Presence Detection in Homes using Aggregated Sensor Data

Melker Georgson

Master's thesis
2024:E9

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

CENTRUM SCIENTIARUM MATHEMATICARUM

Abstract

This thesis presents the development of a binary classifier, which classifies homes as occupied or not. This was done using two different types of neural networks; standard feedforward networks and LSTM-networks. The input to these networks was sensor data collected from devices created by Minut AB. Data from previously used automatic alarm feature was used as ground truth. A major part of the project consisted of preparing and filtering the ground truth and input data. Once this was done a set of suitable hyperparameters was found by tuning the hyperparameters one by one with the other ones fixed. In general the tuning of the hyperparameters was too noisy to make any certain conclusions about which values were optimal.

The classifiers succeeded in identifying some patterns indicative of home occupancy, outperforming a baseline model, which randomly guesses occupancy status. Despite this, the classifiers' performance did not yield the high accuracy required for their intended applications in heating system regulation and other home automation tasks. The feedforward network model got the best results, but LSTM-networks could potentially be equally good for this task, since the LSTM-networks were trained on smaller amounts of data and data quality appeared to affect the result more than model choice. Areas to improve include preprocessing, the method for choosing hyperparameters and quality of the ground truth data.

Contents

1 Introduction and Purpose

Minut AB is a company which produces home monitor sensors. The monitoring is done in a manner which gives the owner of the home or the people renting it as much privacy as possible. A typical customer is an owner of a property being rented out at a short term rental site. One feature, which is of interest, is human presence detection. Knowing whether a space is occupied could aid in, for instance, regulating heating and usage of electricity. The aim of this project is to identify relevant data for a human presence detection algorithm and devise such an algorithm.

In more mathematical terms, the project is a binary classification problem of points in a multivariate time series. That is, given historical and current sensor data, the objective is to determine whether anyone is present (denoted as a 1) or whether the home is empty (denoted as 0). The detection algorithm should operate in semi-real time. That is, it should be able to classify a time point shortly after the input data from that point has been collected. Thus methods using metrics from whole series for classifying the beginning of the series were discarded.

2 Background

2.1 Presence detection

Human presence detection is a task for which various solutions have been proposed. Some of the more common solutions are passive infrared (PIR) sensors (often used for automatically turning on the light in a room) and scalar infrared range-finders (used to keep elevator doors open if someone is about to enter or exit) [1]. Some of the main issues with these techniques, in terms of presence detection in a "home", are that they only respond to activity or presence in a limited area and that they are unable to detect presence without recent movement or activity in the related area. In some applications, larger networks of sensors, such as the above mentioned examples or pressure tiles in the floor, are used. This does, however, mean that many devices need to be installed, which would most likely be both too cumbersome and expensive for the application areas related to Minut AB.

Another group of approaches is methods utilizing collaboration with the people expected to be present in the building of interest. An example of this is bluetooth low energy (BLE) technology [2], which uses signals sent out from, for instance, a tag or mobile app. This requires that all possible occupants of a building install the required app or carry a tag around at all times. For short term rentals this solution seems too dubious. Another collaborative solution is usage of positioning data such as GPS signals. One such solution is geofencing [3]. In short, this uses the position of mobile devices to determine whether the device of interest is located within a certain distance from a building. To collect this data, the users of the property would have to install an app on their mobile devices. As described later in this report, geofencing data has some additional problems such as inconsistency between different mobile devices and inability to register changes in presence status when someone arrives or leaves without the mobile device.

2.2 A brief history of machine learning

The concept of machine learning stems from the 50s with one important event being when Rosenblatt proposed the standard perceptron, which is the fundamental building block in many machine learning models, in 1958 [4]. Despite some more work on the topic during the 80s with developments within, for instance, decision trees and linear regression, funding for machine learning research was decreased during the late 80s and early 90s. This happened partly due to a lack of computational power and a general opinion among decision makers that machine learning had been overhyped. However, in the late 90s and during the 2000s, increased computational power and larger amounts of available data (following the increased usage of the internet) allowed for more complex models to be trained, which lead to a rapid development in deep learning and the development of long short-term memory (LSTM) cells [5]. These have come to play a vital role in the now very developed area of natural language processing (NLP). Later this area was developed further through increased computational power and the introduction of transformer models [6], which were crucial in the development of large language models, such as chatGPT [7]. With the introduction of convolutional neural networks (CNNs) [8] in 2012, the set of problems, which could be tackled by deep learning, increased further to include areas such as image analysis. With this recent development of more sophisticated models and an increasing amount of available data within many areas, machine learning will most likely play an increasingly bigger role in the future.

3 Theory and Definitions

3.1 Neural Networks

Neural networks are used to approximate a function $\hat{f}$. This function could for instance be a mapping from a set of input variables x to a category or class, $\hat{y}$. In the context of this report, x is a vector of recorded sensor values and $\hat{y}$ is a 0 or 1 depending on whether the data point belongs to a populated or empty home. The goal of the neural network is to tune a set of parameters θ for a mapping $y = f(x; \theta)$ such that y and $\hat{y}$ are as similar as possible.

3.1.1 Standard Perceptrons

The simplest type of neural network is a single standard perceptron, which works in the following way:

Each element of the input vector x_i to the perceptron is multiplied by a weight. These weights represent the importance of the respective inputs. Mathematically, this is represented as the product of each input x_i and its corresponding weight w_i. The weighted inputs are summed up. This summation process represents the combined effect of all inputs and their respective weights. The sum z is calculated as $z = \sum_i x_i \cdot w_i + b$, where b is a bias term.

The summed value z is then passed through an activation function, ϕ. This function often introduces non-linearity to the output, but could also be linear. Common activation functions include the step function, sigmoid function, hyperbolic tangent (tanh), or Rectified Linear Unit (ReLU) [9].

The output of the activation function is the final output of the perceptron. For instance, if using a step function as activation function, the output could be binary (0 or 1) depending on whether the computed sum is above or below a certain threshold. In Fig. 1 a schematic overview of a single perceptron is presented. In this image a step function is used as activation function, but this could be replaced by any other function.

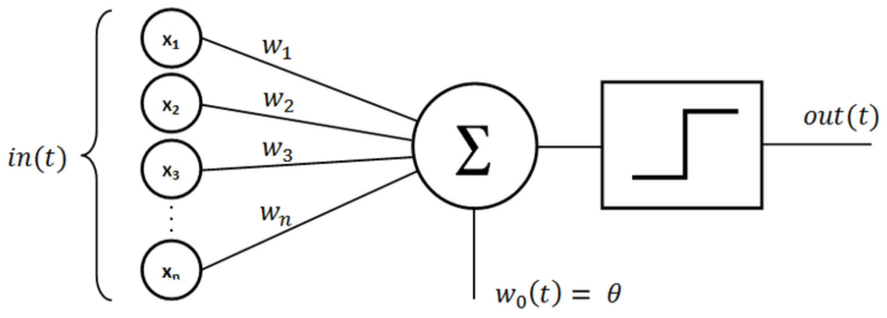

Figure 1: A schematic image of a single perceptron with $\theta(z)$ as activation function [10].

A learning algorithm adjusts the weights and biases based on a function of the difference between the perceptron outputs and the ground truth, $\hat{y}$. This function is called a *loss function*. This process continues iteratively to minimize the loss. The loss minimized is the value of the loss function for a certain portion of the data, called *training data*. The weights are iteratively tuned to yield the lowest possible loss value for this data. The weights are tuned once per *batch* of data. The data is split into batches, where each batch contains a set number of points, to speed up the training and to lower the demands on the computer performing the training. After all batches have been processed once, an *epoch* has passed.

In order to check whether the weights, obtained during and after the training, are suitable for new data, predictions are made a portion of the not yet used data, called *validation data*. These predictions are made during the training process to monitor the generalization performance of the trained model. Once the model with the best validation performance has been found, a final test of the model is made by letting it make predictions on a separate data set called *test data*. No tweaks of the model should be done based on the result of this test. Instead it should serve as an estimate of the generalization capabilities of the final model.

3.1.2 Feedforward Neural Networks

A natural continuation of the standard perceptron is the usage of multiple perceptrons distributed into multiple different layers. These feed-forward neural networks, often referred to as multilayer perceptrons (MLPs), consist of an input layer, one or more hidden layers, and an output layer. For the feedforward neural networks used in this project, each neuron (a perceptron used in combination with other perceptrons in a neural network) in one layer is connected to every neuron in the subsequent layer, and information flows only forward from the input nodes through the hidden nodes to the output nodes. What follows is a short explanation of the different layers.

The input layer receives the initial data. Each neuron in the input layer represents a feature or input. Each node in the input layer works as a standard perceptron. The output from a certain node in the input layer is then fed into the weighted sum of perceptrons located in a hidden layer. The hidden layer works identically to the input layer except for the fact that they process weighted sums of outputs from the input layer and not a weighted sum of the features fed into the network. A network could contain several hidden layers. The output of the final hidden layer is then sent to an output layer, which in this case is a single perceptron with one output value, since the objective of the network is to output a single boolean indicating whether the home is occupied at a certain time point. A general outline of how a neural network can be visualized is presented in Fig.2.

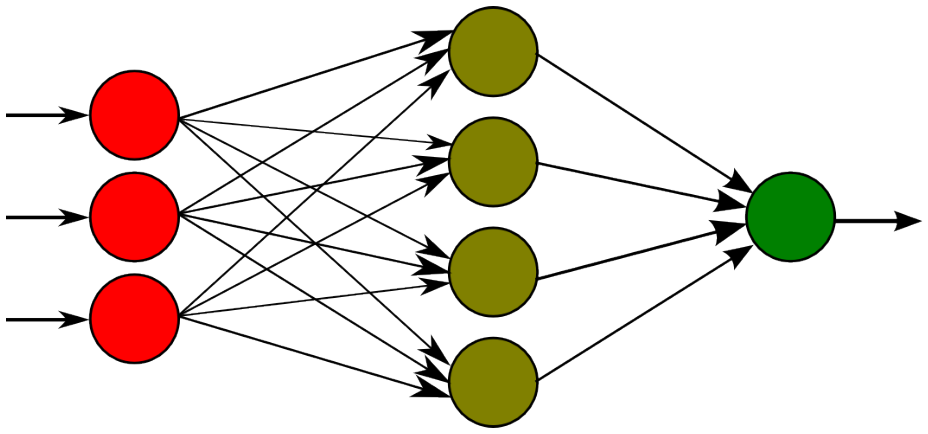

Figure 2: A small feedforward neural network. The nodes in the leftmost layer contain one input value each. The middle layer is a hidden layer. There can be more than one hidden layer. The green node to the right is the output node [11].

3.1.3 Hyperparameters

Hyperparameters are parameters which are not optimized during the training process, but instead chosen prior to the training. Examples of hyperparameters are *learning rate* (regulating how much the weights are updated) and batch size. An important part of machine learning is finding suitable values of these hyperparameters.

3.1.4 Activation functions

As stated in Sec. 3.1.1, activation functions often introduce non-linearities in the nodes in order to model more complex patterns than a method only using linearly weighted sums. In this section, the activation functions used in the project will be defined. ReLU is a commonly used activation function in a wide range of applications and is defined by Eq. (1)

$$f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}. \tag{1}$$

Calculating the gradient of this function is easy and thus the computational complexity is low, which leads to fast trainings. Another property of ReLu is that the gradient never gets small for positive input sums, which could combat the vanishing gradient problem for deep networks [9]. However, this property could be a problem, since the unbounded output could lead to instability. Another potential drawback of ReLu is that it does not make use of negative values. Therefore, the inputs to a node with ReLu might become irrelevant when updating the weights during training [9].

An activation function often used for the output layer of binary classifier is the sigmoid function defined by Eq. (2). Two reasons why it is often used in this context are that it yields an output in the range $[0, 1]$ and has a steeper gradient around 0.5. The output could be interpreted as the estimated probability of the sample belonging to class 1. The function has good properties for introducing non-linearity. A drawback is the fact that gradients for output values close to 0 or 1 get very small, which can result in too small updates of weights connected to the neuron with a sigmoid activation function. Another issue is that the output is not centered around zero, which can lead to slower convergence [9].

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

Finally the tanh activation function is defined by Eq. (3)

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{1 - e^{-2z}}{1 + e^{-2z}}. \tag{3}$$

This function returns a value between -1 and 1. The tanh function could be seen as a translated and stretched version of the sigmoid function with a steeper but narrower peak in the first order derivative function. It is also symmetric around 0, which often leads to quicker convergence. However, tanh suffers from the same problem as the sigmoid function in terms of gradients getting close to 0 when the input to the function has a large absolute value [9].

3.1.5 LSTM-cells

Recurrent neural networks (RNNs) is a class of networks often used for sequential data. The basic idea behind a recurrent neural network is that information stored in a cell (a more complex neuron) while processing data point n in a sequence is used by that cell to process data point $n + 1$. LSTM-networks, a type of RNN, are designed to process sequences of data and are (unlike standard RNNs) capable of learning long-term dependencies. A cell holds a state, C_t, which is used as input for data sample t+1 as shown in Fig. 3. The cell also uses its previous output h_{t-1} as input. The last input to the cell is the feature vector for data sample t [5].

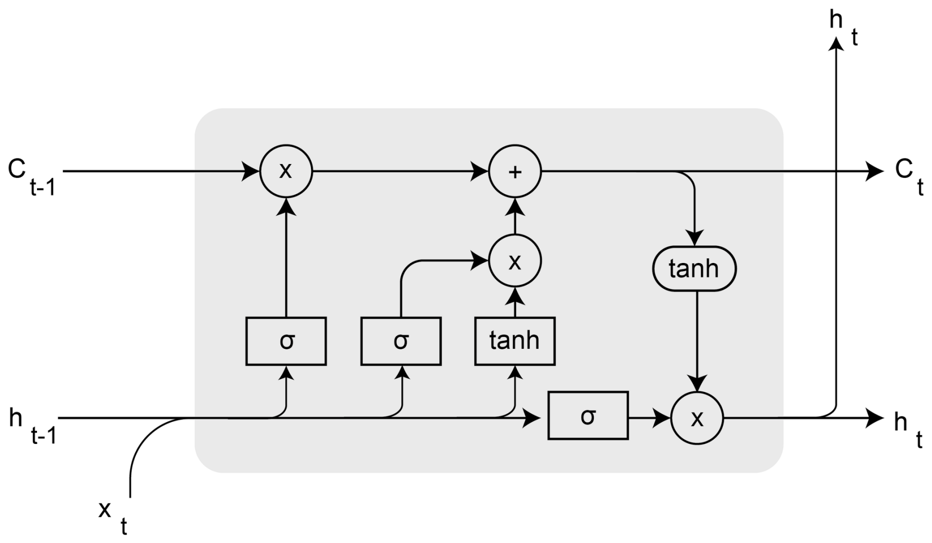

Figure 3: Overview of a typical LSTM-cell.

The LSTM-cell processes the input x and generates an output h using the following equations:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{4}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{5}$$
$$\hat{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{6}$$
$$C_t = f_t * C_{t-1} + i_t * \hat{C}_t \tag{7}$$
$$o_t = \sigma(W_o[h_{t-1}] + b_o) \tag{8}$$
$$h_t = o_t * \tanh(C_t) \tag{9}$$

where W_f,W_i and W_o are trainable weights and b_f, b_i, b_C and b_o are biases. Thus, a network of LSTM-cells has more trainable parameters than a standard feedforward network of the same size, since the LSTM-network also needs to tune the above mentioned weights [5].

3.1.6 Stateless and stateful LSTM-cells

An LSTM-cell can use the cell state C_t and hidden state h_t in two different ways. The first is to reset the states after each batch of training data. Thus, each batch is considered independent from the previous batches. These stateless models are suitable in cases where one wants to model sequential data, but the sequences in a certain batch are independent from the sequences in the previous batch [12].

If sequence n in a batch is related to sequence n in the previous batch, one can use stateful cells. This means that the final states of a batch are used as initial states for the next batch. If we let sequence n in the first batch contain input values 1 to k and target value k of a data series, and let sequence n in the second batch contain input values $k+1$ to $2k$ and target value $2k$, we may, using a stateful network, implicitly model dependencies longer than k without using more than k steps in the backpropagation for each target value. In the case of presence detection, this can be advantageous for multiple reasons. One reason is that the feature should work at night. Since a home with everyone asleep most likely is similar to an unoccupied home, data from the evening most likely is relevent for the classification of homes during the night. Since LSTM-networks with long input sequences take much longer to train, the implicit incorporation of data points further back in time, is a more viable option [12].

3.1.7 Input Normalization

Input normalization is a crucial preprocessing step in machine learning. It involves transforming numerical features to a common scale, without distorting differences in the ranges of values. This process is important for several reasons. Normalization speeds up the learning and convergence process of many machine learning algorithms. Algorithms like gradient descent converge faster when features are on a similar scale because it ensures a smoother optimization process [13].

In datasets with features having different scales, algorithms that are sensitive to the scale of data, such as neural networks, might end up attributing higher significance to features with larger scales. Normalization ensures that each feature (if relevant) contributes to the final prediction. Normalization can also reduce the impact of outliers, as the resulting scale limits the range of values a feature can take [13].

3.1.8 Label Smoothing

Label smoothing is a widely used technique in classification problems. Simply put, label smoothing modifies the target values from 0 and 1 to α and $1-\alpha$ respectively. Here α is a scalar in the range $[0, 1]$. Label smoothing has several potential advantages. One is that it makes the model take label uncertainty into account. Thus the model does not train to get overconfident in predictions based on noisy ground truth. Even if the ground truth is completely free from noise, label smoothing might be beneficial, since it can prevent overfitting (see Sec. 3.1.9) and work as regularization, since the model will not go to extreme lengths to push a prediction close to 0 or 1 a little closer to the target label [14].

3.1.9 Reducing overfitting

An often occuring issue when training a machine learning model is overfitting. This occurs when the model is too complex and starts fitting to specific data points in the training set, rather than picking up more general patterns in the data, which essentially means that the model is tuned to fit noise. To combat this, there are several possible approaches. One is to simply add more training data, since this could make specific samples less important and thus the model potentially learns more general patterns. A second method is to monitor the validation performance and stop the training once the validation loss, or another metric of choice, stops decreasing. Overfitting could also be a result of a too large or complex network architecture in relation to the size of the dataset being used, so one could also try reducing the complexity of the neural network.

A slightly more sophisticated approach is the usage of *dropout* [15]. This essentially means that during each training iteration, each node in a hidden layer has its contribution to the network set to 0 with probability p. This prevents units from co-adapting too much, which makes the model more generalizable and robust.

3.1.10 Binary Crossentropy

In order to update the weights of a model to achieve a good result, a loss function is required. This is a function of the difference between the output of the model and the target values. The goal when training a model is to minimize this function. In this project, the chosen loss function is *binary cross-entropy* defined by Eq. (10)

$$L = -[y \ln(p) + (1 - y) \ln(1 - p)], \tag{10}$$

where p can be interpreted as the predicted probability that the correct class is class 1 and y is the true class label. Binary cross-entropy is the maximum log likelihood estimator of the observed data, has a smooth gradient and often gives a fast and stable convergence for binary classification tasks. Thus it is by far the most widely used loss function for binary classification [12].

3.2 Optimization

Finding the weights w which minimize the loss function L is a complex task without an obvious choice of optimization algorithm. Most algorithms utilize the gradient of L, denoted as $\Delta_\theta L$. A relatively simple optimizer function is SGD (Stochastic Gradient Descent) [12]. This method simply takes a step along the negative gradient of the loss function at the point θ_i, where θ_i is the current values of the weights. Mathematically SGD can be expressed as

$$\theta_{i+1} = \theta_i - \eta \cdot \Delta_\theta L(X_i, y_i), \tag{11}$$

where X_i and y_i is the input and target variables respectively for batch i and η is the *learning rate*, which regulates how much the weights should be updated in the direction of the negative gradient. It is important to find a suitable value of η since a value too small might lead to a slow convergence or the optimization getting stuck in a bad local minimum. However, using a too large learning rate can result in the learning process jumping over or overshooting the minima.

A more advanced method is ADAM (Adaptive Moment Estimation) [16], which utilizes running averages with exponential forgetting of the first and second order moments of the gradient. Equations (12)-(16) define the ADAM method. The updates of the weights θ are presented in Eq. (16),

$$m_{i+1} = \beta_1 m_i + (1-\beta_1)\Delta_\theta L_i \tag{12}$$

$$v_{i+1} = \beta_2 v_i + (1-\beta_2)(\Delta_\theta L_i) \tag{13}$$

$$\hat{m} = \frac{m_{i+1}}{1-\beta_1} \tag{14}$$

$$\hat{v} = \frac{v_{i+1}}{1-\beta_2} \tag{15}$$

$$\theta_{i+1} = \theta_i - \eta \frac{\hat{m}}{\sqrt{\hat{v}}+\epsilon}, \tag{16}$$

where ϵ is a small number ensuring no division by 0 occurs and β_1 and β_2 are forgetting factors for the first and second order moment of the gradient, respectively. Throughout this project, a high level python library based on tensorflow, Keras [17] has been used for building the neural networks. Keras uses the default values: 10^{-7}, 0.9 and 0.999 for ϵ, β_1 and β_2, respectively. The only parameter tweaked throughout this report is η. All variables except for the hyperparameters mentioned in this section and L are vectors. The squares and square roots are calculated element-wise.

3.3 Metrics

In this section, the evaluation metrics used in the report are introduced. These metrics are used to determine which model is the best at distinguishing the two classes (0 and 1) from each other.

3.3.1 Accuracy

Accuracy is a relatively simple metric for binary classification-problems defined as

$$Accuracy = \frac{TP+TN}{FP+FN} \tag{17}$$

where TP, TN, FP and FN stand for true positives, true negatives, false positives and false negatives, respectively. This metric is simply the fraction of classifications the model got correct, and contains no information of how certain the model was on the predictions or whether it was better at identifying one class than another.

3.3.2 AUC

One of the most used metrics in the report is the AUC (Area Under Curve). In order to understand this metric, one has to understand the ROC (Receiver

Operation Characteristic) curve. This curve plots the true positive rate (TPR) against the false positive rate (FPR) which are defined as follows

$$TPR = \frac{TP}{TP + FN} \tag{18}$$

$$FPR = \frac{FP}{FP + TN}. \tag{19}$$

For a model, every point on the ROC curve corresponds to the TPR and FPR for a certain classification threshold. This threshold is the output value required by the network to make the final prediction a 1. The AUC is a metric which measures the area under the ROC [18]. The optimal value is an AUC of 1, since this is the area achieved if the model can distinguish the classes perfectly regardless of the threshold. An AUC value of 0.5 corresponds to the case when the model cannot distinguish the classes at all and is no better than tossing a coin. Figures 4-5 show two possible ROC curves for a binary classification problem. Figure 4 displays the ROC curve for a model, which perfectly distinguishes between two classes. Figure 5 shows a ROC curve based on a model, which has no ability to separate the two classes. The orange lines are the ROC curves and the dotted blue lines are the ROC curve for a model with equal TPR and FPR for all threshold values.

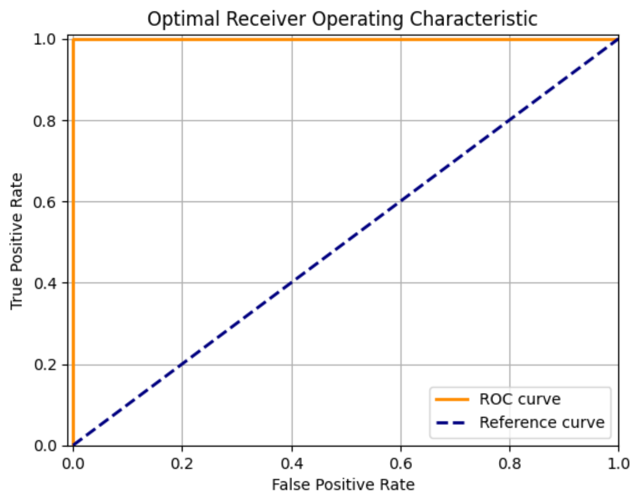

Figure 4: An optimal ROC curve. The orange line is the ROC curve and the dotted blue line is the ROC curve for a model with equal TPR and FPR for all threshold values.

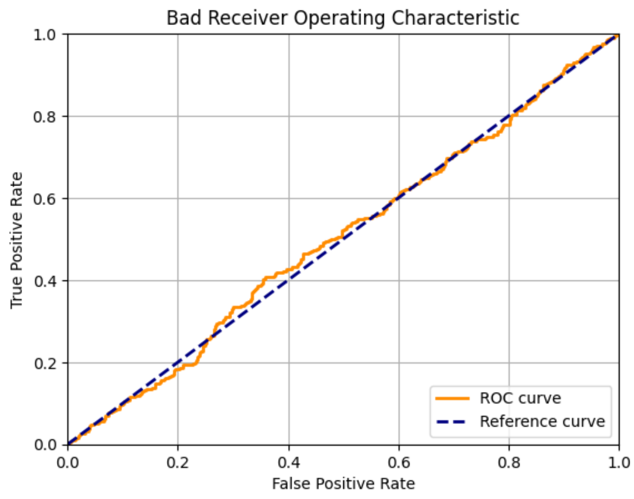

Figure 5: A ROC curve for a model. The orange line is the ROC curve and the dotted blue line is the ROC curve for a model with equal TPR and FPR for all threshold values.

To get the AUC value for a model, Keras' built in metric "AUC" was used. This metric calculates the area by calculating the ROC points for a number of threshold values evenly spaced between 0 and 1. Given these ROC points the AUC is calculated by a Riemann sum.

4 Data

4.1 Devices, Users and Homes

Every Minut device collects data such as humidity, sound level and ambient light intensity. A device looks like a fire alarm and could be set up anywhere in a home. Figure 6 displays such a device in a home.

Figure 6: A Minut device, which collects data about sound level, motion events, light intensity and humidity.

There could be more than one device in a home, which means that each home could potentially have several relevant data series. Every device is linked to a home and there exists a lookup table containing information on which home a device belongs to. Another part of the Minut system is users, which is an account on their user platform. A user is linked to one or more homes.

4.2 Evenly sampled Sensor Data

Most data in the set is collected on a device level and sampled once every 60 seconds. Examples of such data are humidity, light intensity, temperature and sound level. The values of these data points are continuous in a range defined by hardware limitations. For instance a sound level of less than 30 dB or more than 90 dB cannot be measured using the hardware in the devices.

4.3 Event Data

Since part of the service Minut offers is detecting certain events, some data comes in the form of unevenly sampled "device events". An example of such events is motion events, which are recorded and sent to the company database if a device, using a PIR sensor, records motion activity. The event data only says which type of event has been recorded and when. Thus, no information on the magnitude of the measured signal is included in the motion data.

4.4 Geofencing-data

Minut previously provided a presence detection feature based on geofencing data, but due to a lack of commercial advantages (the home alarm feature this was used for is not relevant for Minuts' current customer base of short-term rental guests) and inconsistent results, the feature was removed. The feature did, however often work well and thus it might have the potential to work fairly well as ground truth after some filtering. The data was collected as events recorded every time a mobile device with the Minut app crossed a virtual border set around a certain home. When a mobile device enetered the home zone, it was recorded as a 1 and when it left the zone, it was recorded as a 0. For each event, the home, which the mobile device had entered or left, was provided. In addition, an event included information on which mobile device it was and which account (user) the device belonged to. During some time periods for some devices (which periods and devices is not known) spurious updates were sent even if a border was not crossed. To clarify, some devices sent updates of whether they were located around the home or not even at times at which their states were not changed. The geofencing data suffered from a few issues, which are presented below.

The first issue was that the data series contained holes. That is, a long time could pass between two consecutive data points. Whether a hole was due to no people entering or leaving a property, or due to bugs in the data collection, was not possible to decide, since the dataset contained no information of whether the geofencing-based service was turned on or off and what type of devices were used to collect the geofencing-data. The latter would have been useful to know, since IOS and Android devices had different rules for when to send an update of its position in relation to the geofence. Another issue encountered was users using the feature for a short amount of time at the beginning of the data series and after a long break start using it again. Furthermore, homes could get stuck in one of the two states at the start of the data series, which most likely was not due to people being more stationary during the first weeks of data collection. Some of the homes never left their initial states. In addition to this, some series were very short, which might indicate that the service did not work properly, and both presence detection methods later proposed required multiple data points in order to make one prediction.

There did also appear to exist errors on device and user level. For instance, some homes seemed to have reasonable data with the exception of one or few devices or users, who appeared to always be at home or away. Most other issues described on a home level could be found on a device or user level as well. In addition to this, some devices and users would permanently stop recording data in the middle of the data series. Due to the issues presented in this section, The geofencing data required a bit of preprocessing, which is described in Sec. 5.1.1.

5 Method

5.1 Preprocessing

Initially the data was divided by data type and not by device. In order to turn the data into something useful it had to be sorted by sensor, home and data type (for instance light intensity). This was done by firstly creating a set of all devices with geofencing data. Thereafter, the data on humidity, sound level, motion and light intensity from these devices was collected and saved with the presence data.

5.1.1 Geofencing data

The presence data for a home was, as described above, a series of events which had a state (0 or 1), a device, a user and a mobile client ID. Due to errors related to the geofencing, the fact that the set was unevenly sampled, and that multiple users, devices and mobile clients were relevant to determine whether someone was present, the following preprocessing steps were made.

The first issue encountered was that there were holes in the data. In other words, there could be days or even weeks between consecutive data points. Since there was no information available about which mobile devices were used and whether the geofencing service was on or off, the presence series were split into pieces, which had no holes of more than 3 days. The choice of 3 days was based on the assumption that longer time periods without any data recorded has a sufficiently high probability of being a result of the feature being turned off and thus the labeling being incorrect. Once the time series were split, the newly created sections lacked suspicious holes, but still contained a sufficient amount of data to be processed later on their own. This might have introduced a small bias towards series with much activity, but since it was not possible to determine if the holes were due to a damaged series or a lack of activity around the home for devices without spurious updates, the splitting was deemed the most reasonable choice. Due to this splitting, a home could have several series related to itself. Another issue encountered was users testing the geofencing feature for a short amount of time before turning it off for a longer period. Whether the feature was on or off could not be found in the dataset used in this project. This was accounted for by the splitting mentioned above and by removing the first few points from each sequence. Another issue was that some homes seemed to get stuck at the beginning of the series in either state 0 or 1. Therefore, all events before both a 0 and 1 had been registered were deleted from the dataset.

Series with less than 5% of the points in one of the states (0 or 1) were removed. This was done since visual inspection of the series indicated that it was likely that they got stuck in a state and yielded many outputs of one type in a short amount of time.

It was still possible that a mobile device or user object was faulty and returned only one class. Therefore all series with a user or phone only containing zeroes or ones, were thrown away. The reason why the whole series was thrown away being that if one user or phone was thrown away, a home only occupied by this user or phone would be incorrectly labelled as unoccupied.

Some mobile devices would stop being used during the time window of a series. If the last state of a phone was a 1, then a phone never used by anyone in the household could potentially label someone as permanently home. To avoid this, all data points from a mobile after the last 0 were thrown away. The idea being that a mobile not used would not lock a home in state 1.

Another scenario not covered by the above steps is if a single mobile device temporarily gets stuck in returning 1. To combat this, a check for sections where a mobile client returned only 1s for 6 days was labelled as likely incorrect and the whole series was removed. The choice of 6 days was based on the assumption that people tend to leave their property at least once during 6 days. It is also worth noting that there was a trade-off between the risk of having incorrect labels during a couple of days, losing data and introducing a bias against data where people do not leave their properties.

Finally the series which passed all the checks in this section was passed through a final preprocessing step. In all homes an aggregated presence column was created using an OR operation on all users in the home. To clarify: if for all users, the last registered state was 0, a 0 was added to the new column. Otherwise a 1 was added to the column. Thus this new column represents whether a home is empty or occupied. A summary of the preprocessing for the geofencing data can be found in Fig. 7.

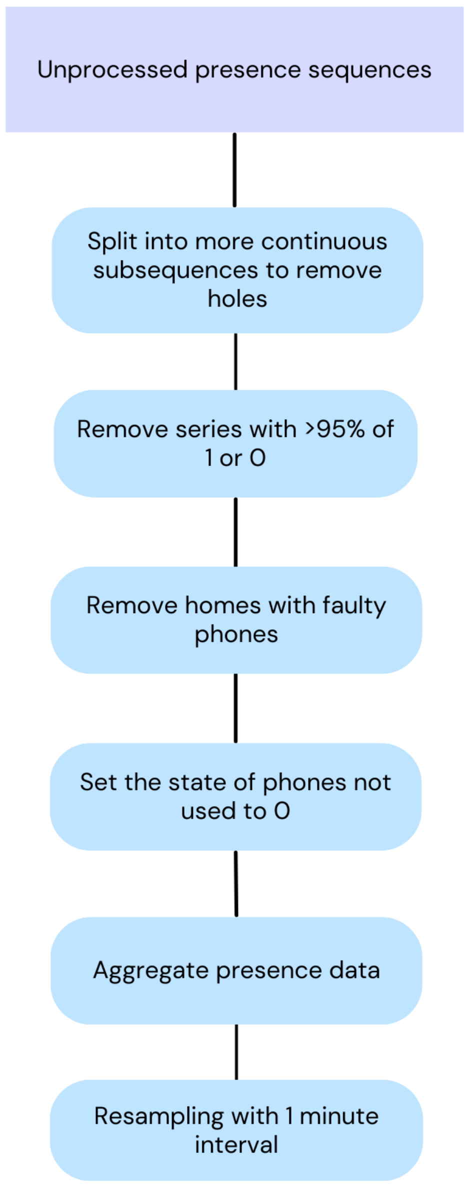
Unprocessed presence sequences
Split into more continuous subsequences to remove holes
Remove series with >95% of 1 or 0
Remove homes with faulty phones
Set the state of phones not used to 0
Aggregate presence data
Resampling with 1 minute interval

Figure 7: Outline of how the presence data was preprocessed.

5.1.2 Merging Data

In order for the collected data to be useful, the series for different sensor values had to be combined into one structure. This was done by first discarding all sensor data outside of the the window in which presence data was available. Thereafter, all data was resampled with a 1 minute interval.

The motion data was also resampled to have a value for every value in the other sensor data series. Every motion event was rounded in time to the nearest time for which other sensor data was recorded. All data points with a rounded motion event was given a 1 in the motion column. The other points were set to 0 in terms of motion.

5.1.3 Splitting the dataset

Since stateful LSTM-networks in Keras need to have sequences of equal length, only sequences with more than 100 000 data points were used. These were then cropped to only contain 100 000 points. Choosing 100 000 data points per sequence was a result of the trade-off between having long sequences in the dataset and not removing too many sequences. The former is important, since these sequences most likely are of good quality, since the service was used continuously. Having data from many different sensors is important, since it most likely improves the generality of the model. Another limitation of the stateful LSTM-networks is that the validation set and training set need to have the same number of sequences. Thus 310 sequences were randomly selected for training, 310 of the remaining sequences were randomly selected to be the validation set, and the remaining 119 sequences were used as the test data set. Towards the end of the project it was discovered that one could zero-pad the validation and test sets, which would allow a greater portion of the dataset to be used for training. Due to time constraints this approach was not employed.

5.1.4 Normalizations and motion interpolations

Various approaches and parameters for normalization of the data were considered. The one used for humidity and sound level was subtracting the lowest value the hardware of the sensor allowed and then dividing by the highest allowed value to ensure that inputs were not greater than 1. In the case of light, using the hardware limit would result in too many very small values and thus a normalization constant, which tended to yield normalized values similar to the other data types, was used. In order to make it possible to downsample the motion signal, and also possibly in order to improve generalization, the motion signal was interpolated by replacing all zeros with

$$m = e^{-0.05 \cdot t} \tag{20}$$

where m is the new interpolated value of the motion signal and t is the number of minutes since the last motion event occurred. The idea behind this interpolation being that a large value indicates that movement activity recently was registered and a small value means that a longer time has passed. Intuitively, the interpolation is also useful since it passes potentially relevant information from data points not included in the input vector for a certain data sample. More simply put there might be an advantage to differentiating between the case when motion was recorded a few hours ago and when the last recorded motion was a week ago. Since most data points were from houses with someone present (65%), weighting of the classes was tried for both models. The loss from a certain data point was multiplied by one weight, if the true state label was a 1, and with another weight if the true state label was a 0. The tried weights where equal weighting of the two classes and weighting inversely proportional to the frequency of the classes in the data set.

5.1.5 Downsampling

Downsampling was tested as a means of reducing the amount of input data, while still capturing long term dependencies. This would allow a model to look twice as long back in time, without increasing the model size or complexity, which could potentially increase model performance. The hyperparameter, which determines how much the signal is sampled down, is called "downsampling step" in this report. A value of 2 of this parameter means that every second data point is used in the input vector.

5.2 Models

5.2.1 Moving window

One approach tried was a moving window approach using standard feedforward layers. This means that for every state value, the input is the n last values from the input series (motion, humidity, sound level and light). The idea behind this approach is that it should be possible to create a classifier which only uses information from the last h hours, since, intuitively, data from several days ago should not be of very big importance to detect whether someone currently is present.

5.2.2 LSTM

The other approach used was a stateful LSTM-network. The reason for this is that stateless networks would need long input sequences in order to include data points multiple hours back in time in the classifications. Training LSTM-networks with long input sequences is very time consuming, which meant that too few training processes could have been carried out throughout this project if stateless networks had been used. Stateful LSTM-models were tested, since these, without increasing the complexity of the model, implicitly use data from time points further back in time.

For each target output, the input was the n last values from the input series. The target values for the first batch were presence value n in each device series and the inputs were thus motion, humidity, sound level and light values number 1 to n. For the second batch, the target outputs were presence value $2n$ and the input were thus the input features for the times $n + 1$ to $2n$, etc.

5.3 Tuning of hyperparameters

Since the training time per model was more than an hour in general and the multidimensional space spanned by the possible values of hyperparameters is very large, the hyperparameters were tuned one at a time. That is, for each parameter a line search was performed. Once the value, yielding the best AUC on the validation set, was found, that hyperparameter value was fixed and used in all subsequent training iterations. The initial parameters were based on less rigorous experimentation prior to the line searches. For the network architecture, four different network structures were considered: The first was a bottleneck structure with 3 layers where layer 1 and 3 were equally wide and layer 2 had half the width of the others. The reasoning behind this structure was that it can enhance the network's efficiency and performance by forcing it to learn a compact and informative representation of the input data, thereby improving generalization and reducing computational cost [19]. The second structure was a flat structure with 3 layers. The third was a bottleneck structure with 5 layers, where layers 1 and 5 were equally wide, layers 2 and 4 were of three quarters the width of the first layer and layer 3 was half as wide as the first layer. Finally, a flat structure with 5 layers was considered. For each structure, different widths of the first layer (and therefore of all layers) were tested. This process was repeated for both the LSTM-cells and for the moving window method.

5.4 Generators

A file which for every state value stores the n last input values would be much larger than the available RAM on the machine learning computer at Minut. Thus the input data had to be generated piecewise. This was done by implementing a generator class inheriting from Keras' sequence class. Due to the fact that the data points need to be processed in chronological order for LSTMs, but not for the moving window approach, different generators were implemented.

For the moving window model, random order of the samples of the dataset was allowed. That is, the order of the subsequences to be classified was random, but internally in each subsequence the data points remained in chronological order. When using a generator which does not randomize samples, the network tended to dive too deep into local minima, since many samples in a row would be very similar. The generator used picked a row at random, generated its corresponding input array, and added the created sample to a list. Once this list had a length equal to the batch size chosen, it was returned by the generator. The rows were drawn without replacement, which means that no row was used twice during an epoch.

Since stateful LSTM-networks do not allow random order of the samples in the dataset, the data for these was generated deterministically as described in the last paragraph of Sec 5.2.2.

6 Results

6.1 Hyperparameter tuning for LSTM-approach

In Fig. 8-18, the validation AUC is plotted as a function of the hyperparameter tuned in the related line search. The figures are presented in the same order as the parameters were tuned. When examining the optimization of dropout, it was noted that there was a flaw in using the best validation AUC as metric, since this favored hyperparameters which contributed to large variations between epochs. In order to avoid hyperparameters causing too unstable training processes, the evaluation metric was changed to the highest value in a smoothed AUC vector. This smoothed vector is created by applying a moving average filter of length 7 to the vector containing the validation AUC values for every epoch in the training process.

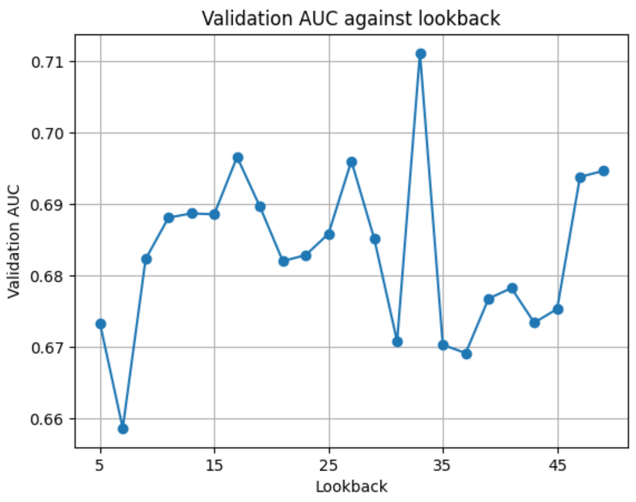

Figure 8: Best validation AUC plotted against lookback time for LSTM-models.

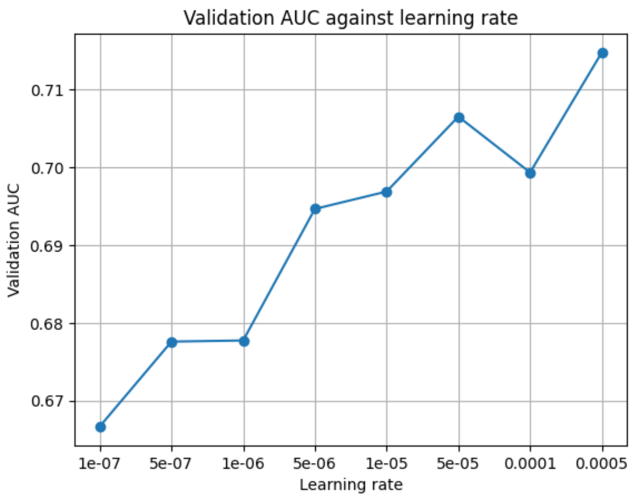

Figure 9: Best validation AUC plotted against learning rate for LSTM-models.

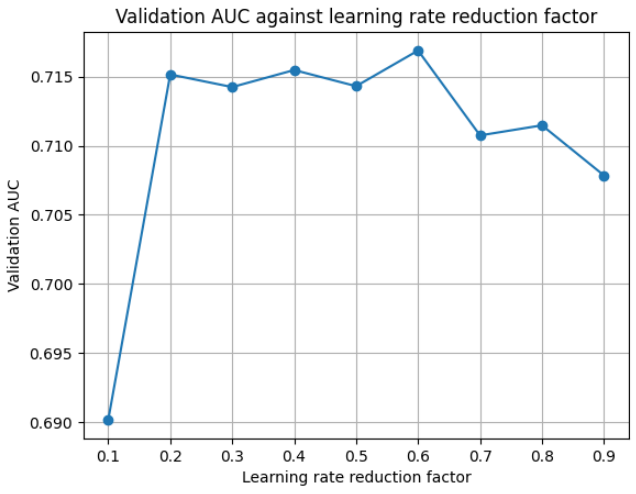

Figure 10: Best validation AUC plotted against the factor by which the learning rate is multiplied after a certain number of epochs without a decrease in validation loss. This plot is based on LSTM-models.

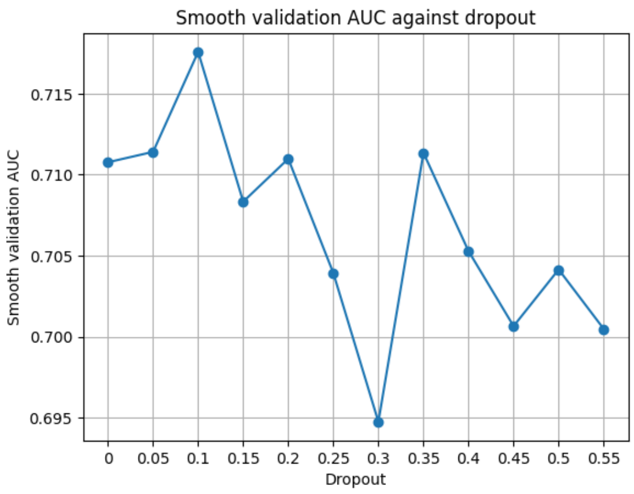

Figure 11: Best smoothed validation AUC plotted against dropout factor. This plot is based on LSTM-models.

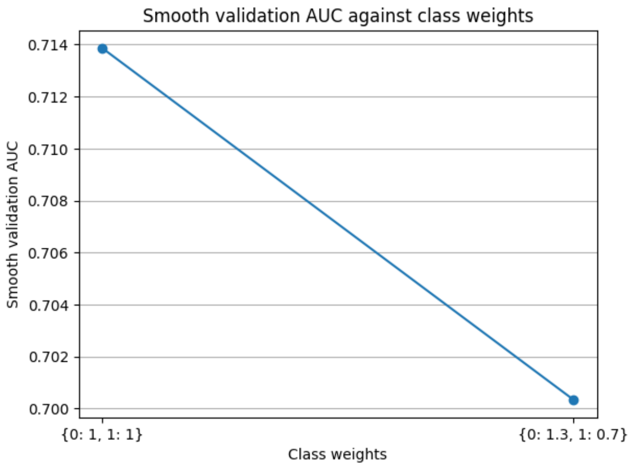

Figure 12: Best smoothed validation AUC plotted against class weights. The leftmost data point has a weight of 1 for both classes. The rightmost point has a weight of 1.3 for class 0 and 0.7 for class 1. This plot is based on LSTM-models.

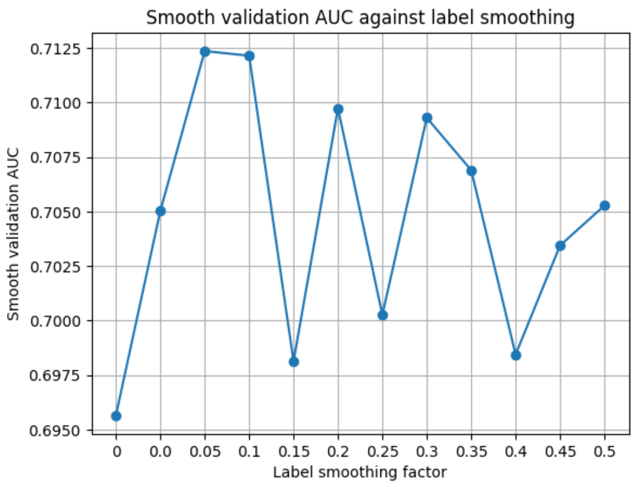

Figure 13: Best smoothed validation AUC plotted against label smoothing factor. This plot is based on LSTM-models.

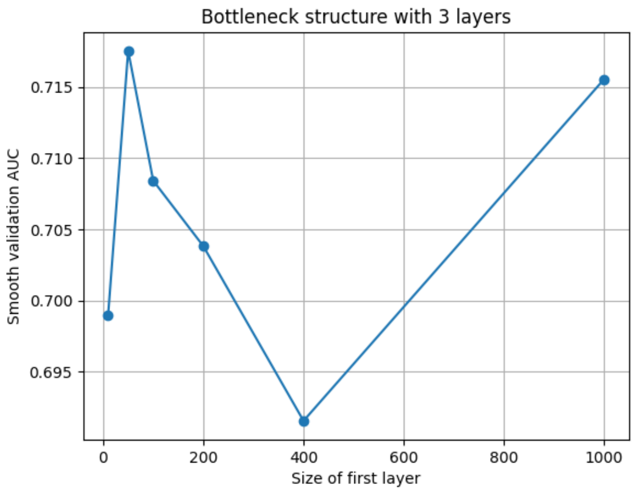

Figure 14: Best smoothed validation AUC plotted against the width of the first layer in a 3 layer bottleneck structure. The plot is based on LSTM-models.

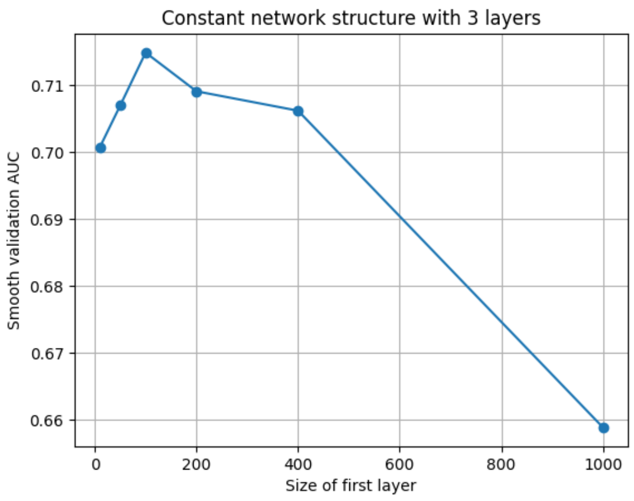

Figure 15: Best smoothed validation AUC plotted against the width of the layers in a 3 layer model with constant width. The plot is based on LSTM-models.

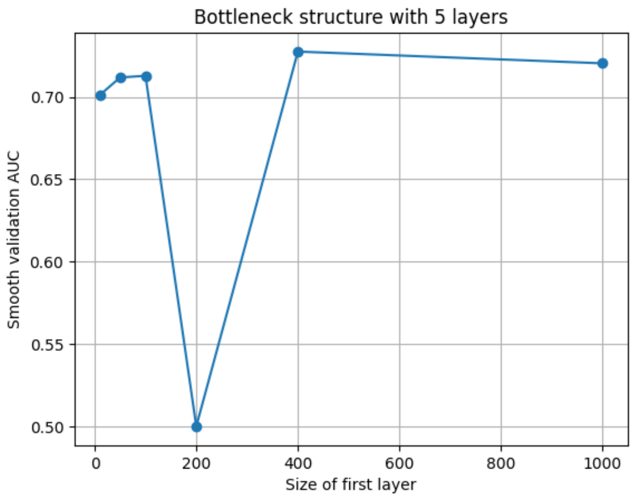

Figure 16: Best smoothed validation AUC plotted against the width of the layers in a 5 layer model with bottleneck structure. The outlier at 0.5 is the result of a model classifying all samples as state 1. The plot is based on LSTM-models.

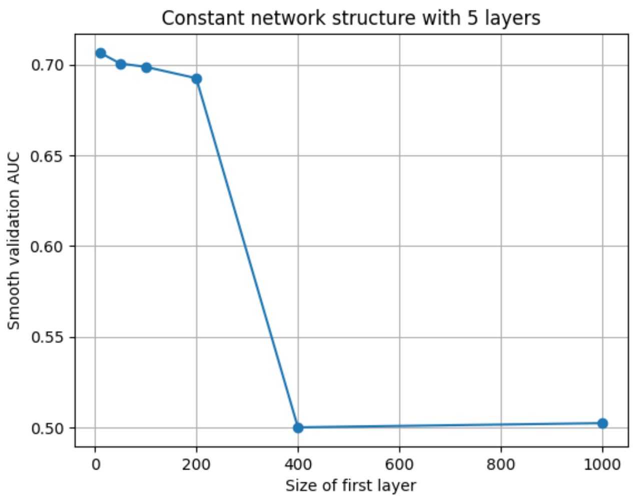

Figure 17: Best smoothed validation AUC plotted against the width of the layers in a 5 layer model with constant width. The outliers at 0.5 come from models classifying all samples as state 1. The plot is based on LSTM-models.

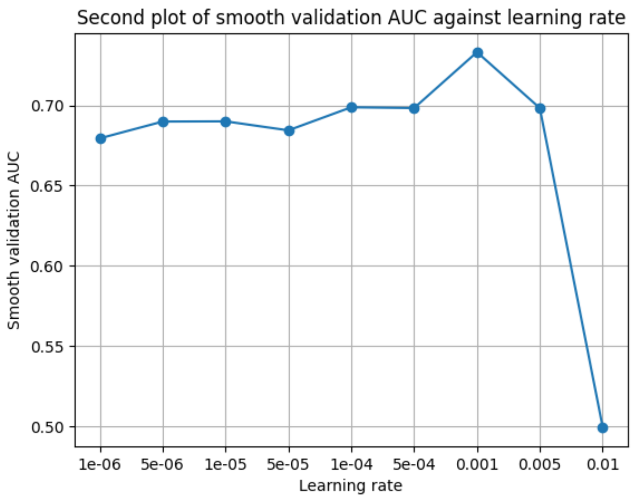

Figure 18: Best smoothed validation AUC plotted against learning rate. This plot shows the second tuning of the learning rate parameter. The outlier at 0.5 is the result of a model classifying all samples as state 1. The plot is based on LSTM-models.

In Fig. 9 the result seems to improve as the learning rate increases. This was realized after all other parameters had been tuned and thus a second tuning of the learning rate was performed lastly. The second tuning scanned over a higher range of values, since the first one did not reach its optimum. The result of the second tuning is presented in Fig. 18. In some figures, for instance Fig. 17, one training iteration got stuck on an AUC of 0.5. After closer inspection of these training processes, it was found that the training algorithm got stuck in a local minima created by guessing that someone was present at all times. It is notable that some plots (for example Fig. 13) experience a noisy behaviour. Regardless of chosen hyperparameters, all models, excluding the aforementioned outliers, yielded a smoothed AUC in the range $[0.65, 0.735]$.

6.2 Hyperparameter tuning for moving window

In Fig. 20-28 below, the best smoothed validation AUC is plotted as a function of the hyperparameter tuned in the related line search. The best smoothed AUC for a model is the maximum value in a vector of smoothed values. The filter length was shorter for the moving window method, since this method required much fewer epochs to converge. Due to this, the training was ended after fewer epochs for this method. Every epoch did, however, take more time to train, which resulted in a total training time close to the training time for the LSTM-models depending on the hyperparameters of the specific training session.

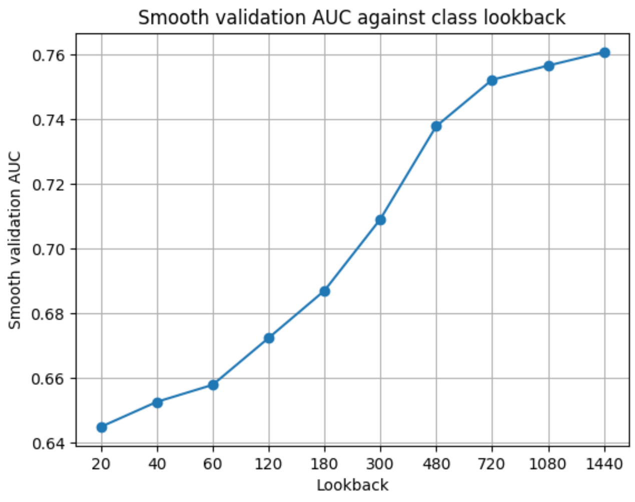

Figure 19: Best smoothed validation AUC plotted against the number of minutes the model looked back in time.

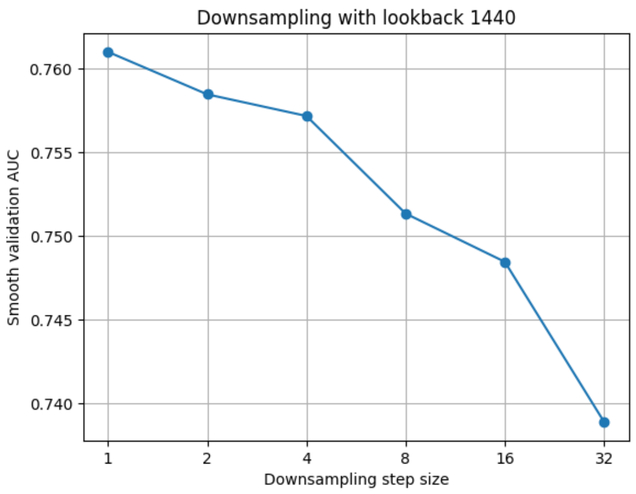

Figure 20: Best smoothed validation AUC plotted against the number of minutes between consecutive time points in the input. A lookback of 1440 minutes was used. The plot is based on moving window models.

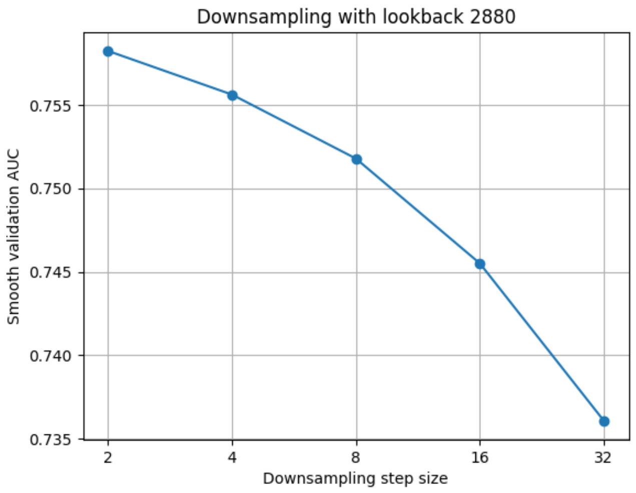

Figure 21: Best smoothed validation AUC plotted against the number of minutes between consecutive time points in the input. A lookback of 2880 minutes was used. The plot is based on moving window models.

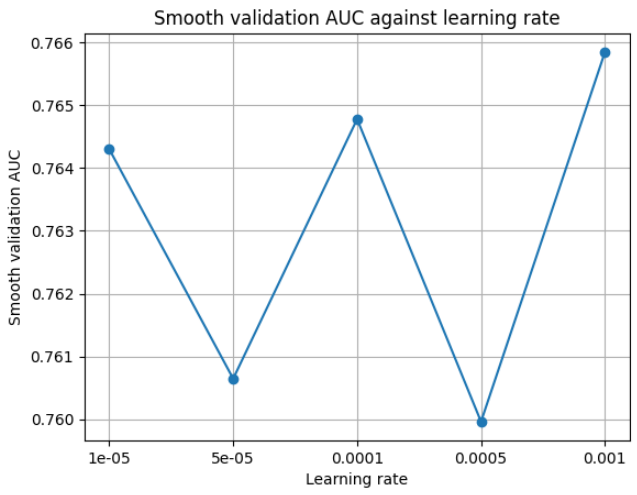

Figure 22: Best smoothed validation AUC plotted against the learning rate. The plot is based on moving window models.

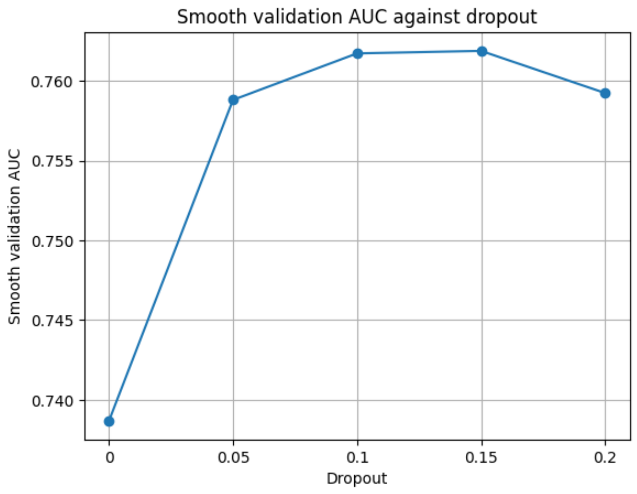

Figure 23: Best smoothed validation AUC plotted against the dropout constant. The plot is based on moving window models.

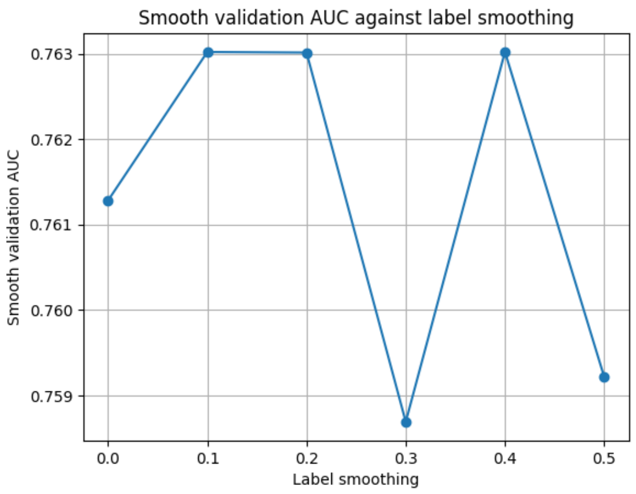

Figure 24: Best smoothed validation AUC plotted against the label smoothing parameter. The plot is based on moving window models.

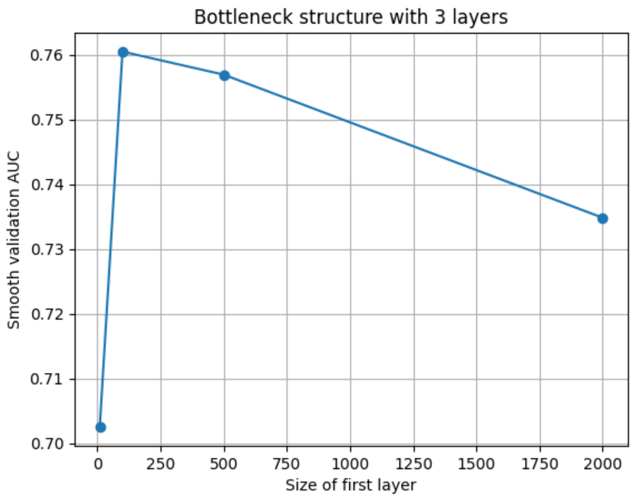

Figure 25: Best smoothed validation AUC plotted against the number of nodes in the first hidden layer of a network with bottleneck structure. The leftmost point is the result of a model with 10 nodes in the first layer. The plot is based on moving window models.

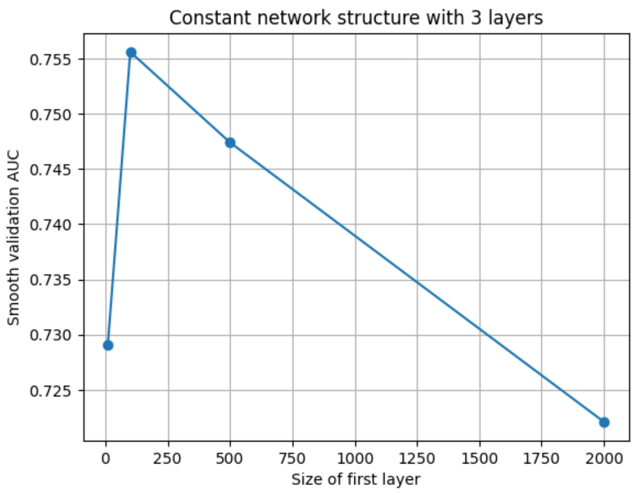

Figure 26: Best smoothed validation AUC plotted against the number of nodes per layer in a network with a flat structure. The leftmost point is the result of a model with 10 nodes in the first layer. The plot is based on moving window models.

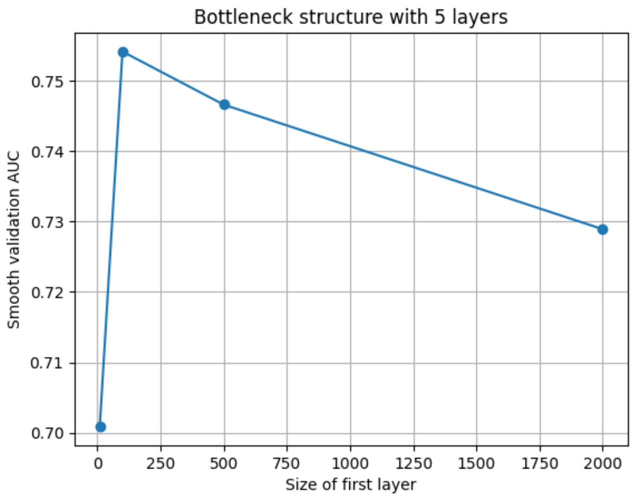

Figure 27: Best smoothed validation AUC plotted against the number of nodes in the first hidden layer of a network with bottleneck structure. The leftmost point is the result of a model with 10 nodes in the first layer. The plot is based on moving window models.

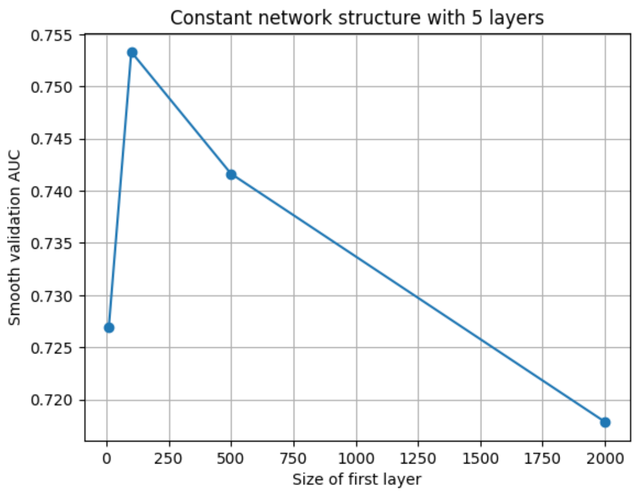

Figure 28: Best smoothed validation AUC plotted against the number of nodes per layer in a network with a flat structure. The leftmost point is the result of a model with 10 nodes in the first layer. The plot is based on moving window models.

Given that no downsampling was employed, the model seems to get better as the lookback time increased. However, no models processing more than 1440 timesteps were considered, since that would require too long training times to complete the rest of the hyperparameter optimization. The time required to train a model had a close to linear relationship to the number of time points the model looked at. Since moving window models took slightly longer to train than LSTM-models, each line search contained fewer tested values compared to the LSTM-search. After setting the lookback time to at least 1440 minutes (24 hours), all tested models yielded a smoothed validation AUC in the range $[0.7, 0.77]$.

6.3 Final Models

Based on the hyperparameter tuning performed in Sec. 6.1-6.2, two models (one moving window model and one LSTM-model) were chosen for evaluation on the test set. The hyperparameters of these models can be viewed in Tab. 1.

Table 1: The resulting hyperparameters after the tuning in Sec. 6.1-6.2

Parameter	**LSTM**	**Moving Window**
Lookback	31	1440
Downsampling Step Size	1	1
Learning Rate	0.001	0.001
Lr Reduction Factor	0.6	0.6
Dropout	0.1	0.15
Class Weights	None	None
Label Smoothing	0.05	0.05
Model Structure	Bottleneck 5 layers 400 nodes	Bottleneck 3 layers 100 nodes

6.4 Results on the test set

In Fig. 29 and 30 the ROC curves for the test set are provided. The resulting metrics on the test set can be viewed in Tab. 2.

Table 2: Accuracy and AUC for the test set.

	AUC	Accuracy
LSTM	0.6920	0.6901
Moving window	0.7205	0.6982

The AUCs are slightly lower than for a typical result on the validation set, but the models are still clearly better than a model which blindly guesses, since such a model would have an AUC of 0.5.

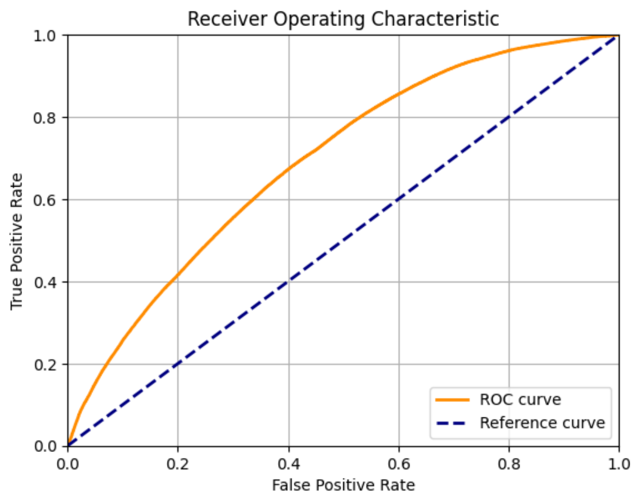

Figure 29: ROC-curve for the test set. The plot is based on the predictions made by the LSTM-model. The yellow line is the ROC curve and the dashed line is a reference line for a model, which blindly guesses its predictions.

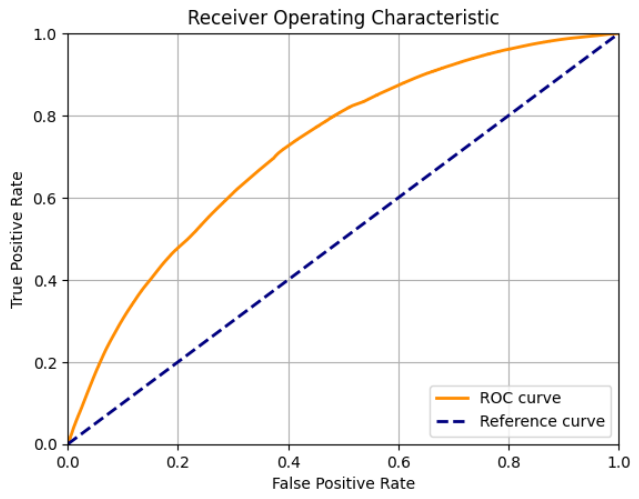

Figure 30: ROC-curve for the test set. The plot is based on the predictions made by the moving window model. The yellow line is the ROC curve and the dashed line is a reference line for a model, which blindly guesses its predictions.

7 Discussion

When evaluating the result, one of the most important things to note is that throughout the tuning of the hyperparameters only one data point per tested hyperparameter value is available. Since the initial weights of the models are random and the order in which the data is processed is random for the moving window model, the method used for choosing the best hyperparameters lacks some statistical significance. The noisiness of the resulting value of the validation metric is visible when the default value of a hyperparameter has been tested in a line search. An example of this is when class weights were tested for the LSTM-model. The leftmost data point in Fig. 12 has the same parameters as the third data point in Fig. 11, but the results are slightly different. Intuitively the curves should be relatively smooth and not have very large oscillations. Looking at, for instance, Fig. 13, it seems that noise dominates the variations. This means that it is difficult to make any conclusions based on some of the line searches made. It is possible to argue that a fixed random seed would solve this issue, but this could introduce a bias towards hyperparameters suitable for one specific random seed, which might lead to networks too specialized on the validation data.

It is possible that a grid search over the hyperparameter space would yield better results, since the optimal value of one hyperparameter depends on the value of another. This would have to be done with grids over smaller subsets of hyperparameters, since a grid search over a high dimensional hyperparameter space would take too much time with the computational power at hand. However, fewer values of each parameter would be tested, which could lead to worse results if the smaller set of tried parameters do not contain a good enough value. It is also possible that a smaller multidimensional search over the hyperparameter space could be done, but since poor data quality appears to be the main issue, a smarter hyperparameter search would most likely not improve the result very much.

An important part of this project is how the geofencing data was preprocessed. This most likely had a large impact on the results, since inaccuarate ground truth means that the models will only model noise for the inaccurate data points. One of the least obvious steps in the preprocessing is the splitting of sequences into more continuous subsequences. Although this might have unnecessarily deleted data and introduced some biases, it most likely resulted in a more correct ground truth. If holes had not been removed, homes with the geofencing feature turned off would contribute with large quantities of wrongly labeled data points. However, it is possible that this issue could have been solved in another way or that the choice of 3 days as the maximum allowed hole size was suboptimal, but it is difficult to find a systematic method to device a good value for the allowed hole size. Other parts of the preprocessing suffer from the same problem, since there was no way of testing the preprocessing on its own.

It is notable that the moving window method outperformed the LSTM-method. This difference could have several explanations: Firstly, this model had longer input sequence and was therefore most likely able to pick up more complicated patterns. Another relevant difference between the models is that the LSTM-model only used $\frac{1}{17}$ of the data points, since it was stateful and a subsequence had to start at the time after the previous subsequence stopped. When monitoring the training process it was evident that the LSTM-networks overfitted the training data more than the moving window networks. This could partially be due to the more complex nature of the LSTM-cells, but the effect can also be attributed to the smaller amount of data points in the training set. Due to the statefulness, the validation and test sets for the LSTM-models also contain fewer data points. This means that the evaluation of the LSTM is less statistically significant and that some differences in terms of performance can depend on the fact that the data sets were different. Since the number of points in every data set was large for both models and the data sets for the LSTM-models were subsets of the sets for the moving windows, this difference should not have very large effects. Another interesting observation is that the LSTM-method was closer to the moving window method in terms of accuracy than AUC. Although the two metrics cannot be directly compared this way, it is a reasonable conclusion, since LSTM-networks often classified more samples as 1, which is advantageous in terms of accuracy, but not in terms of AUC. It is also possible that the moving window model was more confident of its correct predictions, which affected the AUC, but not the accuracy.

The results on the test set were slightly worse than a typical result on the validation set. This may depend on the fact that the hyperparameters have been tuned based on validation performance. However, since the parameter tuning was very noisy, this effect should be very small. Another possibility is that the test set is less similar to the training set. This is, however, fairly unlikely, since the split into training, validation and test set was completely random and the data set contained many independent sequences.

Since the goal is to create a model good enough for commercial applications, it is relevant to set the results into context. The most important possible applications are management of heating systems and other energy conservation based on presence. For such a feature to be practically useful, the accuracy would have to be over 95%. Although some correct patterns have been identified and the result is much better than plain guessing (an AUC of 0.5), the results are far from good enough for the intended applications of the model. It is not clear whether this depends on poor ground truth, bad choices of features, hyperparamaters, and normalizations, or an inherent difficulty in determining whether someone is present based on some fairly limited sensor data from a single sensor in a single room.

8 Future work

Throughout this project several interesting ideas had to be set aside due to a lack of time. In this section some of these ideas are presented. A possible area of improvement is the normalization. One technique, which could be tested, is normalizing the input window based on itself or the values of the n previous windows. The idea being that it might yield better results to put larger emphasis on local differences, rather than the absolute values of the input values. The exception to this idea might be the motion data, which intuitively should not be too affected by background noise and varying base levels. One could also consider normalizing the data in a more sophisticated manner to make the normalized data follow some distribution (e.g normal distribution). For the motion data more constants could be tested in Eq. (20), since there is no empirical evidence that -0.05 is the optimal choice. It might also be worth testing other interpolation methods and to investigate whether interpolation is needed for this data.

In terms of improving the reliability of the ground truth, it would be helpful to get information about whether the geofencing was based on IOS or Android devices. This would make the filtering and interpretation of the data series easier. If the amount of available data would be large enough, a possible way of moving forwards is to only use homes which had no android devices connected to the geofencing service.

If one had more time or computational resources, or if the dataset became smaller due to the point above or any other reason, it might be worth considering using more of the data for LSTM. This could either be done by using stateless models, or by using both the sequences starting at time point 1 and the sequences starting at 2 in batch 1. in the following batch, both the sequences starting at time n and at $n + 1$ would be used. In order to increase the amount of training data, one could zero-pad the validation data and thus allow for a greater proportion of the data to be used as training data.

Something not investigated in this report is the properties of the predicted sequences. That is, how does the prediction vector look? Is it noisy or smooth? If it contains fluctuations between predicted zeroes and ones, a naive, but possibly effective, solution would be to smoothen the predicted sequences by performing max pooling on windows moving over the sequences. It is also quite likely that the predictions have some delay compared to the ground truth. This potential effect should be investigated further to see if any changes could be made to combat this.

During the last couple of months, BLE data was collected from a small number of homes. Since this most likely is a more reliable ground truth, it might be a good idea to collect more data of high quality and use it to train models. That would make it possible to make conclusions about which errors can be attributed to errors in the ground truth and which depend on bad model choices or an inherent difficulty in predicting presence with only the available data. As stated above, the collected BLE data comes from a small number of homes. Should this dataset be too small to train models, it could be used to test models created using geofencing data. This test could provide information on how similar the datasets are and could thus say something about the quality of the geofencing data.

This project also has not investigated different choices of activation functions and learning algorithms. Although this most likely would not have a huge impact on the end result, it might be worth researching. It could also be worth trying and researching different network architectures, since only constant width and bottleneck structure have been tested in this project.

It is not obvious that every input type (e.g humidity) contributes to the classifications. Therefore it would be a good idea to investigate what happens if a feature is replaced with noise or if a model is trained completely without one of the features. If one or more features are deemed irrelevant, it would be possible to make a smaller and faster model with at least as good performance as the model with all four input features.

References

[1] Thiago Teixeira, Gershon Dublon, and Andreas Savvides. A survey of human-sensing: Methods for detecting presence, count, location, track, and identity. *ACM Computing Surveys*, 5, 2010.

[2] Avgoustinos Filippoupolitis, William Oliff, and George Loukas. Occupancy detection for building emergency management using ble beacons. In *Computer and Information Sciences*. Springer International Publishing, 2016.

[3] Reips Schevchenko. Geofencing in location-based behavioral research: Methodology, challenges, and implementation. *Behavioural research methods*, 2(I), 2023.

[4] Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6), 1958.

[5] Schmidhuber Hochreiter. Long short-term memory. *Neural Computation*, 9(8), 1997.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Łukasz Kaiser Aidan N. Gomez, and Illia Polosukhin. Attention is all you need. *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33. Curran Associates, Inc., 2020.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6), 2017.

[9] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 503:92–108, 2022.

[10] Mayranna. Perceptron, 2013. This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.

[11] Sky. This is the structure of a classical multi-layer perceptron (mlp), 2012. This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[13] Kuhn and Johnson. *Applied Predictive Modeling.* Springer, 2013.

[14] Rafael Müller, Simon Kornblith, and Geoffrey E Hinton. When does label smoothing help? In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[15] Hinton G. Krizhevsky A. Sutskever I. Salakhutdinov R Srivastava, N. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1), 2014.

[16] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Published as a conference paper at ICLR 2015*, 2017.

[17] Keras: Deep learning for humans. https://keras.io/. Accessed: 2024-02-02.

[18] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[19] Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. *2015 IEEE Information Theory Workshop (ITW)*, 2015.

Master's Theses in Mathematical Sciences 2024:E9
ISSN 1404-6342

LUTFMA-3526-2024

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/