
activity	3	walking downstairs
activity	4	sitting
activity	5	standing
activity	6	laying

Type	Label	Name
transition	7	stand to sit
transition	8	sit to stand
transition	9	sit to lie
transition	10	lie to sit
transition	11	stand to lie
transition	12	lie to stand

4.6 ETL layer: Recoding output to binary features

The multi-categorical output feature is recoded into binary features - one feature for each category in the original feature. This is called one-hot-encoding and advisable for many algorithms to perform well in multi-classification tasks. The original output feature is dropped. See Table 4 for how the data set size changes after applying this layer.

4.7 Pre-processing layer: noise reduction in input

The raw data contains noise that is removed using a median filter as the authors of the data did (Reyes-Ortiz et al., 2016). See Figure 6 and Figure 7.

4.8 Pre-processing layer: separating bodily and gravitational acceleration

The data from the accelerometer holds both body and gravitational components, which are easier to distinguish during modeling if separated before (Figure 8) (Hees et al., 2013; Veltink, Bussmann, De Vries, Martens, & Van Lummel, 1996). Also this step is performed similar to the authors of the data (Reyes-Ortiz et al., 2016). See Table 4 for the shape of the data after this layer.

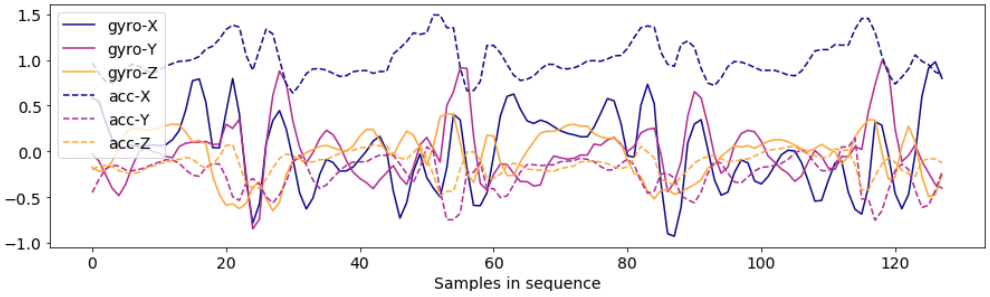

Figure 6: Raw sensory data before applying a median filter

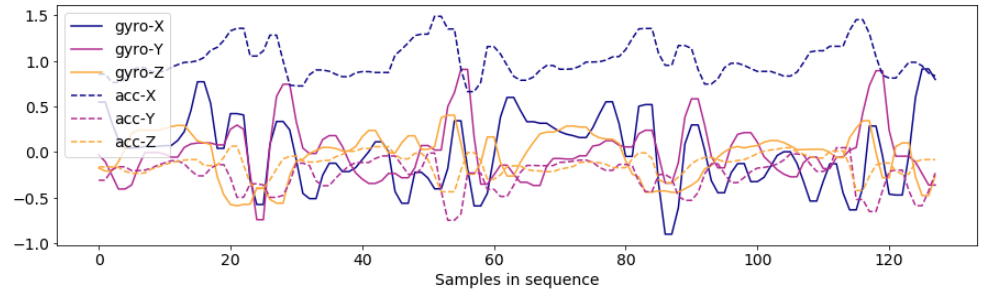

Figure 7: Raw sensory data after applying a median filter with *kernel size = 3*.

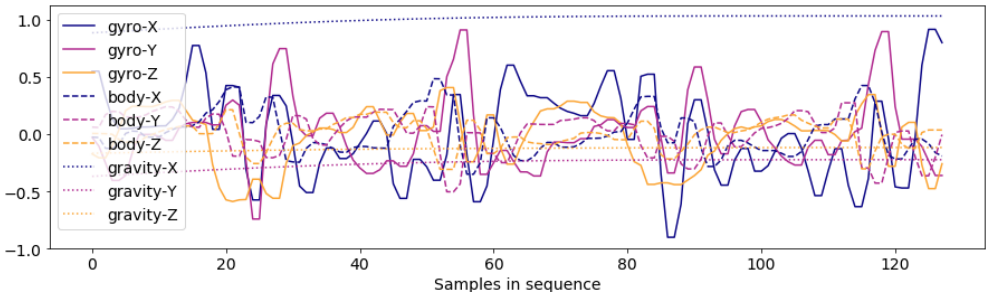

Figure 8: After separating body and gravity components from the acceleration data.

Table 4: Number of items within different dimensions of the data sets after passing through various layers in the pipeline: *o=number of observations*, *f=number of features*, *s=number of sequences*. Note, the sequenced data sets are three-dimensional. X denotes the input features, y the output features. The number of features in the one-hot-encoding equals the number of selected activities (*6* in this case). The sequence length and stepsize used for sequencing are *128* and *64*, respectively.

State and dimension labels	train set	test set	validation set
Loaded sets Xy (o,f)	(687249, 10)	(235711, 10)	(199812, 10)
Sequenced sets Xy (s,o,f)	(10684, 128, 7)	(3666, 128, 7)	(3103, 128, 7)
Cleaned sets Xy (s,o,f)	(6350, 128, 7)	(2173, 128, 7)	(1838, 128, 7)
Separate set X (s,o,f)	(6350, 128, 6)	(2173, 128, 6)	(1838, 128, 6)
Separate set y (s,f)	(6350, 1)	(2173, 1)	(1838, 1)
Activity selection X (s,o,f)	(6128, 128, 6)	(2105, 128, 6)	(1784, 128, 6)
Activity selection y (s,f)	(6128, 1)	(2105, 1)	(1784, 1)
Binary encoded y (s,f)	(6350, 6)	(2173, 6)	(1838, 6)
Body & gravity X (s,o,f)	(6128, 128, 9)	(2105, 128, 9)	(1784, 128, 9)

4.9 Pre-processing layer: normalization

All of the input features are normalized to have a *mean* of *0* and a *standard deviation* of *1*. This layer is still considered to be pre-processing, although actually a mini-model (the mean and standard deviation) is fitted to the train data set and kept to apply it to new data later including the test and validation sets.

4.10 Modeling layer: LSTM

This layer contain a particular kind of recurrent neural network using long-short-term-memory (LSTM) cells (Gers, Schmidhuber, & Cummins, 1999). Neural networks are capable of representation learning (see Section 2.3.6),

i.e. they can extract the most relevant patterns in raw data without prior manual feature engineering. Specifically, LSTMs are capable of learning patterns across long time lags, which makes them well suited for classifying the human behaviors present in the sensory data in the HAPT data set. LSTMS can have different architectures, i.e. a combination of one or more layers. Also, various hyperparameters can be set, like loss functions (a proxy perfomance function), activation functions (how input is converted within the network), optimizers (which parameters to try next), dropout (against over-fitting) and regularizers (keeping the parameters within certain limits).

5 Results

In this section various pipeline architectures are described and their performance on classifying the raw data are presented. More than 100 variants were tested during the pipeline development in total. Presenting all of them would neither be feasible nor helpful to the reader. Therefore, only variants that likely provide helpful insights to the reader are presented. The variants are presented in the order they were discovered, so the reader can follow how new insights led to new architectures and configurations. Each subsection presents a particular architecture and different configurations for each architecture are again presented in sub-subsections. The last subsection of this section validates the best performing variant on the validation set.

The primary performance measure used is accuracy. Accuracy is defined as the proportion of correct classifications compared to all classifications with a score of 1 reflecting a perfect fit of the model. Accuracy can be misleading, if class sizes are very unequal, but this is not the case in the HAPT data set, so accuracy is a save measure. For a more detailed understanding of how well particular classes can be classified a confusion matrix is presented as well. The performance of each variant is tested on the test set. The best performing pipeline will then be tested again on the validation set to understand the pipeline's performance on new data.

The models presented are non-deterministic as they are randomly initiated. To make this thesis entire reproducible a seed is set before running each variant. All variants use the default ETL and pre-processing configurations presented in the first subsection (Section 5.1).

5.1 Default ETL and pre-processing layers

This subsection gives and overview of the default ETL and pre-processing layers applied to every implementation variant (see Table 5). See Listing 1 for

the default configuration passed to the pipeline.

Table 5: Default ETL and pre-processing layers used in every variant

Type	Layer
ETL	Loading and splitting
ETL	Sequencing
ETL	Sequence cleaning
ETL	Separating input and output features
ETL	Label selection
ETL	Recoding output to binary features
Pre-processing	Noise reduction in input
Pre-processing	Separating bodily and gravitational acceleration
Pre-processing	Normalizing input features

Listing 1 Default configuration for ETL and pre-processing layers in all implemented pipeline variants.

```yaml
01_etl:
  data_set_dir: 'data/HAPT Data Set'
  download_url: 'https://archive.ics.uci.edu/ml/
  ↪ machine-learning-databases/00341/HAPT%20Data%20Set.zip'
  data_split:
    train_participant_ids: [20,  6, 22, 18, 26, 27,  3, 11, 13,
    ↪ 30, 19, 12, 10, 17, 21,  4, 14, 24]
    test_participant_ids: [16, 28,  2,  1, 23, 25]
    validation_participant_ids: [ 7,  9, 15, 29,  8,  5]
  selected_labels: [1,2,3,4,5,6]
  channel_names_prior_preprocess: ['gyro-X', 'gyro-Y', 'gyro-Z',
  ↪ 'acc-X', 'acc-Y', 'acc-Z']
  channl_names_post_preprocess: ['gyro-X', 'gyro-Y', 'gyro-Z',
  ↪ 'body-X', 'body-Y', 'body-Z', 'gravity-X', 'gravity-Y',
  ↪ 'gravity-Z']
  sequence_length: 128
  sequence_stepsize: 64
  drop_columns: ['participant_id', 'experiment_id', 'time'] #
  ↪ the columns are loaded initially, because they are needed
  ↪ to sort and group data, but should not be used for
  ↪ modeling
  group_column: 'experiment_id' # data is sequenced within these
  ↪ groups
  sample_rate: 50
02_preprocessing:
  sample_rate: 50
  median_filter_kernel: 3
  acc_columns_idx: [3,4,5] # indices of columns that contain the
  ↪ acceleration data
```

5.2 LSTM - activities only

An LSTM network with 1 or more LSTM layers and a Dense layer as the output layer is implemented. Assuming that activities are easier to model than transitions only sequences that are labeled as activities are considered in this section and transitions are removed. Weights are optimized using the ADAM optimizer (Kingma & Ba, 2014), which needs relatively little computational resources. For the loss function categorical cross-entropy is used, which generally performs best for multi-classification problems. See Listing 2 for the base configuration used for the LSTM. This first configuration was conceived based on experience, but has not been tested on this data.

Listing 2 Base configuration for LSTM

```
lstm_layer:
  units: 100
  activation: relu
  dropout: 0.2
  kernel_regularizer_l2: 0
  activity_regularizer_l1: 0
output_layer:
  activation: softmax
  kernel_regularizer_l2: 0
  activity_regularizer_l1: 0
loss: categorical_crossentropy
optimizer:
  adam:
    lr: 0.001
    beta_1: 0.9
    beta_2: 0.999
    decay: 0.0
metrics: ['accuracy']
epochs: 100
batch_size: 200
```

5.2.1 Variant: LSTM with 2 layers - base configuration

This variant uses the LSTM base configuration (Listing 2) with 2 LSTM-layers and one output layer. This variant has an **accuracy of 27.6%** on the test data set. See Figure 9 for how the model performs in classifying each label. See Figure 10 for how the accuracy and loss evolved during training.

Only labels *sitting* and *standing* are predicted. The model achieves better accuracy in the beginning, but after about 10 epochs the loss increases and stabilizes (Figure 10). The network's architecture might be too complex for this kind of data.

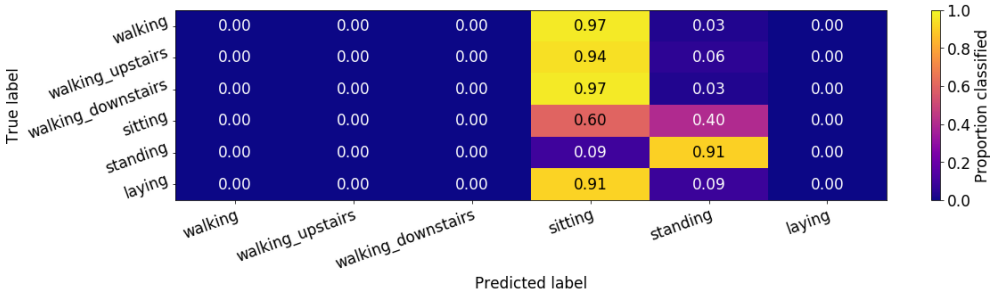

Figure 9: Confusion matrix of the predictions made by the model on the test set. The diagonal reflects the correctly classified proportions for each category.

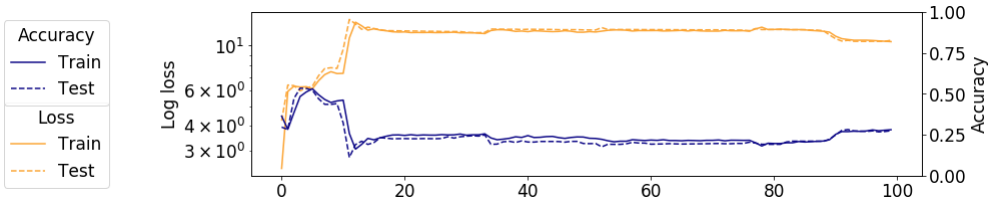

Figure 10: Accuracy and loss on train and test data sets during training of LSTM on the training data set.

5.2.2 Variant: LSTM with 1 layer - base configuration

This variant uses the LSTM base configuration (Listing 2) with 1 LSTM-layer and one output layer.This variant has an **accuracy of 19.0%** on the test data set. See Figure 11 for how the model performs in classifying each label. See Figure 12 for how the accuracy and loss evolved during training.

Although less complex than the previous model (Section 5.2.1), also in this variant the loss increases suddenly and stabilizes at a high level (Figure 12). A reason for this might be vanishing or exploding gradients (i.e. the weights of the model stabilizing at *0* or becoming huge, respectively).

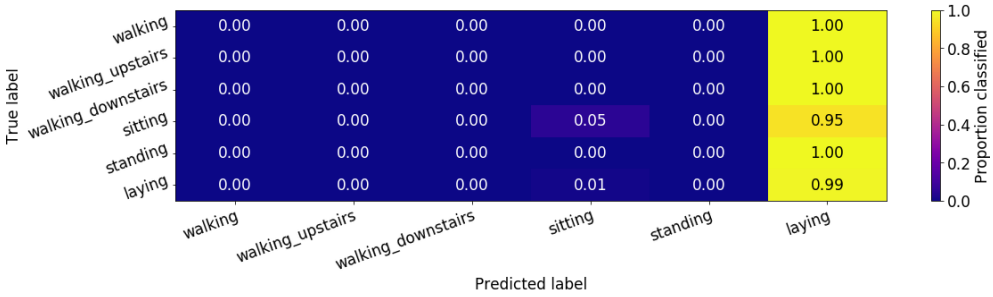

Figure 11: Confusion matrix of the predictions made by the model on the test set. The diagonal reflects the correctly classified proportions for each category.

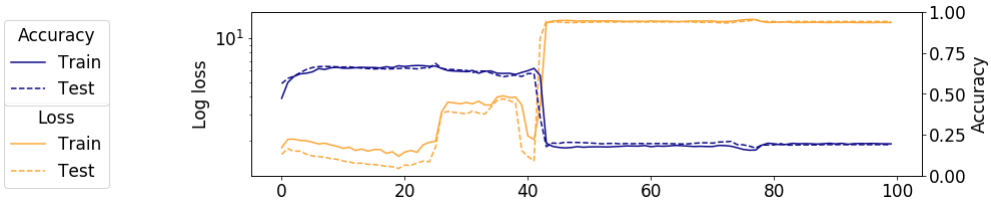

Figure 12: Accuracy and loss on train and test data sets during training of LSTM on the training data set.

5.2.3 Variant: LSTM with 1 layer - regularizers instead of dropout

This variant uses regularizers (0.001 for both l1 and l2 regularizers on LSTM and output layers) instead of dropout (set to 0.0) to control the weights of the nodes in the network (compare Listing 2). The neural network has again 1 LSTM-layer and 1 output layer. This variant has an **accuracy of 86.2%** on the test data set. See Figure 13 for how the model performs in classifying each label. See Figure 14 for how the accuracy and loss evolved during training. Apart from a brief peak the loss constantly decrease and the model can classify the activities quite well. Adding regularizers seems like an important step. Only sitting and standing are still difficult to distinguish for the model.

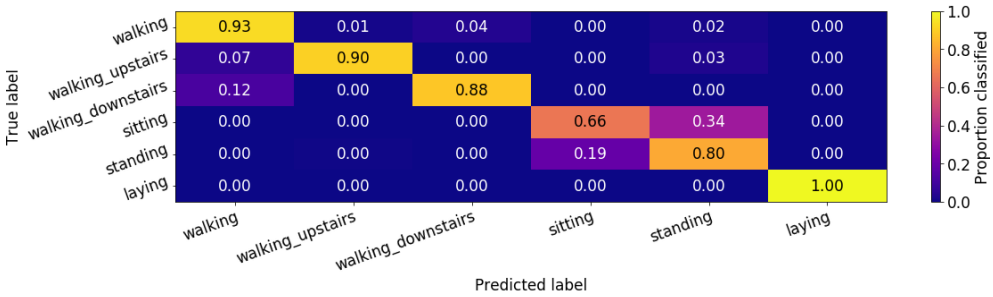

Figure 13: Confusion matrix of the predictions made by the model on the test set. The diagonal reflects the correctly classified proportions for each category.

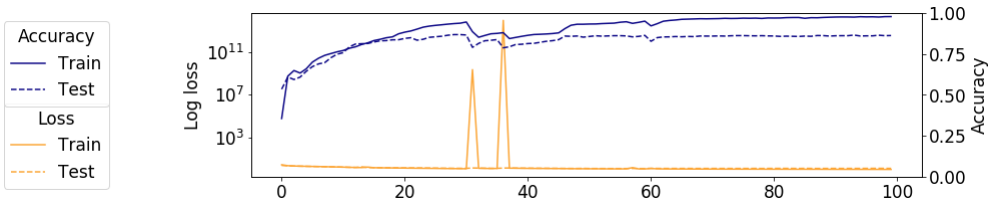

Figure 14: Accuracy and loss on train and test data sets during training of LSTM on the training data set.

5.2.4 Variant: LSTM with 1 layer - shorter sequences

This variant uses a *sequence length* of *64* (instead of *128*) and a *sequence stepsize* of *32* (instead of *64*). Also, only 50 units in its LSTM layer compared to 100 in the base config (Listing 2). It also uses regularizers (0.001 for both l1 and l2 regularizers on LSTM and output layers) instead of dropout (set to 0.0) to control the weights of the nodes in the network (compare Listing 2). The neural network has again 1 LSTM-layer and 1 output layer. This variant has an **accuracy of 88.3%** on the test data set. See Figure 15 for how the model performs in classifying each label. See Figure 16 for how the accuracy and loss evolved during training. Using shorter sequence lengths and stepsizes seems to slightly improve the model fit (compare Section 5.2.3).

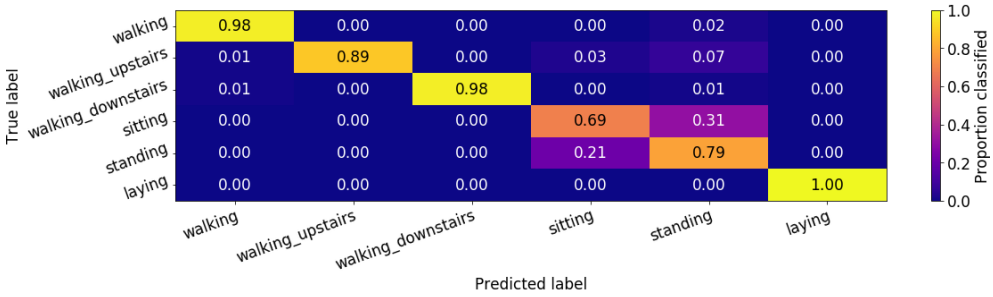

Figure 15: Confusion matrix of the predictions made by the model on the test set. The diagonal reflects the correctly classified proportions for each category.

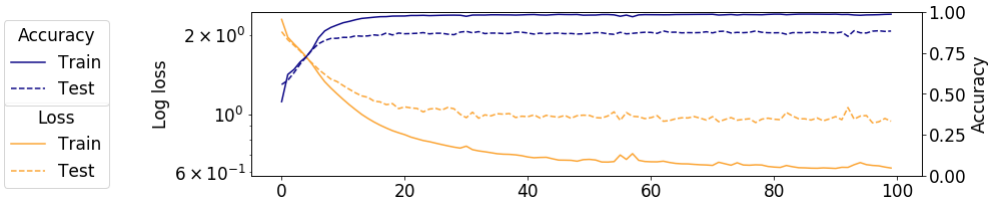

Figure 16: Accuracy and loss on train and test data sets during training of LSTM on the training data set.

5.2.5 Variant: LSTM with 1 layer - activity subset

The activities sitting and standing are removed in this variant. This variant uses only 50 units in its LSTM layer compared to 100 in the base config (Listing 2). It also uses regularizers (0.001 for both l1 and l2 regularizers on LSTM and output layers) instead of dropout (set to 0.0) to control the weights of the nodes in the network (compare Listing 2). The neural network has again 1 LSTM-layer and 1 output layer. This variant has an **accuracy of 96.6%** on the test data set. See Figure 17 for how the model performs in classifying each label. See Figure 18 for how the accuracy and loss evolved during training. Removing the ambiguous activities makes the model perform really well.

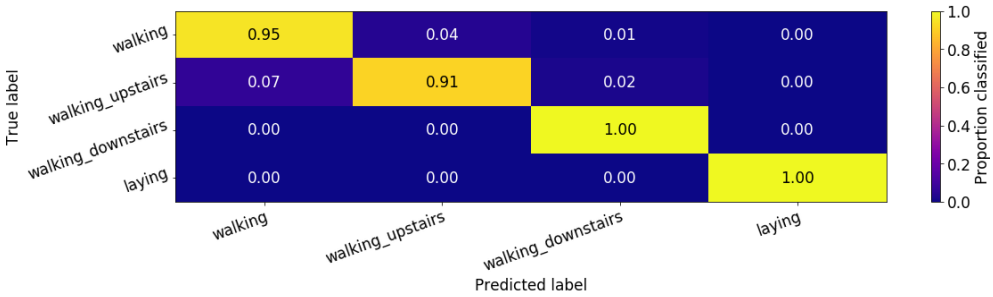

Figure 17: Confusion matrix of the predictions made by the model on the test set. The diagonal reflects the correctly classified proportions for each category.

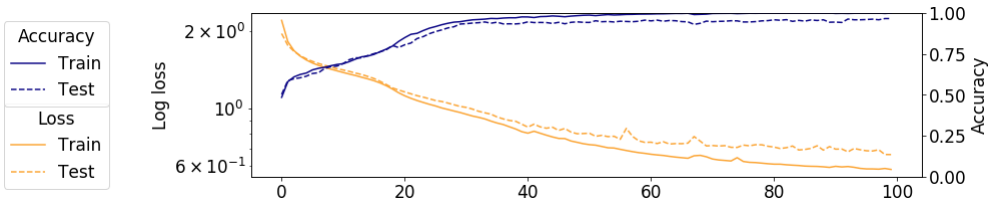

Figure 18: Accuracy and loss on train and test data sets during training of LSTM on the training data set.

5.3 LSTM - activities and transitions

An LSTM network with 1 or more LSTM layers and a Dense layer as the output layer is implemented. Both activities and transitions are classified. See Listing 3 for the base configuration used for the LSTM. This base configuration contains parameters that worked well on classifying only activities.

Listing 3 Base configuration for LSTM

```
lstm_layer:
  units: 50
  activation: relu
  dropout: 0.0
  kernel_regularizer_l2: 0.001
  activity_regularizer_l1: 0.001
output_layer:
  activation: softmax
  kernel_regularizer_l2: 0.001
  activity_regularizer_l1: 0.001
loss: categorical_crossentropy
optimizer:
  adam:
    lr: 0.001
    beta_1: 0.9
    beta_2: 0.999
    decay: 0.0
metrics: ['accuracy']
epochs: 100
batch_size: 200
```

5.3.1 Variant: LSTM with 1 layer - base config

This variant uses the LSTM base config for activities and transitions (Listing 3). This variant has an **accuracy of 71.1%** on the test data set. See Figure 19 for how the model performs in classifying each label. See Figure 20 for how the accuracy and loss evolved during training. The model struggles to classify the transitions correctly and also performs less well on the activities than when only classifying activities without transitions. A possible reason could be that only very few sequences containing are kept during sequence cleaning and the model does not have enough data to train (see Figure 5).

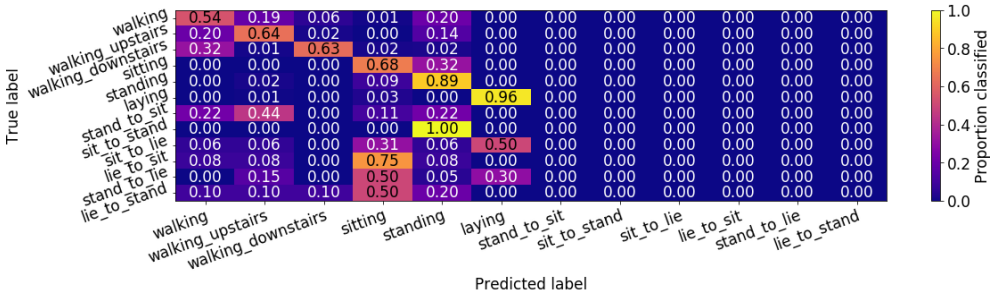

Figure 19: Confusion matrix of the predictions made by the model on the test set. The diagonal reflects the correctly classified proportions for each category.

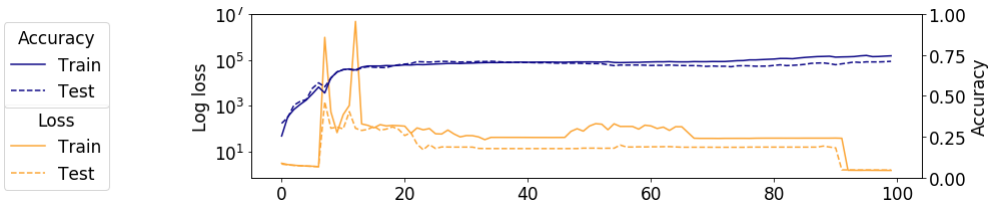

Figure 20: Accuracy and loss on train and test data sets during training of LSTM on the training data set.

5.3.2 Variant: LSTM with 1 layer - shorter sequences

This variant uses a *sequence length* of *64* (instead of *128*) and a *sequence stepsize* of *32* (instead of *64*). This variant has an **accuracy of 80.1%** on the test data set. See Figure 21 for how the model performs in classifying each label. See Figure 22 for how the accuracy and loss evolved during training. The shorter sequence length and stepsize have a significant positive effect on the model's performance. However, transitions are still not being recognized well, some of them are not recognized at all.

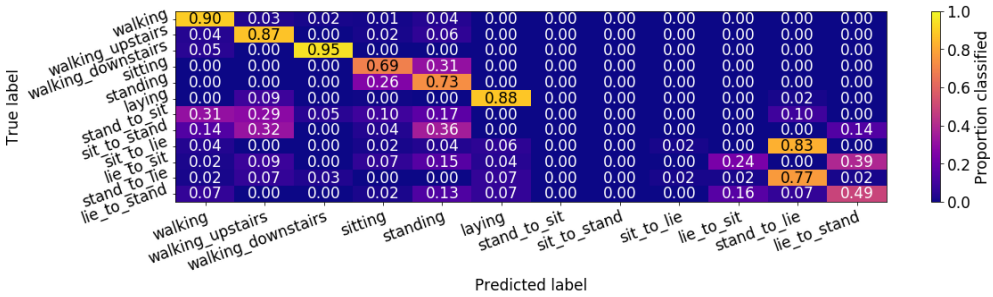

Figure 21: Confusion matrix of the predictions made by the model on the test set. The diagonal reflects the correctly classified proportions for each category.

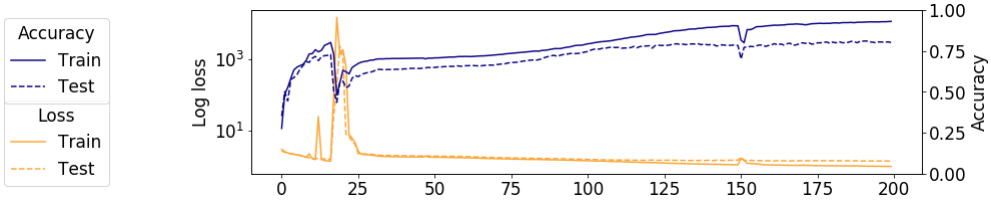

Figure 22: Accuracy and loss on train and test data sets during training of LSTM on the training data set.

5.3.3 Variant: LSTM with 2 layers - less units and regularizer

This variant uses the base configuration for activities and transitions (Listing 3) with 2 LSTM-layers and one output layer. This variant has an **accuracy of 18.6%** on the test data set. See Figure 23 for how the model performs in classifying each label. See Figure 24 for how the accuracy and loss evolved during training. Again, having 2 LSTM layers seems to be a too complex network - none of the patterns is recognized, the model always predicts the same label.

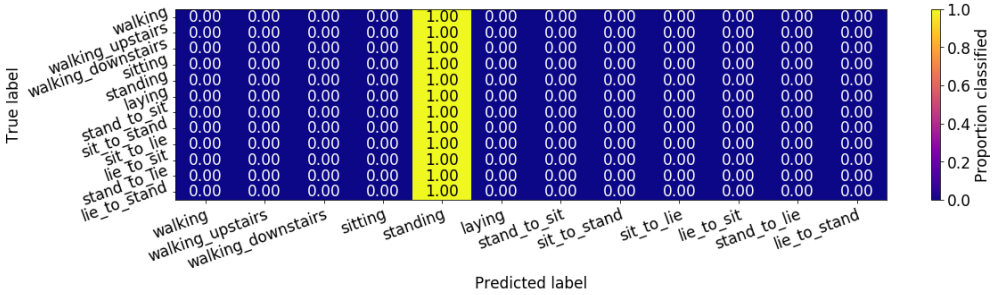

Figure 23: Confusion matrix of the predictions made by the model on the test set. The diagonal reflects the correctly classified proportions for each category.

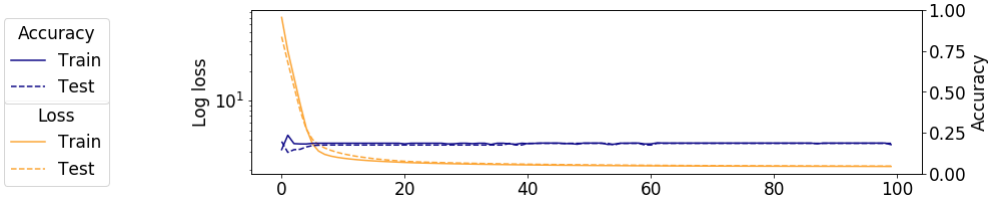

Figure 24: Accuracy and loss on train and test data sets during training of LSTM on the training data set.

5.4 Final models

The best performing models on the test data are validated on the validation set to understand their generalizability to entirely new data. Models are checked both for classifying activities only and classifying activities and transitions.

Listing 4 Configuration of best performing LSTMs on activity and transition classification.

```yaml
lstm_layer:
  units: 50
  activation: relu
  dropout: 0.0
  kernel_regularizer_l2: 0.001
  activity_regularizer_l1: 0.001
output_layer:
  activation: softmax
  kernel_regularizer_l2: 0.001
  activity_regularizer_l1: 0.001
loss: categorical_crossentropy
optimizer:
  adam:
    lr: 0.001
    beta_1: 0.9
    beta_2: 0.999
    decay: 0.0
metrics: ['accuracy']
epochs: 100
batch_size: 200
```

5.4.1 Activities only - performance on validation set

Validating the model performance of the shortened sequence (*length=64*, *stepsize=32*) variant (Section 5.2.4) on the validation set. See Listing 4 for the configuration of the LSTM. This variant has an **accuracy of 88.4%** on the validation data set. See Figure 25 for how the model performs in classifying each label. This performance reflects how well this variant will perform on new data.

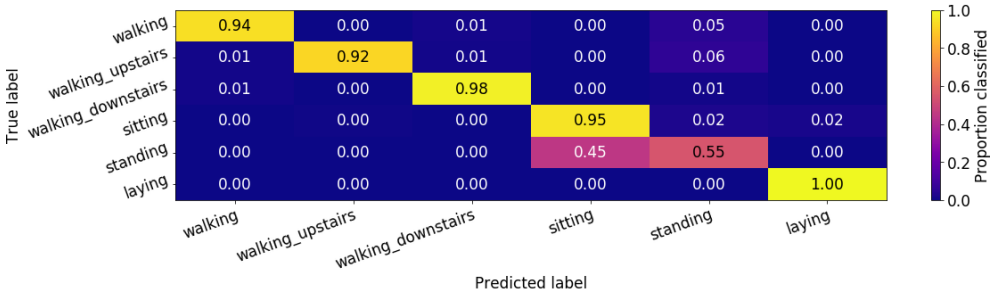

Figure 25: Confusion matrix of the predictions made by the model on the validation set. The diagonal reflects the correctly classified proportions for each category.

5.4.2 Activities and transitions - performance on validation set

Validating the model performance of the shortened sequence (*length=64*, *stepsize=32*) variant (Section 5.3.2) on the validation set. See Listing 4 for the configuration of the LSTM. This variant has an **accuracy of 81.7%** on the validation data set. See Figure 26 for how the model performs in classifying each label. This performance reflects how well this variant will perform on new data.

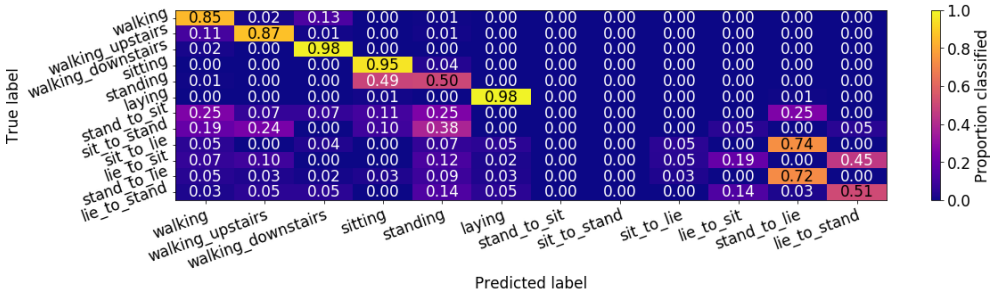

Figure 26: Confusion matrix of the predictions made by the model on the validation set. The diagonal reflects the correctly classified proportions for each category.

6 Discussion

Reviewing and describing essential machine learning concepts as well as building an exemplary machine learning pipeline was helpful in getting a more profound and comprehensive understanding of machine learning in theory and practice. Integrating the written thesis and the software code into a fully reproducible script (see Section 8.1) will hopefully aid others in understanding of how a machine learning pipeline is built. The pipeline was designed to generalize to other data sets in the HCI domain. Therefore a neural network (LSTM) was chosen for the main modeling layer as neural networks are capable of representation learning (Jesus Rubio, Angelov, & Pacheco, 2015) and can therefore be used on other data sets without the need for intensive manual feature engineering. However, finding a good configuration for the LSTM layer was very time consuming as LSTMs can be implemented in various architectures and many hyper-parameters combinations can be tested. Therefore, exploring other neural networks or combinations of neural networks lies outside the scope of this master's thesis.

The best performing variant of the machine learning pipeline for human activity recognition achieves an accuracy of almost 90% on new data when only activities are included. Including transitions this accuracy drops to about 80% with some of the transitions not being recognized at all. The entire pipeline contains only little pre-processing and no manual feature engineering except for the separation of body and gravity components in the acceleration data. Because of this the pipeline can potentially be fit to other HCI time series data sets with only adapting the configuration file of the LSTM and taking out some of the layers.

LSTMs are stochastic models and the fit will be different every time they are trained - even on the same data. To make the results reproducible seeds where used, which guarantee the same results when retraining. However, using

different seeds would lead to different results. Therefore, the presented accuracy measures of the models might be biased and not reflect the average performance of this model. To make the performance measure and the classifications more robust even when retraining the model, an ensemble of models can be built. The same configuration would get trained n times and classifications would be mixed from all n models.

A potential improvement to the modeling block of the pipeline might be to add a convolutional neural network (CNN) before the LSTM (Sainath, Vinyals, Senior, & Sak, 2015). CNNs are good at extracting spatio-temporal features (Yang, Nguyen, San, Li, & Krishnaswamy, 2015), which can then be fed to the LSTM, which might improve the performance of the LSTM.

Identifying a well fitting model architecture and hyper-parameters configuration was done manually for the presented pipeline. However, this process is labor intensive (far more than a 100 different architecture-configuration have been manually tested) or requires enough experience to know which variations are worth testing. Alternatively, a gridsearch[14] can be conducted on a range of hyperparameter. Here, the computer automatically checks many different hyperparameter configurations. The neural network intelligence (NNI) toolkit[15] by Microsoft even automatically tests different neural network architectures. However, both the gridsearch and NNI toolkit require substantial computational resource as a lot of models will be trained and tested during these processes.

[14]https://scikit-learn.org/stable/modules/grid_search.html (2019-05-07)
[15]https://github.com/Microsoft/nni (2019-05-07)

7 References

Arshavsky, Y. I., Gelfand, I. M., & Orlovsky, G. N. (1983). The cerebellum and control of rhythmical movements. *Trends in Neurosciences*, *6*, 417–422. https://doi.org/10.1016/0166-2236(83)90191-1

Baars, B. J. (2002). The conscious access hypothesis: Origins and recent evidence. *Trends in Cognitive Sciences*, *6*(1), 47–52.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Bolt, J., & Van Zanden, J. L. (2014). The m addison p roject: Collaborative research on historical national accounts. *The Economic History Review*, *67*(3), 627–651.

Bostrom, N. (1998). *How long before superintelligence?*

Bourguinon, F., & Morrison, C. (2009). *Inequality among world citizens: 1820-1992. The world bank*.

Boybat, I., Gallo, M. L., Nandakumar, S. R., Moraitis, T., Parnell, T., Tuma, T., ... Eleftheriou, E. (2018). Neuromorphic computing with multi-memristive synapses. *Nature Communications*, *9*(1), 2514. https://doi.org/10.1038/s41467-018-04933-y

Burns, W. E. (2001). *The scientific revolution: An encyclopedia*. ABC-CLIO.

Chalmers, D. J. (1995). Facing up to the problem of consciousness. *Journal of Consciousness Studies*, *2*(3), 200–219.

Conigliaro, T. S., & Coniliaro, S. J. (1989). *U.S. patent No. 4849733A*. Retrieved from https://patents.google.com/patent/US4849733A/en

Cook, M. J., Cranmer, C., Finan, R., Sapeluk, A., & Milton, C.-A. (2017). 15 memory load and task interference: Hidden usability issues in speech interfaces. *Engineering Psychology and Cognitive Ergonomics: Volume 1: Transportation Systems*.

Crick, F., & Koch, C. (2003). A framework for consciousness. *Nature*

Neuroscience, *6*(2), 119.

Deary, I. J. (2001). *Intelligence: A very short introduction*. OUP Oxford.

Dehaene, S., Changeux, J.-P., Naccache, L., Sackur, J., & Sergent, C. (2006). Conscious, preconscious, and subliminal processing: A testable taxonomy. *Trends in Cognitive Sciences*, *10*(5), 204–211.

Di Perri, C., Annen, J., Antonopoulos, G., Amico, E., Cavaliere, C., & Laureys, S. (2016). Measuring consciousness through imaging. In *Brain function and responsiveness in disorders of consciousness* (pp. 51–65). Springer.

Dixit, A., Lannoo, B., Colle, D., Pickavet, M., & Demeester, P. (2015). Energy efficient dynamic bandwidth allocation for ethernet passive optical networks: Overview, challenges, and solutions. *Optical Switching and Networking*, *18*, 169–179.

Domingos, P. M. (2012). A few useful things to know about machine learning. *Commun. Acm*, *55*(10), 78–87.

Farthing, G. W. (1992). *The psychology of consciousness*. Prentice-Hall, Inc.

Feenstra, R. C., Inklaar, R., & Timmer, M. P. (2015). The next generation of the penn world table. *American Economic Review*, *105*(10), 3150–3182.

Finkelstein, D. H. (1999). On the distinction between conscious and unconscious states of mind. *American Philosophical Quarterly*, *36*(2), 79–100. Retrieved from https://www.jstor.org/stable/20009956

Friedman, J., Hastie, T., & Tibshirani, R. (2001). *The elements of statistical learning* (Vol. 1). Springer series in statistics New York.

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999). *Learning to forget: Continual prediction with LSTM*.

Granger, R. H., & Hearn, R. A. (2007). Models of thalamocortical system. *Scholarpedia*, *2*(11), 1796.

Gray, J. R., & Thompson, P. M. (2004). Neurobiology of intelligence:

Science and ethics. *Nature Reviews Neuroscience*, *5*(6), 471.

Hawkins, J., & Blakeslee, S. (2007). *On intelligence*. Macmillan.

Hees, V. T. van, Gorzelniak, L., Dean León, E. C., Eder, M., Pias, M., Taherian, S., ... Brage, S. (2013). Separating movement and gravity components in an acceleration signal and implications for the assessment of human daily physical activity. *PLoS ONE*, *8*(4). https://doi.org/10.1371/journal.pone.0061691

Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, *313*(5786), 504–507.

Hollnagel, E., & Woods, D. D. (2005). *Joint cognitive systems: Foundations of cognitive systems engineering*. CRC Press.

IEA. (2019). Definition and domains of ergonomics IEA website. Retrieved March 28, 2019, from https://www.iea.cc/whats/index.html

Jacob, R. J. K., & Karn, K. S. (2003). Commentary on section 4 - eye tracking in human-computer interaction and usability research: Ready to deliver the promises. In J. Hyönä, R. Radach, & H. Deubel (Eds.), *The mind's eye* (pp. 573–605). https://doi.org/10.1016/B978-044451020-4/50031-1

Jesus Rubio, J. de, Angelov, P., & Pacheco, J. (2015). Ainsworth, s.(2006). DeFT: A conceptual framework for considering learning with multiple representations. Learning and instruction, 16 (3), 183-198. Bengio, y., courville, a., and vincent, p.(2013). Representation learning: A review and new perspectives. Pattern analysis and machine intelligence, IEEE transactions on, 35 (8), 1798-1828. *International conference on machine learning*, *105*, 112.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv:1412.6980 [Cs]*. Retrieved from http://arxiv.org/abs/1412.6980

Koch, C. (2004). *The quest for consciousness: A neurobiological approach*. Roberts; Company Englewood, CO.

Koza, J. R., Bennett, F. H., Andre, D., & Keane, M. A. (1996). Automated design of both the topology and sizing of analog electrical circuits using genetic

programming. In *Artificial intelligence in design'96* (pp. 151–170). Springer.

Legg, S., & Hutter, M. (2007). Universal intelligence: A definition of machine intelligence. *Minds and Machines*, *17*(4), 391–444.

Lutz, W., Butz, W. P., & Samir, K. ed. (2017). *World population & human capital in the twenty-first century: An overview*. Oxford University Press.

Mackintosh, N., & Mackintosh, N. J. (2011). *IQ and human intelligence*. Oxford University Press.

Mukhopadhyay, S. C. (2015). Wearable sensors for human activity monitoring: A review. *IEEE Sensors Journal*, *15*(3), 1321–1330. https://doi.org/10.1109/JSEN.2014.2370945

Müller, K.-R., Tangermann, M., Dornhege, G., Krauledat, M., Curio, G., & Blankertz, B. (2008). Machine learning for real-time single-trial EEG-analysis: From brain–computer interfacing to mental state monitoring. *Journal of Neuroscience Methods*, *167*(1), 82–90.

Noble, D. (2017). *Forces of production: A social history of industrial automation*. Routledge.

Novack, G. E. (1965). *The origins of materialism*.

Parvizi, J., & Damasio, A. (2001). Consciousness and the brainstem. *Cognition*, *79*(1), 135–160.

Peltzman, S. (2009). Mortality inequality. *Journal of Economic Perspectives*, *23*(4), 175–190. https://doi.org/10.1257/jep.23.4.175

Proctor, R. W., & Van Zandt, T. (2018). *Human factors in simple and complex systems*. CRC press.

Rani, P., Liu, C., Sarkar, N., & Vanman, E. (2006). An empirical study of machine learning techniques for affect recognition in human–robot interaction. *Pattern Analysis and Applications*, *9*(1), 58–69.

Rautaray, S. S., & Agrawal, A. (2015). Vision based hand gesture recognition for human computer interaction: A survey. *Artificial Intelligence Review*,

43(1), 1–54.

Reader, S. M., Hager, Y., & Laland, K. N. (2011). The evolution of primate general and cultural intelligence. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, *366*(1567), 1017–1027.

Reyes-Ortiz, J.-L., Oneto, L., Samà, A., Parra, X., & Anguita, D. (2016). Transition-aware human activity recognition using smartphones. *Neurocomputing*, *171*, 754–767.

Rogers Jr, H. (1987). *Theory of recursive functions and effective computability, paperback edition*. Cambridge: MIT Press.

Russel, S., & Norvig, P. (2010). Artificial intelligence: A modern approach thrid edition. *Person Education, Boston Munich*.

Sainath, T. N., Vinyals, O., Senior, A., & Sak, H. (2015). Convolutional, long short-term memory, fully connected deep neural networks. *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, 4580–4584. https://doi.org/10.1109/ICASSP.2015.7178838

Seth, A. K., Izhikevich, E., Reeke, G. N., & Edelman, G. M. (2006). Theories and measures of consciousness: An extended framework. *Proceedings of the National Academy of Sciences*, *103*(28), 10799–10804.

Sheridan, T. B. (2016). Human–robot interaction: Status and challenges. *Human Factors*, *58*(4), 525–532.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... Graepel, T. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, *362*(6419), 1140–1144.

Simon, H. A., & Newell, A. (2017). Human problem solving: The state of the theory in 1970 1. In *Structural learning (volume 2)* (pp. 131–151). Routledge.

Suchman, L. A. (1987). *Plans and situated actions: The problem of human-machine communication*. Cambridge university press.

Sutton, R. S., & Barto, A. G. (1998). *Introduction to reinforcement learning* (Vol. 135). MIT press Cambridge.

Tan, D., & Nijholt, A. (2010). Brain-computer interfaces and human-computer interaction. In D. S. Tan & A. Nijholt (Eds.), *Brain-computer interfaces: Applying our minds to human-computer interaction* (pp. 3–19). https://doi.org/10.1007/978-1-84996-272-8_1

Tononi, G. (2004). An information integration theory of consciousness. *BMC Neuroscience*, *5*(1), 42.

Tononi, G., Boly, M., Massimini, M., & Koch, C. (2016). Integrated information theory: From consciousness to its physical substrate. *Nature Reviews Neuroscience*, *17*(7), 450.

Tononi, G., & Edelman, G. M. (1998). Consciousness and complexity. *Science*, *282*(5395), 1846–1851. https://doi.org/10.1126/science.282.5395.1846

Tufte, E. R., Goeler, N. H., & Benson, R. (1990). *Envisioning information* (Vol. 126). Graphics press Cheshire, CT.

Veltink, P. H., Bussmann, H. J., De Vries, W., Martens, W. J., & Van Lummel, R. C. (1996). Detection of static and dynamic activities using uniaxial accelerometers. *IEEE Transactions on Rehabilitation Engineering*, *4*(4), 375–385.

Verleysen, M., & François, D. (2005). The curse of dimensionality in data mining and time series prediction. In J. Cabestany, A. Prieto, & F. Sandoval (Eds.), *Computational intelligence and bioinspired systems* (pp. 758–770). Springer Berlin Heidelberg.

Von Neumann, J. (1945). First draft of a report on the EDVAC, 30 june 1945. *Contract No W-670-ORD-492, Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. Google Scholar.*

Wald, A. (1950). *Statistical decision functions.*

Yang, J., Nguyen, M. N., San, P. P., Li, X. L., & Krishnaswamy, S. (2015).

Deep convolutional neural networks on multichannel time series for human activity recognition. *Twenty-fourth international joint conference on artificial intelligence.*

8 Appendix

8.1 A - repository reproducible master's thesis

This folder on the digital storage contains the entire repository from which the master thesis was produced. This includes the markdown files, python code files, figures, documentation and the bash scripts for automating the whole process. All files in this repository have been produced by the author of this thesis unless stated otherwise. Also the digital version of the master's thesis is included in this repository. The folder *sections_in_progress* holds the files that needed code execution. These files also contain all the code that was written for the pipeline. Once these sections are executed the output of that process is stored in *sections_executed*. Also, some sections like the title, abstract or appendix are directly edited in *sections_executed* as these sections to not contain any code and therefore to not require execution. See Section 8.2.2.5 in the appendix for how exactly files are executed.

8.2 B - reproducing the thesis

8.2.1 Google Colaboratory - no local setup required (recommended)

Google provides online resources for interactively running python code in jupyter notebooks. They also provide GPU processing units, which allow for high processing speeds. A demo of Google Colaboratory can be found here[16]. The following steps describe how to run parts of this thesis on Google Colaboratory without the need for setting up a local machine.

- Sign in to google.com (create a free account if needed)
- Go to drive.google.com -> Click on 'New' -> 'More'

[16] https://colab.research.google.com (2019-06-03)

- Check if 'Colaboratory' is in the list, if not click on 'Connect more apps' and add 'Colaboratory'
- Upload the .ipynb file that should be reproduced from the 'accompanying_digital_storage:Appendix A/sections_executed' directory to google drive
- just double-clicking the uploaded ipynb file on google drive should open it in Google Colaboratory
- Go to "Runtime" (top menu bar) -> "Change runtime type" -> Set "Hardware accelerator" to "GPU" and "Save"
- Open the sidebar by clicking on the arrow on the left -> Go to "Files" and "Upload"
- Upload all files from the 'accompanying_digital_storage:Appendix A/code' directory. These are the custom python modules and necessay for execution of the code.
- Click on the first cell and start executing cells one by one by hitting "Shift+Enter".
- Note: some figures might not be presented correctly for two reasons: 1) the figures are not uploaded, 2) the code produces only references to figures, and the figures are only integrated in a later processing step, which is not performed here

8.2.2 Locally

In this section the steps to reproduce the entire master thesis are explained. The intructions are written for Linux and OSX machines using the default terminal commands. Although the same packages might be available for Windows the presented commands will need to be adapted.

8.2.2.1 Base setup It is helpful to use virtual environments to keep the packages of a machine learning project separate from the operating system. The conda package manager offers virtual environments as well as python package distributions for machine learning and will be used here. Alternatively the pip package manager and python's virtual environments can be used.

Get and install miniconda (Python 3) for your system[17]. Once miniconda is installed open a terminal and install the jupyter package with: `conda install jupyter`. This will install the jupyter package in the base environment. This is needed so we can later register other conda environments as so called ipython kernels, which makes them available to other tools that run independent of the conda environments like the nbconvert command which is used later to run the files.

8.2.2.2 Setup environment for executing code with GPU and CPU This section contains an examplary setup process of GPU drivers, CUDA and cudnn on an ubuntu machine. The process likely looks different for other systems.

System specifications:

- Ubuntu 18.04
- Kernel: 4.15.0
- GPU: (Nvidia) Asus GeForce GTX 970, 4GB
- CPU: AMD FX 3850

8.2.2.2.1 Installing GPU drivers The drivers can also be installed during installation of the CUDA toolkit, however this did not work for me as the nouveau drivers could not be deactivate during that process. So I installed the GPU drivers first manually and the toolkit afterwards.

[17] https://conda.io/en/latest/miniconda.html

```
sudo apt install openjdk-8-jdk


sudo add-apt-repository ppa:graphics-drivers/ppa
sudo apt update
sudo apt upgrade

# remove everything nvidia related
sudo apt purge nvidia*

#install new drivers
ubuntu-drivers devices
sudo ubuntu-drivers autoinstall

# instead of the autoinstall one can also do for example
sudo apt-get install nvidia-390

sudo reboot
```

- check if drivers are properly installed with: `nvidia-smi`
- check that **no** nouveau driver is loaded with (this should not give any output): `lsmod | grep nouveau`

8.2.2.2.2 Install CUDA toolkit

- check which gcc/g++ version and CUDA version are needed for your kernel: https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#system-requirements
- install gcc/g++ if needed, check version with `gcc --version` and `g++ --version`

- follow installation guide: https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#introduction
- don't install graphic drivers during setup, only toolkit!

```
sudo sh cuda_XXX_linux.run --toolkit
sudo reboot
```

8.2.2.2.3 Install cudnn

- Follow: https://docs.nvidia.com/deeplearning/sdk/cudnn-install/index.html

8.2.2.2.4 Set up conda environment

- Create, register (needed for later) and activate environement:

```
conda activate base
conda create -n kirsch_gpu anaconda tensorflow-gpu keras
ipython kernel install --user --name=kirsch_gpu
conda activate kirsch_gpu
```

- Then check in python if tensorflow recognizes the GPU. The output should not be an empty string, but something with '/device':

```
import tensorflow as tf
tf.test.gpu_device_name()
```

8.2.2.3 Setup environment for executing code with CPU only This requires no further steps, but to set up a conda environment with the correct packages.

```
conda activate base
conda create -n kirsch anaconda tensorflow keras
ipython kernel install --user --name=kirsch
conda activate kirsch
```

8.2.2.4 Setup environment for automating thesis production To automate the execution of the machine learning pipeline and creation of a PDF output the following packages need to be installed.

```
# A system wide tex distribution
sudo apt-get install texlive


# An environment for pandoc
conda create -n kirsch_pandoc
conda activate kirsch_pandoc
conda install jupyter
conda install -c conda-forge pandoc=2.7
conda install -c conda-forge pandoc-crossref
```

8.2.2.5 Execution Copy the entire `accompanying_digital_storage:/Appendix A` directory to the machine local drive. Change directory to the `Appendix A` folder. Everything will be executed from here. All following paths are relative to this location. The scripts in the `scripts/` directory are used to wrap some commands for running the code and transforming the output into a beautiful PDF. Each `*.md` file in the `sections_in_progress` directory needs to be executed separately to produce the final PDF. The order of which these files are executed is not important.

```
# Activate the pandoc conda environment first
conda activate kirsch_pandoc
```

8.2.2.5.1 Conversion from markdown (md) to juypter notebooks (ipynb) The files in `sections_in_progress` are markdown files with python code snippets. Lines like `::: {.cell .markdown}` denote what kind of cell the following part in the file will become in the jupyter

notebook (markdown or code cell). The conversion is done by the `scripts/convert_md_to_ipynb.py` python script with the bash wrapper script `scripts/b1_convert_md_to_ipynb` on top. This script expects both the filename and the jupyter kernel / conda environment. Run it likes this:

```
./scripts/b1_convert_md_to_ipynb sections_in_progress/ \
    100_working_002_intro_theory.md \
    kirsch_gpu
```

8.2.2.5.2 Execution of the ipynb file Next, the generated ipynb file will run headlessly using the `scripts/b2_exec_ipynb_headless` script. This spins up a notebook server and, executes the notebook and stops the server again using `nbconvert`. The executed notebook file is stored `sections_executed`. This script expects only the name of the file. Run it like this:

```
./scripts/b2_exec_ipynb_headless \
  sections_in_progress/100_working_002_intro_theory.md
```

8.2.2.5.3 Conversion of executed ipynb back to markdown The executed ipynb is then converted back to markdown using nbconvert again with some manual fixes of the output. The markdown output file is stored in `sections_executed`. Run it like this:

```
./scripts/b3_convert_body_iipynb-execd_to_md \
  sections_in_progress/100_working_002_intro_theory.md
```

8.2.2.5.4 Concatenation and formatting Finally, all the markdown files in `sections_executed` are concatenated and then transformed using `pandoc` and `latex` engines to a properly formatted PDF file. The title, author, table of contents and the bibliography are also added in this step. Run it like this:

```
./scripts/c0_make_pdf
```

8.2.2.5.5 Wrapper The scripts b1, b2, b3 and c0 and are wrapped by `scripts/a0_run_all`, which can be run like this:

```
./scripts/a0_run_all \
  sections_in_progress/100_working_002_intro_theory.md \
  kirsch_cpu
```

8.3 C - Running unit tests

To run the unit tests activate the conda environment containing the anaconda distribution (this should be either *kirsch* or *kirsch_gpu* following the instructions in Section 8.2.2) with `conda activate kirsch_gpu` in a terminal. Change directory to *accompanying_digital_storage:/Appendix A/code* and run `pytest` in the terminal. This will automatically execute all functions starting with 'test' in all python files which names start with 'test'.

8.4 D - Machine learning cheat sheets

See Figure 27 and Figure 28 for an overview of algorithm groups and how to choose algorithms based on requirements. More cheat sheets can be found at `accompanying_digital_storage:/Appendix /C`.

Figure 27: Overview of different machine learning approaches and algorithms. Downloaded from https://becominghuman.ai/cheat-sheets-for-ai-neural-networks-machine-learning-deep-learning-big-data-science-pdf-f22dc900d2d7 on 2019-05-31.

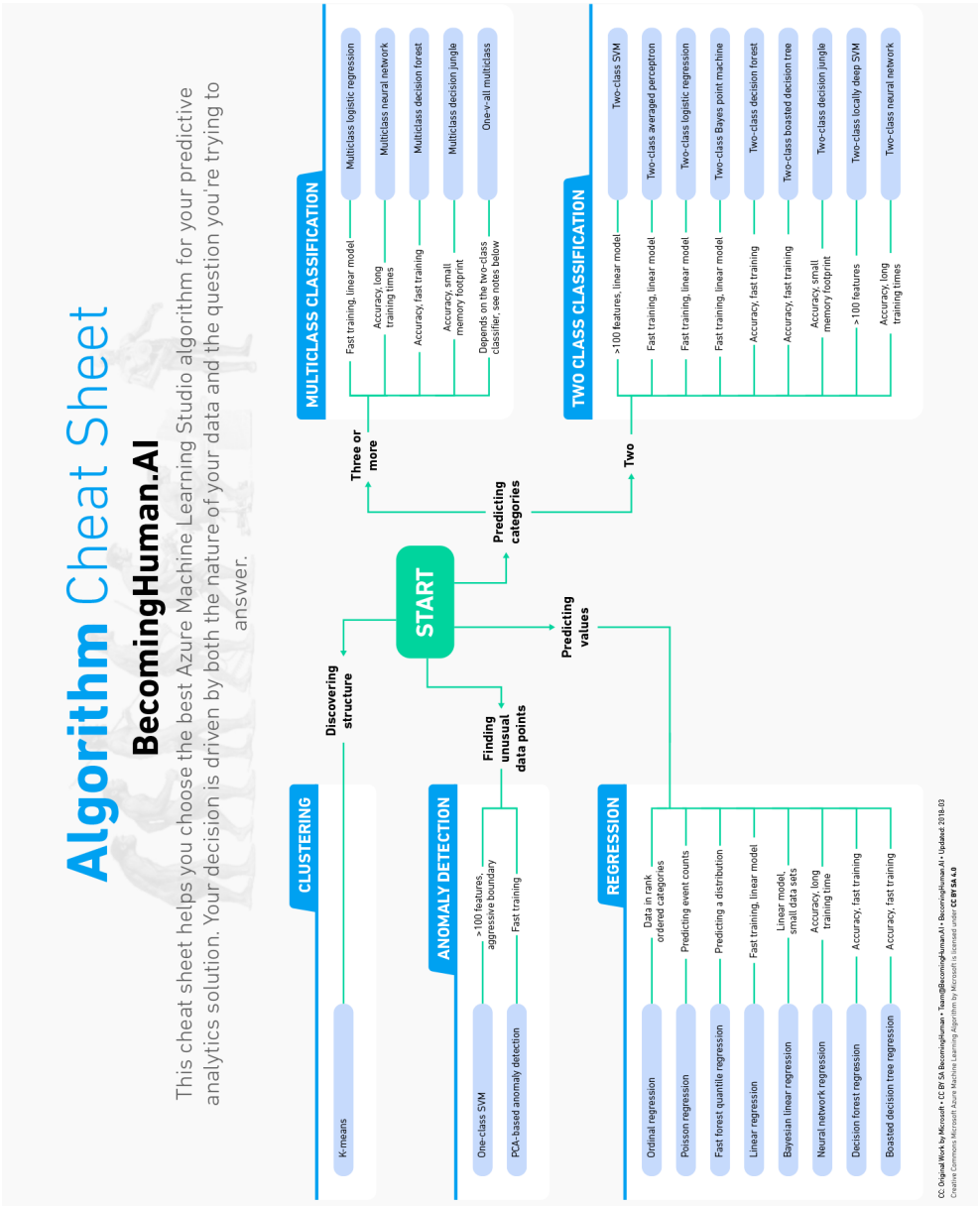

Figure 28: Map to find the best algorithm. Adapted from Microsoft Azure Machine Learning Map. Downloaded from https://becominghuman.ai/cheat-sheets-for-ai-neural-networks-machine-learning-deep-learning-big-data-science-pdf-f22dc900d2d7 on 2019-05-31.