Universität Regensburg

Location prediction for pedestrian indoor navigation using smartphone WiFi sensors

Bachelorarbeit im Fach Medieninformatik am
Institut für Information und Medien, Sprache und Kultur (I:IMSK)

Note for online publication

- Code for the mentioned Android apps is not included; it's predominantly not mine I don't know if the systems are designed to have endpoints published safely in this manner.
- Code for the artificial intelligence and server communication parts of this thesis is available here: https://github.com/BaumerEDV/Classification-Backend (the purpose of each file is explained in the thesis)

Table of Contents

Zusammenfassung

In dieser Arbeit wurde eine Ansammlung von Software erstellt, die es erlaubt annotierte WLAN-Daten in Räumen der Universität Regensburg zu messen, welche dann zum Trainieren eines Klassifikators, der den Raum bestimmt, aus dem mit der höchsten Wahrscheinlichkeit eine nicht-annotierte WLAN-Daten-Messung stammt, und so die Position des messenden Gerätes (im Anwendungsfall ein Smartphone) vorhersagen kann. Dieser Klassifikator wurde dann durch einen Server für die UR-Walking-Fußgängernavigations-App zugänglich gemacht, sodass diese nun in Echtzeit Vorhersagen über den aktuellen Raum des Nutzers erhält. Die konkrete Implementierung wurde durch ein Abändern einer bestehenden Kotlin-Android-App zum Messen der WLAN-Daten durchgeführt; Es wurden, auf Basis von WLAN-Daten aus dem Vielberth-Gebäude aus einer ersten Messung unter Verwendung der zuvor erstellten App, mehrere Klassifikatoren mittels Python und der sklearn-Bibliothek optimiert und daraufhin anhand von Daten aus einer zweiten Messung aus dem Vielberth-Gebäude getestet. Der beste Klassifikator – Random Forest, mit einer durchschnittlichen Genauigkeit von 96,9% pro Vorhersage pro Raum – wurde dann abgespeichert und wird von einem Python-Web-Server verwendet, der in Form von JSON WLAN-Daten empfängt und in JSON die Vorhersagen des Klassifikators zurücksendet. Diese JSON-Kommunikation wird von UR-Walking verwendet, um Echtzeit-Updates der aktuellen Nutzerposition zu erhalten. Die Ergebnisse sind interessant, da andere Klassifikatoren konventionell als besser angesehen werden und das Unterscheiden von Räumen in unterschiedlichen Stockwerken in der Literatur als schwer angesehen wird, der Klassifikator aber in beiderlei Hinsicht gute Ergebnisse erzielt. Allerdings hat das System im aktuellen Zustand einige Mängel, wie fehlende Unterstützung für SSL-verschlüsselte Kommunikation, und es ist nicht erwiesen, dass das System von nur einem Gebäude auf die gesamte Universität skalierbar ist.

Abstract

For this thesis, a software suite for measuring WiFi data in rooms at the University of Regensburg was created, classifying said data to their origin room and making this classifier available for the Android app "UR-Walking," the university's pedestrian indoor navigation solution. The measuring of WiFi data was done with an Android app written in Kotlin, while the classification and server functionality was implemented in Python using the sklearn library. Lastly, the UR-Walking app was changed to communicate with the Python server using JSON to receive real time updates on user positioning.

Classifiers were fitted on a training set from a measurement of the "Vielberth-Gebäude," a building on campus, and tested on a separate measurement of the same building on a different day for confirmation of classifier performance. The best performing classifier, with a mean accuracy of 96.9% across all rooms and measurements, was random forest, and was persisted to later be loaded by the server. While the system has issues such as not supporting SSL encryption, and its scalability to the entire campus (rather than a single building) is not confirmed, its results follow up on literature mentions of the potential of ensemble algorithms, and the classifier performs well for determining which floor of the building the user is on, which is often a problem discussed in the field.

Aufgabenstellung

Ortsbestimmung innerhalb von Gebäuden für Fußgängernavigation ist schwer, da GPS-Signale zu ungenau sind. Es gibt Ansätze, Ortsbestimmung mit WLAN-Signalen durchzuführen, was zur Verbesserung von aktuellen Navigationsverfahren, die in der universitären UR-Walking-App verwendet werden, verwendet werden kann.

Diese Arbeit soll eine Software erstellen, die die Messungen einer Handy-WLAN-Antenne einem Ort in einem Universitätsgebäude möglichst akkurat zuweisen kann. Dabei sollen zuerst WLAN-Daten gemessen und annotiert werden, dann soll auf Basis dessen ein Klassifikator erstellt werden, dessen Ergebnisse dann in der UR-Walking-App verwendbar gemacht werden sollen.

Schritt 1: Eine App, die es erlaubt, eine Position auf einer Karte der Universität auszuwählen, WLAN-Daten zu messen, und diese, zusammen mit Meta-Daten über Zeit, Mess-Gruppe, Standort und Gerät, abzuspeichern.

Schritt 2: Python-Skripte, die die eben erstellten Daten auslesen, und einen Klassifikator erstellen, der WLAN- und Meta-Daten verwendet, um dem Standort vorherzusagen. Dieser Klassifikator wird dann abgespeichert.

Schritt 3: Ein Server-Interface, dem man WLAN- und Meta-Daten senden kann, und es antwortet mit einer Standort-Vorhersage auf Basis des vorher erstellten Klassifikators. In UR-Walking wird ein Sub-Prozess erstellt, der diese Daten an den Server schickt und die Standort-Vorhersage entgegennimmt.

1 Introduction

Determining user positioning for pedestrian indoor navigation is a challenging task that requires the positioning system to determine the user's starting location as well as track their movement and direction over time. It is made even more difficult by the technical and financial restrictions placed on it: the only technology that a user can be expected to carry is a smartphone, which has a restricted array of sensors for navigation. GPS is not a suitable solution as its accuracy inside a building is insufficient for navigation tasks if it is able to determine a location at all (He & Chan, 2016, p. 466). Inertial sensors can approximate relative changes in direction and position, but they can neither determine the absolute starting point nor direction of a navigation; determining position via Bluetooth would require the installment of additional hardware at the navigation site, a cost factor. WiFi exists in most modern buildings that would require navigation, and smartphones have built-in hardware to measure and use WiFi connection, WiFi is therefore an integral element of pedestrian indoor navigation. (He & Chan, 2016)

The University of Regensburg developed a software for pedestrian navigation called "UR-Walking." It is available as web app and Android app (https://urwalking.ur.de/navi/?lang=en). The mobile app currently uses inertial sensors for navigation; as mentioned above, this comes with the problem of being unable to determine the starting point of a navigation, which is why in this thesis, a software suite for determining user location based on WiFi signals received by the user's smartphone was developed so that later on the two approaches can be combined into a hybrid system utilizing the two methods' complementary advantages.

If the UR-Walking app was able to accurately assess user positioning and movement, it would allow for future research on human interaction with pedestrian navigation systems and would greatly increase the quality of naturalistic log data that the system currently generates.

The software suite developed for this thesis consists of an Android app, written in Kotlin, which allows the annotated (by room) measurement and recording of WiFi data; this process is called fingerprinting and will be discussed more closely in *2 Related Work*. Using Python and the sklearn library, based on data recorded using that

app in the "Vielberth-Gebäude," a building on the campus of the University of Regensburg, a set of classifiers was trained and compared to pick the best performing one. The classifiers' generalizability was assessed by testing them on a second dataset taken in the same building on a different day, to eliminate potential for order effects, overfitting, or other unforeseen phenomena that might falsify the performance metric of the classifier. The best performing classifier, random forest, was able to predict the user location with a mean accuracy of 96.9% per room and measurement. This classifier was then used as the backbone of a server interface that allows the retrieval of location predictions based on WiFi data sent by the UR-Walking Android app, which is now able to utilize the accurate location prediction algorithmically.

In this document you are presented with an overview of the related work in the field of WiFi based indoor navigation, the requirements of the software suite, its implementation as well as the reasons for implementation choices, the results of the implementation and an evaluation of classifier performance, conclusions to be drawn from those results, as well as the approach's limitations, and emergent questions for future research and projects.

2 Related Work

In the Android API, apps are given various information upon completing a scan of available WiFi connections ("Android API ScanResult Documentation"). The most relevant information for determining user location is BSSID, also called the MAC address, that uniquely identifies every network interface controller, of which routers – often called access points (AP) (He & Chan, 2016, p. 466) – have many ("MAC address Wikipedia," 2020b March 2). The second important information is the received signal strength indicator (RSSI) (He & Chan, 2016, p. 466), measured in dBm.

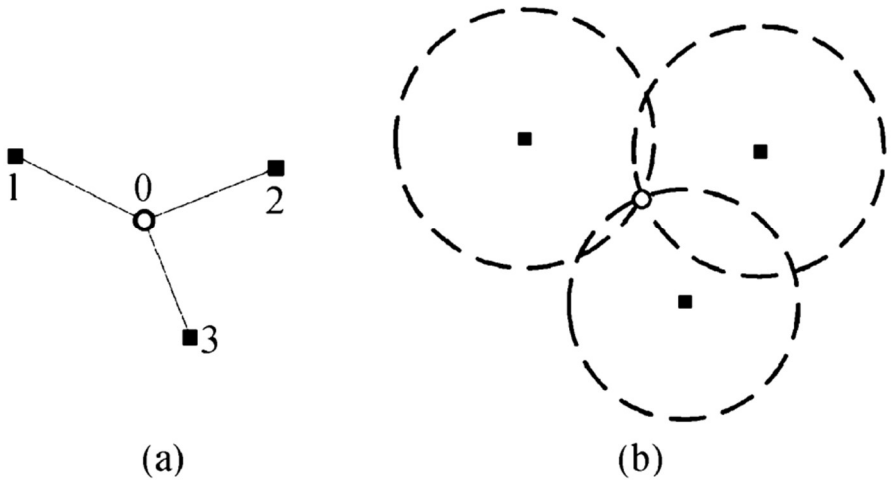

Figure 1 "Trilateration. (a) Measuring distance to 3 reference nodes. (b) Ranging circles." (Yunhao Liu, Zheng Yang, Xiaoping Wang, Lirong Jian, 2010 March, p. 276)

One approach of calculating the position of a device is called multilateration. With this method, at least 3 non-linearly aligned APs, with known real-world location, are needed and the RSSI of each individual signal is used to calculate the real-world distance from the device to the AP. Since the APs are all non-identical and non-linearly aligned, as well as their real world positions known, this data is enough to perform trilateration (or if more APs are available, multilateration) to determine the device location (Yunhao Liu, Zheng Yang, Xiaoping Wang, Lirong Jian, 2010 March, p. 277), see Figure 1. However, the calculation from RSSI to real world distance to an AP becomes unreliable once there are physical obstacles between the device and the AP, such as doors, walls, or people (He & Chan, 2016, p. 466), therefore other strategies need to be considered.

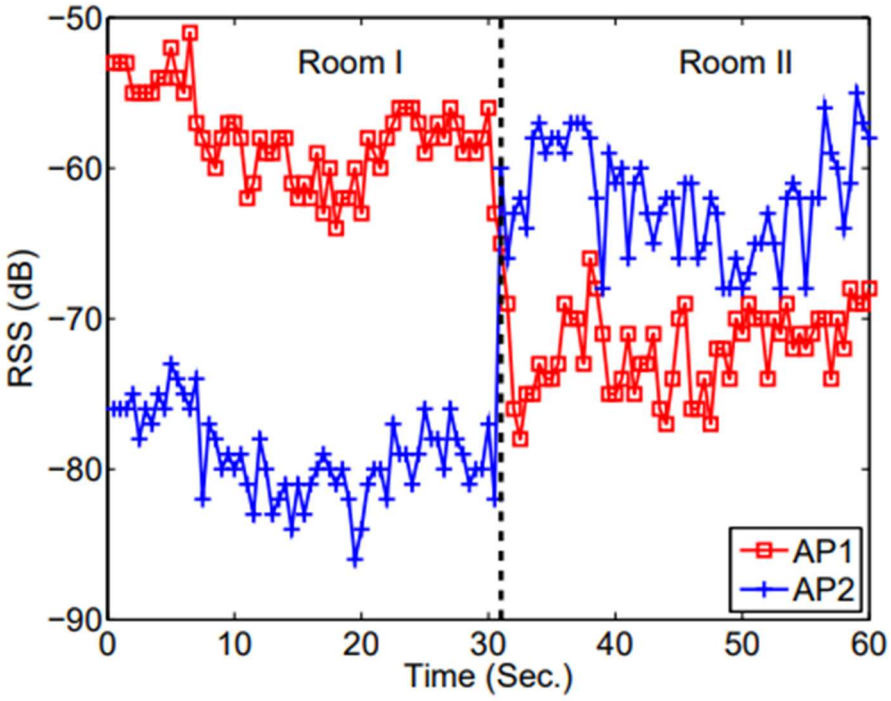

Figure 2 "Abrupt signal changes through a wall. AP1 is deployed in Room I and AP2 in an adjacent Room II." (C. Wu, Yang, Liu, & Xi, 2012, p. 841)

One such strategy is WiFi fingerprinting, which utilizes the phenomenon that most rooms have a unique signature of received signal strength (RSS) from every AP due to the dampening of RSS by walls, as discussed in the multilateration approaches, a *fingerprint*. A large amount of room-annotated data is then recorded so that a new, un-annotated vector of RSSIs can be matched based on the previously saved fingerprints (C. Wu et al., 2012, p. 840). A simplified use case of WiFi fingerprinting is shown in Figure 2: Room I can be identified by its fingerprint of AP1 having a higher RSSI than AP2, and Room II can be identified by its fingerprint of AP1 having a lower RSSI than AP2; so if now a vector of "AP1: -55 dB, AP2: -90dB" needed to be matched to a location, a room, then the prediction would be Room I, as its existing data points are very similar to that vector. This basic principle can then be scaled to many rooms and many access points, each with a mostly identifiable fingerprint. A big weakness of this approach is the rate at which a smartphone receives scans of nearby WiFi signals: in practice it takes multiple seconds until the system gets a new reading, meaning that the predicted position is usually lagging behind the user's actual location.

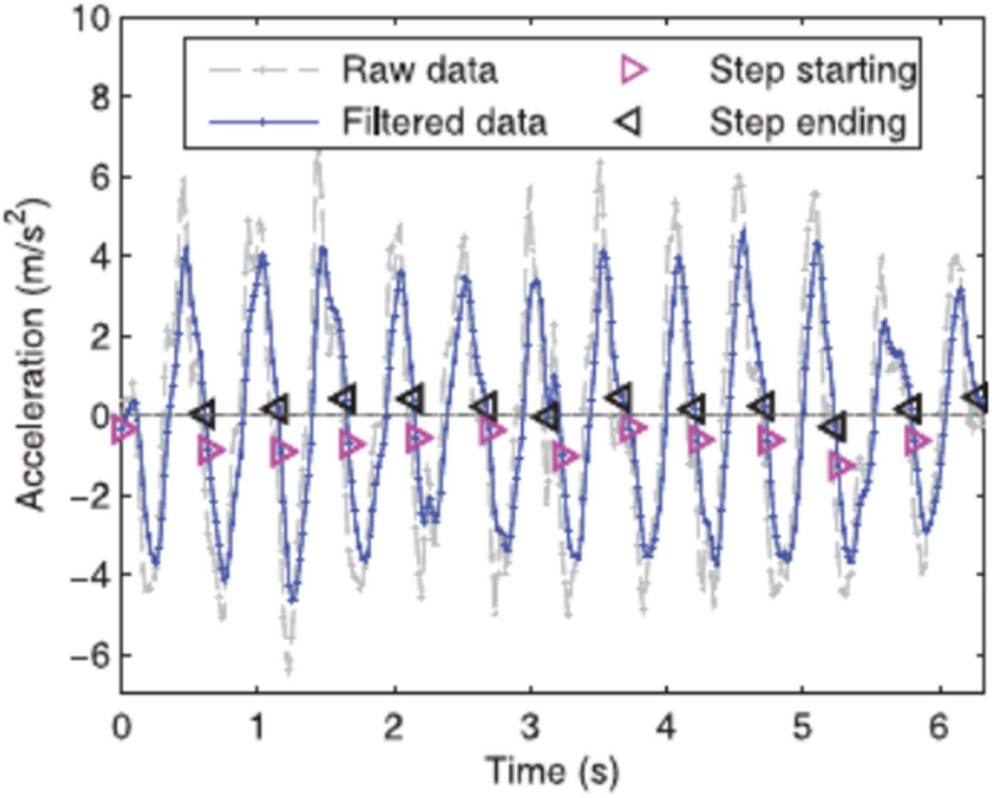

Figure 3 „Acceleration magnitudes during a sample walking demonstrating the repetitive patterns of walking." (Yang et al., 2015, 54:8)

Another relevant approach is the use of inertial sensors, which utilizes accelerometers, sensors measuring acceleration ("Accelerometer Wikipedia," 2020a March 2), gyroscopes, sensors measuring "orientation in principle of angular momentum" (Yang et al., 2015, 54:5; "Gyroscope Wikipedia," 2020 March 1), and magnetometers, sensors measuring "the strength and the direction of magnetic fields" (Yang et al., 2015, 54:5; "Magnetometer Wikipedia," 2020 February 2). "[G]yroscopes are often exploited in conjunction with accelerometers to derive robust direction information (e.g., X-, Y-, Z-axis acceleration with the extent and rate of rotation in roll, pitch, and yaw" (Yang et al., 2015, 54:5; "Gyroscope Wikipedia," 2020 March 1). This means these sensors can be used to determine user speed and to detect when a user turns and how much they turn relative to their previous direction of movement. However, they are prone to increasing error in accuracy due to various factors such as: sensor noise, position of the device on the user's person (a smartphone in hand is moved very differently to one in the backpack or in a pocket), and humans having a vast number of different ways of walking; these imprecisions then lead to a quadratic error as acceleration needs to be integrated twice to derive distance traveled (first it is

integrated from acceleration to velocity, and then from velocity to distance traveled) (Yang et al., 2015, 54:8). The accuracy of these sensors can be increased by detecting individual steps (see Figure 3,) or the user's mode of movement (i.e. running or walking) (Yang et al., 2015, 54:7). The main weakness of this approach is that starting position cannot be determined by the data given and absolute walking direction is being determined by the magnetometer, which carries a risk of giving off false readings when near electromagnetic fields created by electric devices.

Hybrid approaches (also called fusion approaches) for combining WiFi RRSI measurements and inertial sensors have been proposed and tested successfully (Leppäkoski, Collin, & Takala, 2013). The two approaches complement each other very well as WiFi fingerprinting can determine absolute starting position and rough navigation direction (by comparing a series of position predictions) while inertial sensors can accurately track movement and turning; WiFi fingerprinting's weakness of lagging behind the user's actual position is mitigated by the strength of the inertial sensor approach of tracking movement in real time, and inertial sensors' tendency to become inaccurate over extended navigation can be mitigated by position predictions from WiFi fingerprinting. For the fusion of these two approaches the Kalman filter is used (Leppäkoski et al., 2013, p. 287), which works similar to a hidden Markov model, except that its hidden values are not discrete but instead continuous (Roweis & Ghahramani, 1999). The Kalman filter, or variations of it such as the Extended Kalman filter, adjusts for sensor noise and other forms of random or uncontrolled error continuously, resulting in systems as accurate as 1.3 m (Leppäkoski et al., 2013, p. 287). It is the objective of this thesis to provide a WiFi positioning system that can be a used as a basis for later implementation of such a fusion approach.

In Bozkurt, Elibol, Gunal, and Yayan (2015) the classification of floors for multistory buildings is given special attention and it is shown that some algorithms do not perform well in this context and the overall accuracy is relatively low. Furthermore, they show performance of algorithms that specifically only predict floor level, not rooms on different floors which would be a more challenging task. Because of this the performance in multistory classification will be given special note when analyzing classifiers.

3 Project Requirements

The overarching goal of this set of software is to provide a location prediction on the campus of the University of Regensburg to the UR-Walking Android app. This goal has various milestones, or sub-goals, with requirements of their own, which are presented in this chapter in chronological order of use.

One important definition for this project is the meaning of "location". UR-Walking uses a graph structure for navigation, where each intersection (such as a door, a T-intersection, or a point in a hallway that allows you to turn and walk into a room) is a node. Each node is connected to other nodes, which can be reached from the initial node directly on foot without passing another node, with edges. There are also special, named nodes, which represent rooms and clearly identifiable places, called landmarks; rooms might include "H 26" ('Lecture hall 26') while landmarks might include "the wooden pin board". The combination of landmark- and room-nodes was deemed sufficient as a prediction target because the unnamed intersection nodes become too frequent and tightly packed in areas with a lot of doors since every door but also every point where someone walking out of that door would enter the normal pathway of the hallways are considered intersection nodes; this granularity would not be helpful for location prediction. Therefore, if the word "location" or "position" is used in the description of the software project, it henceforth refers to "a named node in the UR-Walking graph model" and most commonly describes a single room.

3.1 Fingerprinting

Firstly, a piece of software to scan available WiFi networks and save them, annotated with the location they were recorded at, is required, as a basis for the fingerprinting approach presented in *2 Related Work*. Since the users, that will eventually use UR-Walking to navigate, will use Android smartphones, this piece of software should be an Android app as well, rather than a software monitoring sensors connected to a computer such as a Raspberry Pi; this will help to have more similar RSS readings compared to a sensor array with possibly different quality to that of the average smartphone.

3.2 Classification

The classification part of the software must be able to predict the user position as accurately as possible based on measurements taken using the app created in the previous step. The classification can be done offline and does not need to be part of an app, as long as the final predictions are made available to UR-Walking in some form.

Since it is useful to track success during development time to judge if one is getting closer to a solution of the task or not, it is useful to define what exactly "accurately predicting" means, as that is the thing that needs to be optimized. A normal metric of being accurate in predictions is accuracy, which for multiclass problems (which this is as each predictable node is a class for classification) is calculated as the number of correct predictions on a test set divided by the total number of predictions made on that test set, in other words it is "the fraction of correctly classified samples" ("sklearn documentation accuracy score"). However, this is a flawed metric for the problem at hand: since rooms have different sizes, there will be more measurements for larger rooms, the classes for classification (the individual IDs of the nodes) will not all be equally distributed in the dataset, the dataset will be *imbalanced*. This results in the accuracy score yielding a higher result if common classes are being predicted better than uncommon classes. For example: given *Big Room* with 90 measurements and *Small Room* with 10 measurements, a classifier that always predicts *Big Room* without even looking at the data would predict all 90 measurements from *Big Room* correctly and all 10 from *Small Room* incorrectly, which would be described by an accuracy score of 90/(90+10) = 90%. There is, however, no evidence that users are going to try to predict their location nine times more often in *Big Room* than in *Small Room*; it is unlikely that the value of 90% accurately describes user experience, therefore a different metric must be found.

A better metric would answer the question "If the classifier predicts my location in any, equally randomly chosen room on campus, what is the chance, that this prediction is correct?", as it will match user experience more closely because users might navigate anywhere on campus. Balanced accuracy score answers this question. It can be defined as the arithmetic mean of all accuracy scores calculated for each class individually, or in other words "[i]t is defined as the average of recall obtained on each class" ("sklearn documentation balanced accuracy score"). In the earlier example this would mean that

first the accuracy for each class is calculated, which is 90/90 = 100% for *Big Room* and 0/10 = 0% for *Small Room*; then the arithmetic mean of those to values is calculated yielding (100% + 0%)/2 = 50%. This is a significantly more representative scoring method for the problem and was therefore used during classifier development as well as evaluation.

3.3 Prediction in UR-Walking

In the last step, the UR-Walking Android app must be made able to use the previously created prediction algorithm, and it must be able to do that in a way appropriate for smartphones – the platform the UR-Walking Android app uses. This means the prediction process cannot be too computationally expensive or require a lot of memory or disk space. It can, however, utilize mobile internet and WiFi connection as both are available on campus (WiFi especially indoors and mobile internet especially outdoors). In any case when the implemented algorithm comes to a prediction, it must make that prediction known to the rest of the program in form of an event, or something similar like some form of observer-observable (or listener) pattern.

4 Project Implementation

The project was implemented using various technologies: the measuring of WiFi fingerprints is done using an Android App written in Kotlin, which saves measured data as .CSV or .TXT files; this data is then unified into a big .CSV table by a Python script, which then also creates, optimizes, and tests classification algorithms, the best of which is saved; the saved classifier is then loaded by a Python web server, which awaits UR-Walking's WiFi scan in JSON and upon receiving said scan, classifies the location based on the measured WiFi data and answers with predictions to the requester, also using JSON. More detailed explanations of each implementation follow:

4.1 Fingerprinting

The Android app, for measuring WiFi data and saving it in an annotated manner for nodes of the UR-Walking graph structure, was implemented in Kotlin using Android Studio. It was based on an already existing app with the purpose of taking video footage of landmarks on campus. This was useful as essential functionality like

selecting and displaying a building's floor plan and displaying UR-Walking's named nodes on that floor plan was already implemented. All that needed to be changed was that instead of tracking data and recording camera footage of landmarks, it needed to track WiFi scans for each location, so it saves the data of the `ScanResult` object that is yielded with every WiFi scan and associates it with the current timestamp, and the currently selected node, and orientation (explained below). Since GPS readings, atmospheric pressure, the current UNIX epoch time, and the phone model being used were trivial to track, their tracking was also implemented in case that data can be used later. The error prone and memory and disk space intensive recording of camera footage was removed.

The original app had the concept, that a measurement had to be taken either towards or from a cardinal direction (North, North-East, South, etc.) in relation to a node, deeply ingrained into its `ViewModel`. Removal of that underlying logic was deemed too work intensive to remove since the upside of doing so would be saving a single tap when taking a measurement, which is insignificant to the time a measurement takes.

After the app was completed a first set of measurements was taken on the ground floor of the "Vielberth-Gebäude" ('Vielberth building') on campus to test the functionality of the app and to have data to develop the classification algorithms with.

4.2 Classification

In this section, the preprocessing of data, training, optimization, and functionality of classifiers, as well as the serialization of important artefacts is described.

4.2.1 Transforming the Measurement Data

Classification was implemented using Python script. First, the data created by the app from step *4.1* was transferred into the directory of the Python project. Static values useful to multiple scripts are saved in `SETTINGS.py`

`transformMeasurementsToUnifiedDataset.py` loads and cleans the data and saves it as a single .CSV table.

It iterates over every measurement folder and loads all data that was saved with the Android `ScanResult`: timestamp of measurement, BSSID, RSSI, and SSID (the human readable name of the WiFi network). It then discards all rows that contain a

BSSID that is not from a university WiFi router (such as someone's smartphone hotspot that might have been picked up during measuring), as well as all rows that contain the SSID "conference.uni-regensburg.de", which is an additional network the university only turns on when a conference takes place on campus. These removals are important because the networks that they record are not ever-present; someone using the smartphone as a hotspot is likely to leave in the evening and unlikely to do it every day for years, and the university doesn't always host conferences; and if a not-ever-present network is used as part of a room's WiFi fingerprint, then recognizing that room when the network is gone becomes harder and leads to unnecessary errors in prediction.

Next pressure data and its associated timestamp is loaded into another pandas data frame.

It then turns the WiFi data frame, which is a long list of individual, received BSSIDs, lots of them at identical times, into a data frame that maps a timestamp to the RSSI for every recorded BSSID.

This data was then appended by the measured atmospheric pressure closest in time to the measured WiFi data, as there was not usually one identical in time because the WiFi scans and the pressure sensor do not operate at the same rate of measurement.

The node-ID (a unique identifier for every UR-Walking node) is appended to the data. The node-ID is known because every directory in the iteration of the program identifies the node it was saved for.

All data frames of this sort are then iteratively combined with each other, leading to a large table in which a node-ID is mapped to a timestamp, a pressure value, and the RSSI for every BSSID that has been measured over the course of the entire measurement set, for each `ScanResult` Android yielded for the node. Many of the RSSI cells contain no information at all as not all BSSIDs are in range for every measurement.

This data frame is then stored as `combined_data.csv`.

4.2.2 Readying the Transformed Data for Classification

`combined_data.csv` is then loaded into a data frame by `createClassificationReport.py`, which fulfills multiple purposes:

preprocessing data, optimizing classifiers, testing classifiers, and persisting the best classifier.

The first obstacle in loading `combined_data.csv` is the question of how to handle the missing values for RSSIs as there must be a value in those cells otherwise classification algorithms will usually not be able to use the data. Instead of leaving the cell empty, the value -100 is used, which stands for an RSSI of -100 dBm and is lower than the lowest signal strength received during measurement, but it is still in the same order of magnitude as the other RSSIs (max: -36; min: -96).

Next the data frame is scaled using sklearn's `MinMaxScaler`, which "scales and translates each feature individually such that it is in the given range on the training set, e.g. between zero and one" ("sklearn documentation MinMaxScaler"). Scaling was chosen so no one feature that has s different order of magnitude would dominate estimators; e.g. RSSIs have values around -60 while pressure has values that exceed 900; an estimator is likely to attribute greater importance to values of a greater order of magnitude and large numbers might lead to excessive RAM use during the internal calculations of classifiers. The `MinMaxScaler` specifically was chosen for multiple reasons: some classifiers, such as Multinomial Naïve Bayes, do not work with negative feature values ("sklearn documentation MultinomialNB"); the mapping of values onto the continuous range from 0 to 1 is semantically sensible since -100 dBm is mapped to 0, which represents not receiving at all, and the maximum dBm measured is mapped to 1, which represents as much as possible, this also comes with the effect that now the larger numbers are no longer the less important ones; `MinMaxScaler`, `StandardScaler`, `RobustScaler`, and `MaxAbsScaler` were all tested during development time and `MinMaxScaler` lead to the highest balanced accuracy.

4.2.3 Train-, Validation, and Test-Sets

In machine learning literature, three descriptions of distinct datasets are often used. The most intuitive one is the training set, which is the data a classifier is trained on.

There are two more sets of data: one set which is used to assess classifier performance at development time, and one set to assess a final classifier's real-world generalizability before use. These are called validation-set and test-set, however, there

are different practices on which set is given which name. (“Training, Validation, and Test sets Wikipedia,” 2020 February 19)

For the purposes of this document, the set used during development time will be called validation set, and the set used to assess a final version of a classifier will be called test set.

4.2.4 K-Fold Cross-Validation

During development time, classifiers were assessed using balanced accuracy during k-fold cross-validation, which describes the separation of a training set into k equally sized parts (where k is a positive integer greater 1). Once separated, one part is used as validation set while all other parts are used as training set. The classifier is trained, and its performance recorded, then the next part is used as a validation set, and the former validation set becomes part of the training set and the classifier is trained and tested anew. This process is repeated k times, then the mean of all performances is calculated and reported as the performance in k-fold cross validation.

The advantage of this approach is that it prevents overfitting and manages to get more use out of the training data compared to a normal split in a single training- and a single validation set (“Cross-validation: evaluating estimator performance,” 2020).

A detail that can be added to k-fold cross-validation is stratifying. When a data set is partitioned in stratified k-fold, it means that in addition to the k parts being of equal size, the proportion of classes to each other is also equal. This has the big advantage that is will not create unfair or unsolvable arrangements where one class is only in the validation set but not the training set and so doesn’t arbitrarily punish classifier inaccuracy simply because of an unfavorable data split (“Cross-validation: evaluating estimator performance,” 2020).

During development time of this software project 5-fold stratified cross-validation was used.

4.2.5 Oversampling Training Data

As discussed in *3.2 Classification*, the measured data set has to be expected to be imbalanced. Some classifiers generate biases towards overpopulated classes, which hurts their performance in some real-world applications and their balanced accuracy score.

To mitigate this problem, a process called oversampling (or upsampling) can be used. When oversampling, a training set with imbalanced classes is transformed into a training set with equally distributed classes. This is achieved by checking for every class if it is represented in the set as much as the most represented class; if yes, the next class is checked; if no, a random data piece of that class is copied and added to the training set, then the class balance is checked again ("Random Oversampling and Undersampling for Imbalanced Classification").

During development of this software, imblearn's `Pipeline` was used, to be able to specifically pick the sampling method for each classifier individually as some classifiers might perform worse with oversampling. Each classifier was tested with and without oversampling using non-optimized hyperparameters and then set up in a way to use the more favorable sampling method. The reason this was done before optimization is that it seems unlikely that the behavior will change with different hyperparameters and that it would double the computation time of hyperparameter search to perform a search of equal quality.

4.2.6 Hyperparameter optimization

Machine learning classifiers have a variety of parameters, which have a strong influence on classifier performance, which can be set prior to training and prediction. Picking the right ones for the problem at hand is therefore crucial to building a well performing classifier.

There are two main strategies for hyperparameter optimization: grid search and random search ("sklearn documentation Tuning the hyper-parameters of an estimator").

In grid search, a set of sensible values is listed for every parameter. Then the search is started, and every possible combination of these values is tested. At the end of the process the performance of the different combinations is reported, usually ordered from best to worst. This has the upside of providing a definitive answer of the best configuration for the parameters and values given, but it comes at the downside of high computation time, especially since the computation time unnecessarily increases multiplicatively if value sets are given for parameters that do not influence classifier performance ("sklearn documentation GridSearchCV").

The alternative is random search, where for every parameter either a set of parameters, or a random function returning a random parameter in a sensible range is provided. The search then combines all parameters randomly to train and test a predefined number of configurations. This comes with the upside that non-relevant parameters do not slow down search, as they are simply picked and then do not influence the result, but it comes with the downside of not definitively returning the best combination of parameters ("sklearn documentation RandomizedSearchCV").

`createClassificationReport.py` uses randomized search in 5-fold stratified cross-validation to optimize hyperparameters with a number of iterations handpicked for every classifier individually based on the number of different sensible hyperparameters.

4.2.7 Classifiers

In this section you are provided with information about every classifier that was tested for the location prediction. Every classifier used was from the sklearn Python library. Each subsection starts with a short description of the algorithm itself and the reason for its consideration, followed by a statement if oversampling of training data was used, and which hyperparameters were tuned during random search.

4.2.7.1 K-Nearest Neighbors

> "K-Nearest Neighbor (K-NN) [(D. W. Aha, D. Kibler, & M. K. Albert, 2011, pp. 37–66)] classifier is also known as a distance based classifier that classifies instances based on their similarity. It is one of the most popular algorithms in machine learning and is a type of Lazy learning in which the function is only approximated locally and all computation is delayed until classification. The unknown tuple in K-NN is assigned to most common class among its K-nearest neighbors. When K = 1, the unknown tuple is assigned the class of the training tuple that is closest in the pattern space [(C. Shah & A. G. Jivani, 2013)]." (Bozkurt et al., 2015, p. 4)

It was used because it was the best performing algorithm in Bozkurt et al. (2015) and is commonly used as a solution for this type of problem (He & Chan, 2016, p. 468).

Oversampling was not used, as it defeats the purpose of testing the k-nearest neighbors since the duplication of data points would just make them count twice.

Hyperparameters tuned were k, whether to weigh data points closer to the point to be classified as more important for classification, and whether to use Euclidian or Manhattan distance.

4.2.7.2 Feedforward Neutral Network / Multi-Layer Perceptron

> "A layered feedforward network is a network which has a distinct set of input units onto which values are clamped. These values are then passed through a set of weights to produce the inputs to the next layer of "hidden" or internal units. These units modify their input using a nonlinear function (usually sigmoid in shape) to produce their outputs. As many layers as desired of this form can be stacked, and the units of the final layer become the outputs of the network." (Sanger, p. 10)

A feedforward neural network based classifier, also known as Multi-Layer Perceptron classifier ("sklearn documentation MLPClassifier"), was used due to International Conference Automation (2017) showing its effectiveness in fingerprinting problems.

Oversampling was used because Johnson and Khoshgoftaar (2019, p. 21) reported "In almost all experiments, over-sampling displayed the best performance, and never showed a decrease in performance when compared to the baseline."

Hyperparameters tuned were: the size and number of hidden layers, the activation function, the solving algorithm, the L2 penalty, whether or not and how to change the learning rate across iterations, the initial learning rate, the maximum number of iterations, the "[m]omentum for gradient descent update" ("sklearn documentation MLPClassifier") (only for sgd solver), whether to use Nesterov's momentum (only for sgd solver), the "[e]xponential decay rate for estimates of first moment vector in adam" ("sklearn documentation MLPClassifier") (only for adam solver), and the "[e]xponential decay rate for estimates of second moment vector in adam" ("sklearn documentation MLPClassifier") (only for adam solver).

4.2.7.3 Random Forest

> "Random forest is a combination of tree predictors where each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest , where, in each tree, If the number of cases in the training set is N, a sample of N cases is taken at random from the original data. This sample will be the training set for growing the tree, then, if there are M input variables, a number m << M is specified such that at each node, m variables are selected at random out of the M input variables, then,

> the best split on these m is used to split the node. The value of m is held constant during the forest growing. The random forests algorithm can be used in supervised and unsupervised learning techniques. In supervised technique, it can be used in classification and regression. With regard to classification, the random forests algorithm is consisting of a collection of multiple tree classifiers {t(x, Θk), k = 1, ...} where the {Θk} are independent identically distributed random vectors and each tree gives a unit vote for the most popular class at a determined input x. To classify an observation, it is passed through each decision tree to calculate the suggested class for this tree, giving a set of votes to all possible classes, and finally, the estimated class of this observation will be chosen as the class having maximum number of votes." (Elbasiony & Gomaa, 2014, p. 2)

Random forest was selected because Elbasiony and Gomaa (2014) used it in their experiment to achieve a mean localization error of ± 36 cm.

Oversampling was used, as it showed better performance in 5-fold cross-validation during development time.

Hyperparameters tuned were the number of estimators (trees), "[t]he function to measure the quality of a split," ("sklearn documentation RandomForestClassifier") the function used to determine "[t]he number of features to consider when looking for the best split," ("sklearn documentation RandomForestClassifier") and "[w]eights associated with classes." ("sklearn documentation RandomForestClassifier")

4.2.7.4 Multinomial Naïve Bayes

> "A Naïve Bayes [(John & Langley, 2013)] classifier based on Bayes Theorem is a supervised learning algorithm [(Chug & Dhall, 2013)]. It is robust to noisy data, easy to build, shows high accuracy and speed when applied to large databases and performs more complicated classification models. Hence, it is widely used in classification tasks. It calculates probability of each attribute in the data assuming that they are equally important and independent of each other." (Bozkurt et al., 2015, p. 3)

Furthermore, "The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work." ("sklearn documentation MultinomialNB") Since min-max-scaled RSSIs are similar in nature to tf-idf indices and Naïve Bayes has shown promise in works such as Bozkurt et al. (2015), it was chosen as a candidate classifier here.

Oversampling was used as it showed better performance in 5-fold cross-validation during development time.

Hyperparameters tuned were the "[a]dditive (Laplace/Lidstone) smoothing parameter," ("sklearn documentation MultinomialNB") "whether to learn class prior probabilities or not," ("sklearn documentation MultinomialNB") and whether to learn the class probability from the training data or use an equal probability for all classes.

4.2.7.5 C-Support Vector Classification

A support vector classifier creates a hyperplane that divides two classes. The support vectors shape that hyperplane (Chang & Lin, 2011, 27:3). In multiclass problems either "one vs one" or "one vs rest" approaches are used to divide the data ("sklearn documentation SVC").

This classifier was used because it has been proven to work within this problem space before (C.-L. Wu, Fu, & Lian, 2004).

Oversampling was not used as it performed worse than not oversampling in 5-fold cross-validation at development time.

Hyperparameters tuned were the regularization parameter, the kernel function, the "[d]egree of the polynomial kernel function" ("sklearn documentation SVC") (only for the polynomial kernel function), the "[k]ernel coefficient for 'rbf', 'poly' and 'sigmoid'," ("sklearn documentation SVC") the "[i]ndependent term in kernel function. It is only significant in 'poly' and 'sigmoid'," ("sklearn documentation SVC") "[w]hether to use the shrinking heuristic." ("sklearn documentation SVC")

4.2.7.6 AdaBoost with Decision Tree

> "AdaBoost (Adaptive Boosting) [(Freund & Schapire, 1996, pp. 148–156)] is an ensemble learning algorithm. Generally, it can be used with weak machine learning algorithms to improve their performance. It is simple to implement, fast, and less susceptible to overfitting. It improves unstable classification algorithms such as J48, DecisionStump, etc. The idea behind this algorithm is to obtain highly accurate classifier by combining many weak classifiers. It works by repeatedly running a given weak learning algorithm on various distributions over the training data, and then combining the classifiers produced by the weak learner into a single composite classifier [(Shams & Mercer, 2013, pp. 657–666)]. The classifiers in the ensemble are added one at a time so that each subsequent classifier is trained on data which have been difficult for the previous ensemble members. Weights are set to instances in the data set, in a rule that the

> instances that are difficult to classify get more weight. This rule drives subsequent classifiers to focus on them [(Sharif, Kuncheva, & Mansoor, 2010, pp. 1168–1172)]." (Bozkurt et al., 2015, p. 4)

It was used because Bozkurt et al. (2015, p. 1) reported "ensemble algorithms such as AdaBoost […] are applied to improve the decision tree classifier performance nearly same [sic] as k-NN that is resulted as the best classifier for indoor positioning."

Oversampling was used as it performed better in 5-fold cross-validation during development time.

Hyperparameters tuned were the number of estimators, the learning rate, the algorithm used, and the maximum depth of the decision tree classifier that is being boosted.

4.2.7.7 Gradient Boosting Classification

> "Gradient Boosting for classification […] builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage *n_classes_* regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced." ("sklearn documentation GradientBoostingClassifier")

It was used because it is an ensemble learning algorithm like AdaBoost and Bagging, which had been shown as promising in Bozkurt et al. (2015, p. 1), but Gradient Boosting itself was not used, so its evaluation here can add to the literature on ensemble learning algorithm performance for WiFi fingerprinting.

Oversampling was used as it performed better in 5-fold cross-validation during development time.

Hyperparameters tuned were "[t]he number of boosting stages to perform," ("sklearn documentation GradientBoostingClassifier") "maximum depth of the individual regression estimators," ("sklearn documentation GradientBoostingClassifier") "[t]he fraction of samples to be used for fitting the individual base learners," ("sklearn documentation GradientBoostingClassifier") the learning rate, and the function to determine "the number of features to consider when looking for the best split." ("sklearn documentation GradientBoostingClassifier")

4.2.7.8 Bagging Classification with Decision Tree

> "Bagging [35] builds bags of data of the same size of the original data set by applying randomly selecting different subsets of the training data with many examples appearing multiple times. This process named as bootstrap replicate of the training data. The idea behind this technique is to build various classifiers by using these subsets. Each subset is used to train one individual classifier. This ensemble approach uses number of classifier as a priori [35]." (Bozkurt et al., 2015, p. 4)

It was used because Bozkurt et al. (2015, p. 1) reported "ensemble algorithms such as […] Bagging are applied to improve the decision tree classifier performance nearly same [sic] as k-NN that is resulted as the best classifier for indoor positioning."

Oversampling was used as it performed better in 5-fold cross-validation during development time.

Hyperparameters tuned were "[t]he number of base estimators in the ensemble," ("sklearn documentation BaggingClassifier") "[w]hether to use out-of-bag samples to estimate the generalization error," ("sklearn documentation BaggingClassifier") and the maximum depth of the decision tree classifier being used.

4.2.8 Creating the Test Set

To evaluate the models as described in *5.2 Classification*, the data measured for the test set was transformed and loaded analogous to the data for the training set in *4.2.1 Transforming the Measurement Data* and *4.2.2 Readying the Transformed Data for Classification*, with the exception that it was scaled using the same scaler that was used to scale the training set.

4.2.9 Serialization

The random search was conducted, then the performance of all optimized classifiers was evaluated on the test set as described in *5.2 Classification*. The best performing classifier was then serialized, alongside the `MinMaxScaler`, using the joblib module. They were saved named `classifier.joblib` and `scaler.joblib`, respectively.

Note that "pickle (and joblib by extension), has some issues regarding maintainability and security. Because of this[:] [1] Never unpickle untrusted data as it could lead to malicious code being executed upon loading. [2] While models saved using one version of scikit-learn might load in other versions, this is entirely

unsupported and inadvisable. It should also be kept in mind that operations performed on such data could give different and unexpected results." ("sklearn documentation Model persistence") Because of this `createClassificationReport.py` should be run at least once after installation of the software suite before starting the server software described in *4.3 Prediction in UR-Walking*.

Furthermore, the column names of the data frame used to train the classifier is saved in `feature_vector_head.csv` so that in the future values can be given to the classifier in the correct order.

4.3 Prediction in UR-Walking

Since classification is often computationally expensive and executing algorithms generated in Python within the UR-Walking Android app is challenging, instead a client-server infrastructure was created, with UR-Walking acting as the client, and a specifically for this purpose created Python web server acting as the server.

4.3.1 UR-Walking

The UR-Walking Android app was changed, so that when it finishes a WiFi scan (which it was already programmed to perform), it creates a message for the server in JSON.

JSON was chosen due to it being supported well on both platforms, and because its abstraction layer is neither higher nor lower than this type of communication requires.

The message contains "model", "timestamp", and all received BSSIDs as keys, and the phone model (`Build.MODEL` ("Android API Build Documentation")), the current UNIX-timestamp in seconds, and all associated RSSIs as values. After construction, the message is sent to the server, which's IP address and port (8111) is hardcoded into the version of UR-Walking (these values can be changed in `MapLocator`). The port number was arbitrarily chosen.

When UR-Walking receives a response from the server with a prediction of the most likely user location node, that node is published in a `WifiLocationEvent` to all listeners.

For this development version of the app, the UI-thread listens for the event and displays the node name as a toast.

4.3.2 Classification Server

`classificationServer.py` is the script that executes the Python web server. The files `classifier.joblib` and `scaler.joblib` from *4.2.9 Serialization* must have been created on the same machine that is supposed to run the server and they must have been created before `classificationServer.py` is launched.

One important problem in server systems is that they must be able to answer to multiple clients at once, otherwise only one person at a time would be able to receive a location prediction. While Python's `http.server.HTTPServer` is a single threaded server, it was combined with socketserver's `ThreadingMixIn` using multiple inheritance into a custom class named `ThreadingSimpleServer`, which can now handle client requests in a multithreaded, non-blocking manner.

`classificationServer.py` first loads the feature vector column names, scaler, and classifier previously created and makes them globally available to all threads. It then starts the server on port 8111.

The server communicates using HTTP. It answers any GET request with a status code of 501 ("not implemented").

POST requests are looked at more closely: If the content type of the message is not set to "application/json" or if the data provided is not in valid JSON, the request is answered with status code 400 ("bad request"). If, however, the request is in the proper format, the JSON is transformed into a Python dictionary, and sent to the classification logic.

The request dictionary is combined with the feature vector head data frame to create a new data frame that has the same columns as `feature_vector_head.csv` and no additional columns. Any missing values are filled in and the vector is then scaled in the same manner as in *4.2.1 Transforming the Measurement Data* and *4.2.2 Readying the Transformed Data for Classification*. This data frame then has the proper form to be given to the classifier for prediction and the data is classified.

The classification result is put into JSON, listing the predicted likelihood (value) for every node (key), and listing the highest likelihood node (value) in the key "prediction". In the construction of this JSON data, all nodes with likelihood zero are omitted to not needlessly generate traffic and waste computational power of the receiving device.

This JSON is then sent as an answer to the request alongside status code 200 (“ok”).

5 Project Results and Evaluation

All parts of the software suite were successfully created and used. A detailed look to their operation as well as an evaluation of the classifiers follows in this chapter. All Android apps were run on the same Samsung Galaxy A5.

5.1 Fingerprinting

The fingerprinting app is still called “LandmarkImages” like the app it is based upon, as a name change was not deemed necessary for the functionality of this software suite.

Upon startup the app requests permissions (see Figure 4) for location, file system, microphone, and camera, all of which should be granted to allow the app to work optimally. The microphone and camera permissions are artefacts of the base version this one was derived from, but their removal would have led to big changes in the app’s `ViewModel`, which weren’t justified considering the issue can be resolved with two clicks.

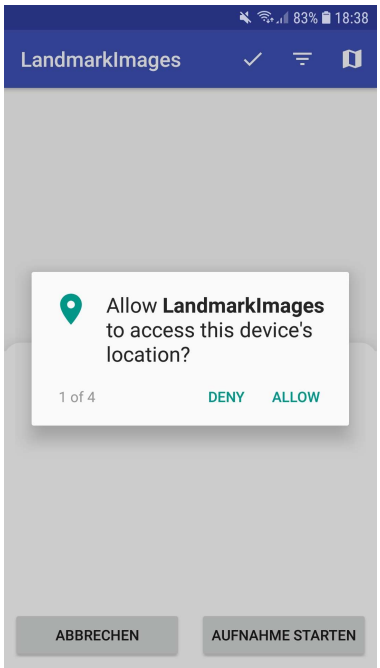

Figure 4 The app asking for user permissions (original screenshot)

Once started up, the app starts parsing internal XML data and cannot be used until that data is generated, see Figure 5. As far as was apparent during the tests of the

newly created version of the app, the tooltip telling the user to wait is faulty and the process never actually completes. The app should be left to calculate for five minutes and then be closed, either normally closed via the apps overview or force stopped via the app settings window. Then the app should be restarted again.

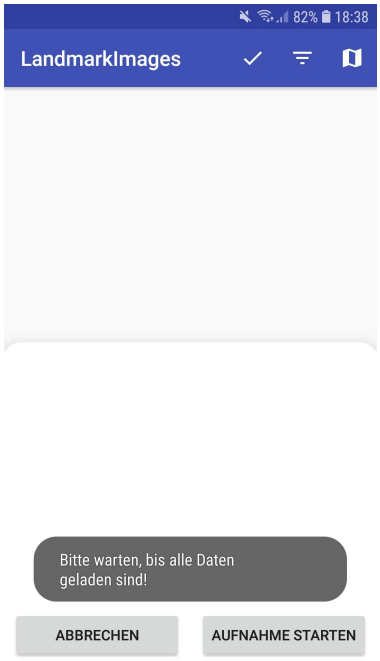

Figure 5 The app parsing XML data. "Please wait until all data is loaded!" (original screenshot)

Once restarted the app displays one floor plan of the university, see Figure 6. A single, named, blue node can be seen at the top of the screen with the label "1.41", which consists of the current floor, a period, and the node's unique number on that floor; together with the building/area name it can be combined to the node-ID.

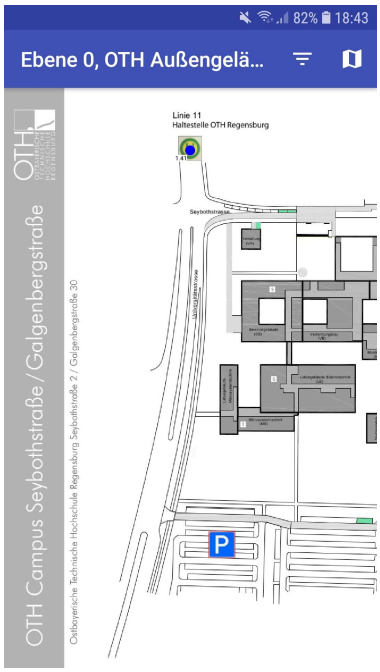

Figure 6 The first visible floor plan after restarting the app (original screenshot)

Now the floor plan for the building and floor to be measured can be selected by clicking the map icon in the top right corner, which opens a list of all floor plans that can be scrolled through, see Figure 7. For this thesis "UR Vielberth Gebäude, Ebene [0, 1, 2]" were measured.

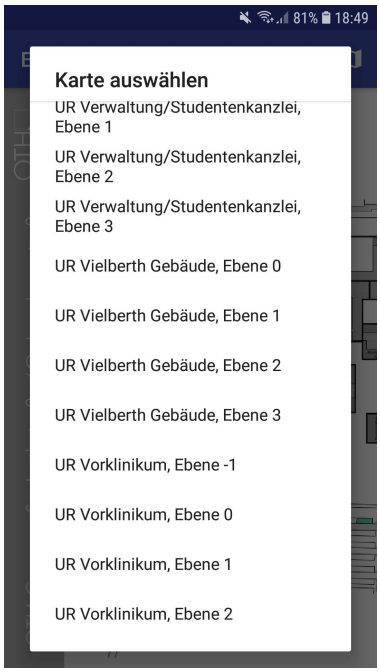

Figure 7 Floor plan selection. "Pick map" (original screenshot)

When viewing a map, swiping across the screen allows the user to move the map around.

Before a measurement can be taken, both the WiFi and the Location options of the Android system must be activated.

The user can zoom in on the floor plan by pinching two fingers on the screen like in most other applications. If zoomed in far enough, the app displays the node's human readable name, in addition to its unique identifier.

Now the node, that a measurement is being taken for, can be tapped once. The app has an option to create a new node at a new position should that be desired, but this isn't useful for the created software suite since that newly create node will not be known to UR-Walking, it is therefore important to tap an existing node. A selected node will display as a red circle, see Figure 8.

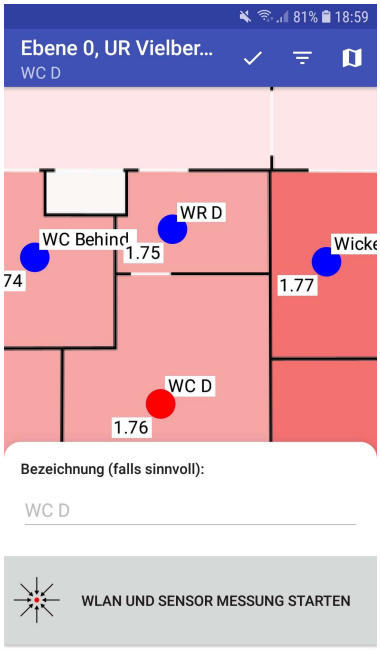

Figure 8 A node is selected (original screenshot)

Then the user has to press "WLAN UND SENSOR MESSUNG STARTEN", this displays the orientation selection mentioned in 4.1 Fingerprinting, see Figure 9.

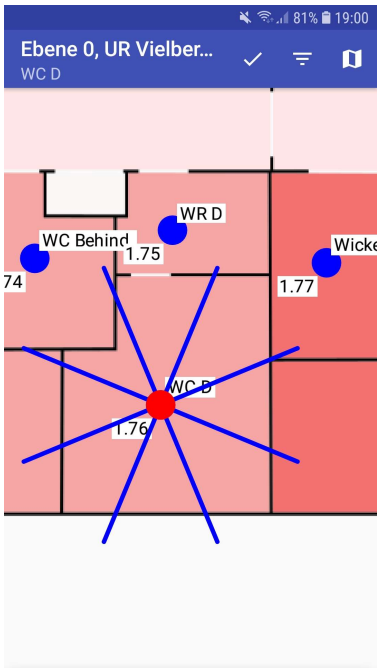

Figure 9 Orientation selection after clicking a node (original screenshot)

The user can then tap into a sector to get a display of the measurement menu. Pressing "AUFNAHME STARTEN" starts the measuring process, see Figure 10.

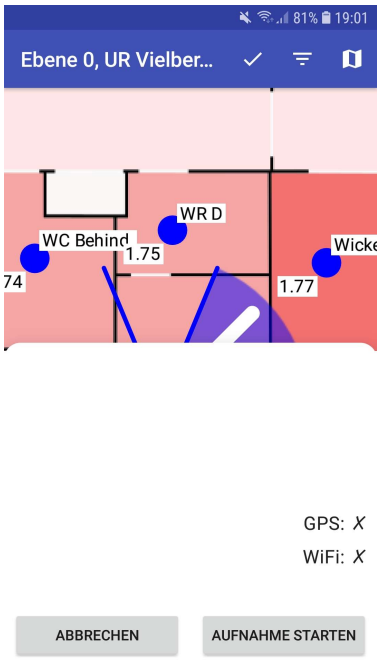

Figure 10 Screen after selecting a sector. The buttons are labeled "CANCEL" and "START MEASUREMENT" (original screenshot)

There is a display for GPS and WiFi displaying an "X" at the start of the measurement. Once GPS or WiFi have received a measurement event from the Android system, its "X" turns into a checkmark. Due to the original app's `ViewModel`, WiFi data cannot reliably be saved until at least one GPS measurement has been taken,

so the measurement must not be ended before both "X"s have turned into checkmarks. The user should walk around the room while collecting data. Every time a `ScanResult` becomes available, the app makes a sound, this allows the user to monitor that the app is in fact doing something. The recording must be ended by pressing "AUFNAHME BEENDEN", so all the measurements are saved as files, see Figure 11.

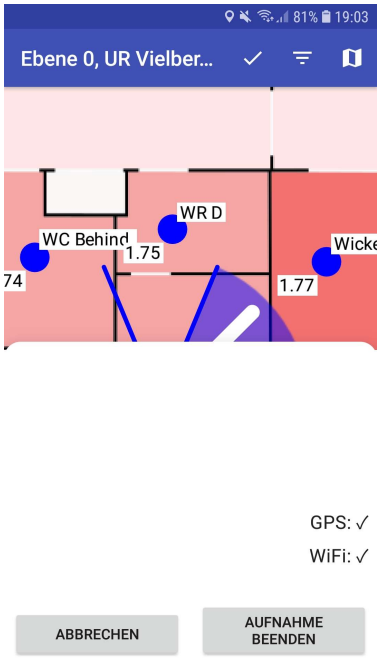

Figure 11 The recording screen after measurement has been going on for multiple seconds. The buttons are labeled "CANCEL" and "END MEASUREMENT" (original screenshot)

There is a bug that causes the app to crash during an extended measurement, usually it crashes when the 14th `ScanResult` for the same measurement is received, the reason for this is unknown and it is not consistently reproducible, but it is advisable to end the measurement and switch to a different sector (see Figure 12) when measuring bigger rooms to prevent data loss.

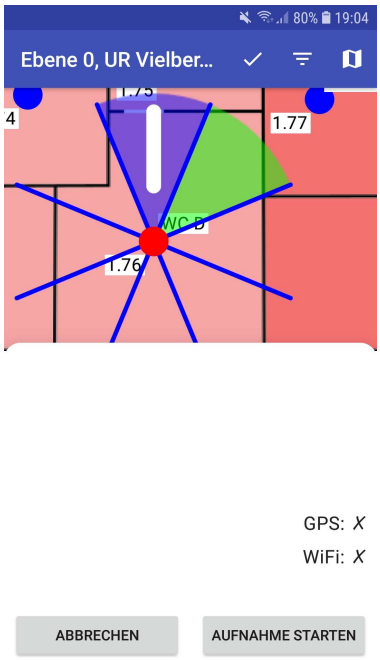

Figure 12 Selecting a second sector for measurement (original screenshot)

The app saves data into the phone's internal memory into the folder `LmImages > landmarks > BUILDING_NAME.LEVEL.NUMBER.ORIENTATION` where the timestamped information about GPS, pressure, WiFi, and orientation readings are saved as .CSVs alongside the phone model and detailed metadata about the node itself, see Figure 13.

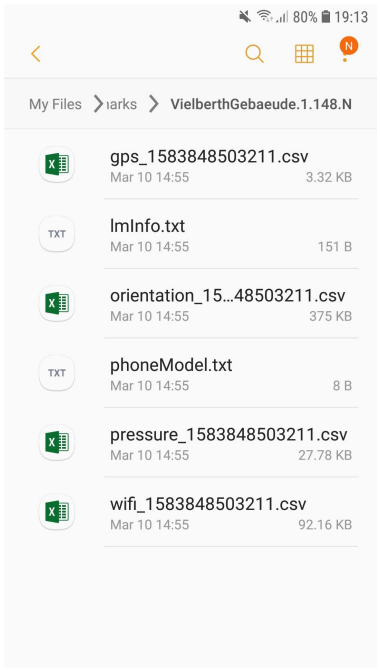

Figure 13 Measurement data saved on the phone (original screenshot)

For classification two datasets were created measuring most of the ground level of the Vielberth building, and the eastern and central part of the 2nd and 3rd story as well.

The measurements were taken on different days to make them as independent as possible. The first measurement series was taken with a measurement roughly every 1.5m in a grid like fashion; to allow for a high-quality fingerprint of every room. The second measurement series was taken while walking at roughly 3 to 4 km/h through the rooms and contains less data points per room; to speed up the creation of the dataset and to more closely mirror user behavior during navigation (when tracking the position of a user in real time, the system must be able to cope with a moving user.)

5.2 Classification

5.2.1 Preprocessing

Two preprocessing scripts were created, one for the preprocessing of the measurement data to train the classification algorithms with (named `transformMeasurementsToUnifiedDataset.py`,) and one for the preprocessing of the measurement data to test classifier performance (named `transformConfirmationMeasurementsToUnifiedDataset.py`.) These scripts pull data from the `measurements` and `confirmation_measurements` directory, respectively. Each one of these directories contains the landmarks folder seen in Figure 13 of each set measurement to be processed. If there are multiple sets of measurement to be added into a single preprocessing directory, they should be renamed to prevent data loss, as seen in Figure 14 and Figure 15.

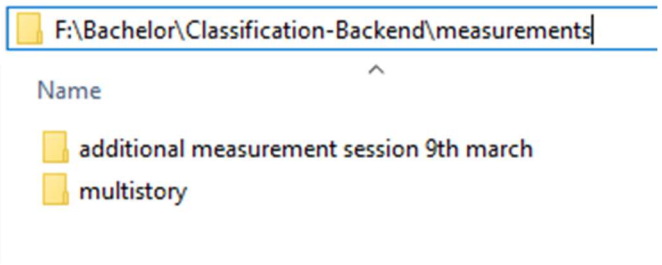

Figure 14 measurements directory with two different measurement sets (original screenshot)

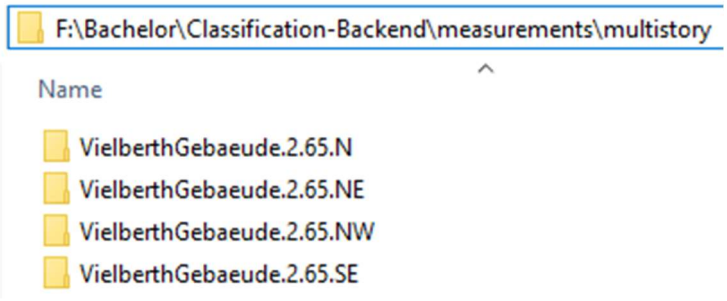

Figure 15 Content of one of the measurement directories (original screenshot)

The result of these preprocessing scripts is `combined_data.csv` and `confirmation_data.csv`, respectively and contain the information described in *4.2.1*, see Figure 16.

	A	B	C	D	E	DO	DP	DQ	DR
1		timestamp	00:a0:57:2d:87:8d	00:a0:57:2d:dc:c9	00:a0:57:30:bd:c8	pressure	nodeId	00:a0:57:30:be:fb	00:a0:57:35:2c:43
2	0	1583771393266				970.4630000000001	VielberthGebaeude.1.148		
3	1	1583771398446				970471	VielberthGebaeude.1.148		
4	2	1583771403558			-72.0	970.4989999999999	VielberthGebaeude.1.148		
5	3	1583771408828	-74.0			970447	VielberthGebaeude.1.148		
6	4	1583771413918			-64.0	970.4960000000001	VielberthGebaeude.1.148		

Figure 16 Part of combined_data.csv, the unified table of individual measurements for the purpose of training the classification algorithms (original screenshot)

5.2.2 Classification Script Settings

The script `createClassificationReport.py` handles the optimizing, training, persisting, and reporting of all tested classifiers. Its specific function must be set by the user by changing global variables at the start of the script. `K_FOLD_NUMBER` determines k for the purpose of k-fold cross validation. `SKIP_SEARCH` will cause hyperparameter optimization to be skipped if set to true. `OVERSAMPLE_VALIDATION_DATASET` will cause the development time test set to be oversampled if set to true. `OVERSAMPLE_KFOLD_VALIDATION_DATASETS` will cause test sets for individual folds in k-fold cross validation to be oversampled if set to true. `KFOLD_TEST_UPPER_STORIES_ONLY` will cause the script to only report classifier performance for floors above the ground floor. `SCALER` sets the scaler to be used for the feature vectors, this ended up being a `MinMaxScaler`. `OVERSAMPLE_ALL_CLASSIFIERS` will oversample the training sets for all classifiers if set to true. `SHOW_VALIDATION_SET_PERFORMANCE` will cause the script to display performance metrics of classifiers on the development time test set if set to true. `SHOW_KFOLD_CROSS_VALIDATION_PERFORMANCE` will show performance metrics averaged through k-fold cross validation on the entire training data. `SHOW_CONFIRMATION_SET_PERFORMANCE` will show performance metrics of

classifiers trained on all training data predicting the data of the second, independent set of measurements from the `confirmation_measurements` directory. `SUPPRESS_ALL_WARNINGS` can toggle the display of Python warnings in the program output, this is useful during hyperparameter optimization as the process produces a large amount of warnings about lack of convergence as well as redundant or incorrect parameter sets. `CLASSIFIERS_WITH_HYPERPARAMETER_DISTRIBUTIONS` contains all classifiers and their ranges of hyperparameters to be tested, if a classifier should not be optimized during execution of the program, it must be commented out here. `CLASSIFIERS_FOR_EVALUATION` contains classifiers with fixed hyperparameters for the purpose of testing them on the independent confirmation measurements. `MODEL_TO_SERIALIZE` contains the one classifier that should be persisted to be used by the server that communicates with the UR-Walking app.

`OVERSAMPLE_ALL_CLASSIFIERS`, `OVERSAMPLE_VALIDATION_DATASET` and `OVERSAMPLE_KFOLD_VALIDATION_DATASETS` are made redundant by imblearn's Pipeline and should be set to false; `RANDOMIZED_SEARCH_ITERATIONS` is also made redundant and has no effect.

In a typical workflow with this script selection of classifiers and optimization of their hyperparameters comes first, for this purpose `K_FOLD_NUMBER` should be set to at least 5, `SKIP_SEARCH` to false, `SCALER` to the appropriate scaler for the chosen classifiers, `SHOW_VALIDATION_SET_PERFORMANCE` to true, `SHOW_KFOLD_CROSS_VALIDATION_PERFORMANCE` to true, `SHOW_CONFIRMATION_SET_PERFORMANCE` to false. `SUPPRESS_ALL_WARNINGS` should be set to false during a first test execution to make sure that hyperparameter ranges are valid but should be set to true for the actual hours of calculation time of hyperparameter optimization. This will display optimal found hyperparameters and their performance.

In the next step these found hyperparameters for the classifiers should be entered into `CLASSIFIERS_FOR_EVALUATION`. `SKIP_SEARCH` should be changed to true, `SHOW_VALIDATION_SET_PERFORMANCE` and `SHOW_KFOLD_CROSS_VALIDATION_PERFORMANCE` should be changed to false, `SHOW_CONFIRMATION_SET_PERFORMANCE` should be set to true. Execution of this setting will show classifier performance trained on the entire training data when

predicting the independent test measurements and will also save the confusion matrix of every classifier into the `confusion_matrices` directory.

The best performing classifier should then be entered into `MODEL_TO_SERIALIZE`, `SHOW_CONFIRMATION_SET_PERFORMANCE` should be changed to false, `SUPPRESS_ALL_WARNINGS` should also be set to false. Execution of this setting will serialize the best classifier, trained on both the training data and the independent test data to later be loaded by the server for real world use.

5.2.3 Classifier Hyperparameters and Performance

This section lists each tested classifier with their best found hyperparameters and their balanced accuracy when trained on the entire training set and tested on the independent second set of measurements.

Classifier	**Balanced Accuracy**
K-Nearest Neighbors	91.6%
Feedforward Neural Network	91.5%
Random Forest	96.9%
Multinomial Naïve Bayes	89.1%
C-Support Vector Classification	92.3%
AdaBoost with Decision Tree	94.3%
Gradient Boosting Classifier	94.3%
Bagging Classification with Decision Tree	93.5%

Table 1 Classifier Performance

5.2.3.1 K-Nearest Neighbors

The optimal classifier found uses k = 3, known points closer to the data point to be classified "weigh […] by the inverse of their distance" ("sklearn documentation KNeighborsClassifier"), chooses between tree based and brute force approaches automatically based on the training data, and uses Manhattan distance. Oversampling was not used. It achieved a balanced accuracy of 91.6%.

5.2.3.2 Feedforward Neutral Network

The optimal classifier found uses a logistic activation function, alpha = 0.01, one hidden layer with 300 nodes, a constant learning rate of 0.001, stops training after 700

iterations, and uses the adam solver with beta_1 = 0.9 and beta_2 = 0.999. Oversampling was not used. It achieved a balanced accuracy of 91.5%.

5.2.3.3 Random Forest

The optimal classifier found uses 200 trees, the maximum number of features used by an estimator is $\log_2$ of the number of features, Gini impurity was used "to measure the quality of a split" ("sklearn documentation RandomForestClassifier"), and no weight was given to classes. Oversampling was used. It achieved a balanced accuracy of 96.9%.

5.2.3.4 Multinomial Naïve Bayes

The optimal classifier found uses alpha = 0.11678761101807189, a class prior that ascribes equal probability to all classes (equal to 1 divided by the number of classes, in this case 1/32), and set to not learn class prior probabilities from the data. Oversampling was used. It achieved a balanced accuracy of 89.1%.

5.2.3.5 C-Support Vector Classification

The optimal classifier found uses a linear kernel, C = 0.5533914322723179, decision function shape ovo, does not use the shrinking heuristic, and is set to predict probabilities rather than just classify as this is necessary for both performance analysis and server functionality. Oversampling was not used, though the use of oversampling showed no negative effect on performance during development time. It achieved a balanced accuracy of 92.3%.

5.2.3.6 AdaBoost with Decision Tree

The optimal classifier found uses 100 estimators, a learning rate of 0.9, and the SAMME.R real boosting algorithm. The used base estimator is a `DecisionTreeClassifier` without pruning, with equal class weights, using Gini impurity, a maximum depth of 12, maximum features equal to the number of features, unlimited number of maximum leaf nodes, splitting at a minimum decrease of impurity of 0, a minimum of 1 sample per leaf, a minimum requirement of 2 samples to split a node, and the splitter strategy best. Oversampling was used. It achieved a balanced accuracy of 94.3%.

5.2.3.7 Gradient Boosting Classification

The optimal classifier found uses 400 estimators, a learning rate of 0.1, a maximum number of features equal to the square root of features, a maximum depth for individual regression estimators of 3, and base learners used 0.215 times the number of samples. Oversampling was used. It achieved a balanced accuracy of 94.3%.

5.2.3.8 Bagging Classification with Decision Tree

The optimal classifier found uses 200 estimators, does not "use out-of-bag samples to estimate the generalization error" ("sklearn documentation BaggingClassifier") and its base estimator is a `DecisionTreeClassifier` without pruning, with equal class weights, using Gini impurity, a maximum depth of 18, maximum features equal to the number of features, unlimited number of maximum leaf nodes, splitting at a minimum decrease of impurity of 0, a minimum of 1 sample per leaf, a minimum requirement of 2 samples to split a node, and the splitter strategy best. Oversampling was used. It achieved a balanced accuracy of 93.5%.

5.2.4 Confusion Matrix

This section will analyze the confusion matrix of the best performing classifier, random forest, which had a balanced accuracy of 96.9%. Out of 166 samples in the test set, four were classified incorrectly as shown in Table 2.

	1.62	1.70	2.68	2.78
1.64	1			
1.68		1		
2.78			1	
3.23				1

Table 2 Confusion matrix of random forest. Expected / correct prediction is the row name, actual / erroneous prediction is the column name. Values inside cells are the number of incorrect predictions for each set, empty cells have 0 incorrect predictions. Room names are coded as "[floor number].[room number]" so 1.68 is room 68 on the 1st floor.

The software predicts 1.62 (foyer) when 1.64 (catering area) is correct, this is easily explained as the rooms are directly next to each other with an open doorway between them and measurements were taken very near to that doorway, see Figure 17.

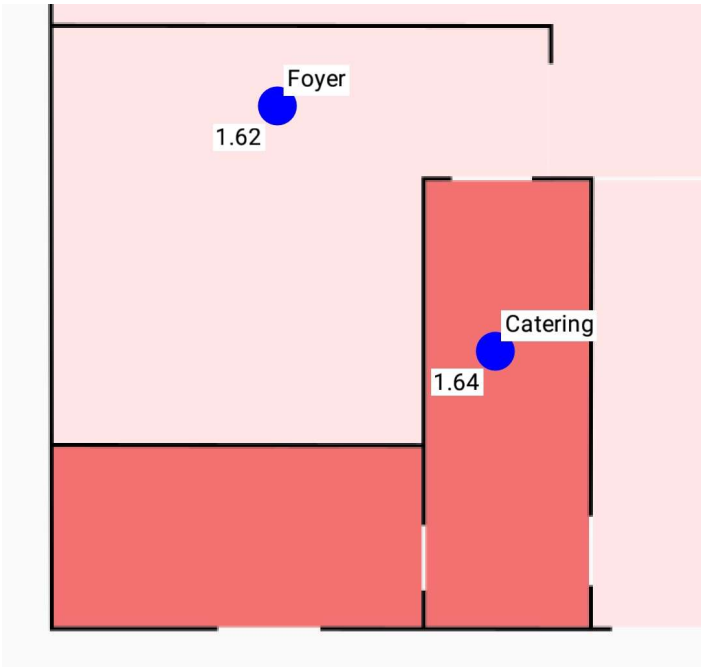

Figure 17 Room 1.62 to the left and room 1.64 to the right (original screenshot)

The error of predicting room 1.70 (men's restroom) instead of 1.68 (staircase and elevator area) can be seen in Figure 18. There is no special explanation for this mistake except possibly a weak WiFi connection as there was some trouble measuring any signal at all during both measurement sets in both areas.

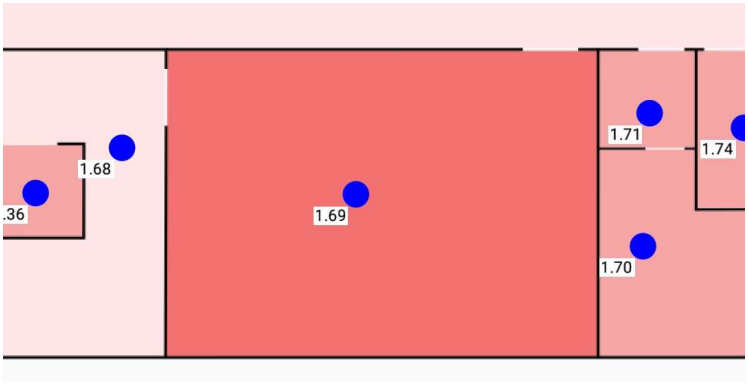

Figure 18 Room 1.68 to the left and room 1.70 to the right (original screenshot)

The error of predicting room 2.68 (men's restroom) instead of room 2.78 (staircase) can be seen in Figure 19. It seems to be a similar case to the former problem and again had trouble receiving WiFi signals during measurement.

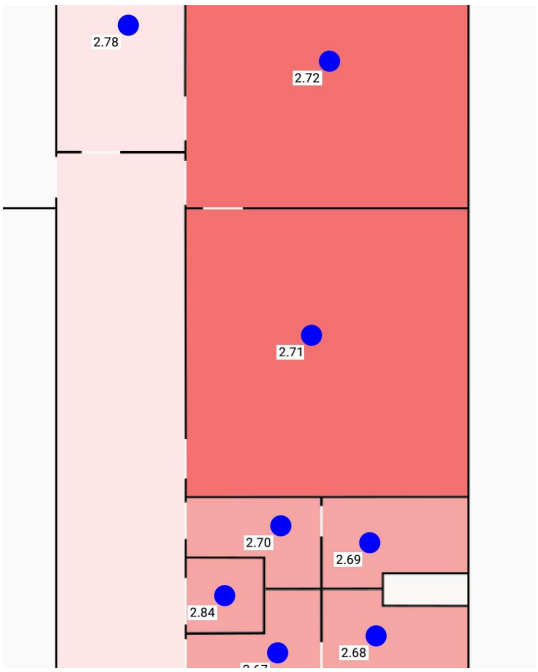

Figure 19 Room 2.78 at the top and room 2.68 in the bottom right (original screenshot)

The error of predicting room 2.78 instead of room 3.23 (see Figure 20) has a significantly easier explanation: it is the same room, different ends of the same staircase with no ceiling between them and measurements were as close as 1 meter from each other. The noteworthy detail here is not that this error occurred, but that this error occurred only once despite there being 4 separate staircases reaching across 3 floors, indicating that random forest is very good at solving this multistory problem.

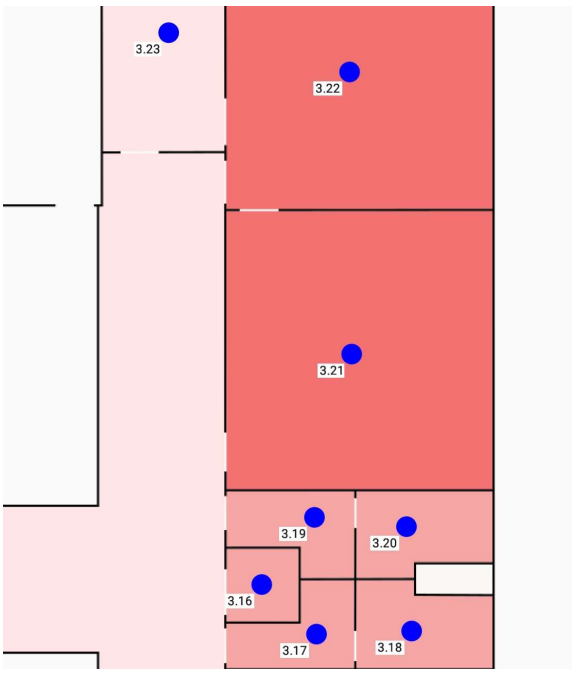

Figure 20 Room 3.23 at the top left (original screenshot)

5.3 Prediction in UR-Walking

The script classificationServer.py loads the saved scaler and classifier and allows the UR-Walking app to communicate with the classifier and receive predictions about the user's location. Once the server is running, it can be tested using curl, a sample command is contained within `classificationServer.py`, the server response looks like this when entered into the command line:

```
{"VielberthGebaeude.1.148": 0.73, "VielberthGebaeude.1.149":
0.005, "VielberthGebaeude.1.57": 0.07,
"VielberthGebaeude.1.60": 0.035, "VielberthGebaeude.1.62":
0.01, "VielberthGebaeude.1.64": 0.025,
"VielberthGebaeude.1.68": 0.045, "VielberthGebaeude.1.71":
0.01, "VielberthGebaeude.1.77": 0.03,
"VielberthGebaeude.3.47": 0.02, "VielberthGebaeude.3.48":
0.02, "prediction": "VielberthGebaeude.1.148"}
```

The server responds in JSON format with the predicted probabilities for different rooms, cutting away rooms with a probability of 0. The server also puts the room that is considered the most likely as the value of a key called prediction so that the mobile device will not have to search for the most likely prediction if that is required.

This server must be installed on a machine with an opened port and a consistent IP or domain address. IP/domain and port must then be entered into the code of the UR-Walking app so that it can request from the running server.

When the server was installed on Linux, an error with the decode function occurred and had to be fixed locally. The code attached to this document was developed on a Windows 10 machine and different operating systems might have different default behaviors regarding data types in Python.

The server with a development version of the UR-Walking app was successfully tested using real, live data and network communication. It proved accurate, though slightly delayed, usually displaying the position of a few seconds ago, as was to be expected according to literature. The use case for this information is purely internal, but for this paper a version of the app was created that displays the room prediction as toast, see Figure 21.

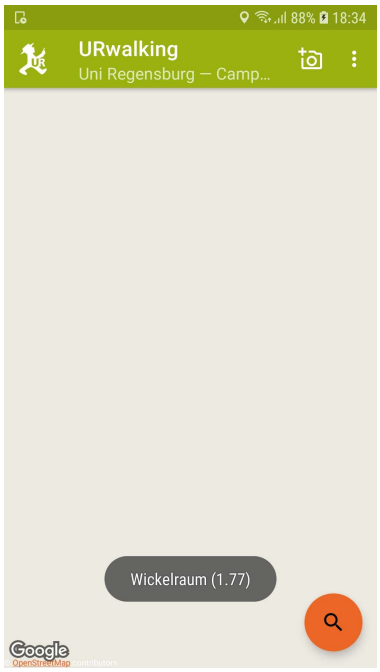

Figure 21 UR-Walking app displaying toast with location prediction. Map is not loaded due to the screenshot being taken off-campus due to the COVID-19 pandemic (original screenshot)

6 Conclusion

Random forest was the best classifier with a balanced accuracy of 96.9%, performing well overall and in multistory problems which reaffirms the strength of ensemble learning algorithms for this type of problem as stated by Bozkurt et al. (2015, p. 1).

The developed software suite meets the requirements with good accuracy and could be used as a base for future development of the UR-Walking app.

K-nearest neighbor performed worse than random forest even though the former is often reported to perform the best for many problems in the literature, it is unclear if this is an outlier or if ensemble learning outperforms KNN with appropriate hyperparameters.

When accounting for the limitations discussed in the next section, the findings of this project can be used for future implementations of indoor navigation software, some of which will be discussed in *8 Future Work*.

7 Limitations

It is unclear if random forest's performance is an indicator of general strength of the classifier for this type of problem, or if it only performed well due to random chance or the structure of the building the measurements were taken in.

The UR-Walking node system does not represent rooms well and sometimes the question of what a room is, is unclear. Some rooms are considered to be two rooms while physically and pragmatically being 1 room, for example in Figure 18 rooms 1.70 and 1.71 are one restroom, 1.70 contains toilets and 1.71 contains sinks. A navigation system needs to think about the concept of room carefully before building its representation of the buildings to be navigated through.

If a router is replaced, the area around the router needs to be fingerprinted again and existing fingerprints need to be deleted, furthermore the classifier needs to be re-trained; solutions to this problem are proposed and analyzed in Zhuang, Syed, Li, and El-Sheimy (2016).

Fingerprinting using a phone that only records WiFi data every four to five seconds is not very efficient. Using a specifically built, higher frequency device, or using multiple phones would allow for more and quicker data collection.

The developed fingerprinting app has a non-reproducible bug that causes it to crash, thus making taking fingerprints hard.

As noticed during the real-world testing, the WiFi positioning lags behind the actual user position. This must be accounted for when WiFi based systems are used for problems; these systems must either find a way to compensate the lag via prediction or the lag must be acceptable to the nature of the problem the system is trying to solve.

The software suite developed here might have scaling issues due to the effectively quadratic space complexity of the used data: the software uses a table that uses BSSID as columns and room-IDs as rows, thus the table has a space complexity of m * n and as new buildings are added, they add both rooms and BSSIDs. In the long term this increases processing and RAM and disk space requirements significantly.

The process of optimizing hyperparameters is very manual and calculation time expensive, also it is hard to know if a correct range for hyperparameters is used. Implementing a more intelligent search than random search would be advantageous.

The Python code for classification is not a streamlined pipeline, very problem specific, and requires editing of the actual Python code rather than automating the entire workflow. Now that the initial problem is solved, individual flaws are better visible, and the required steps to solve the problem are more understood and defined, the workflow presented in this paper can be more cleanly implemented by simply refactoring the source code of this project with a larger focus on software engineering than proof of concept.

Using the same feature vector scaler for every classifier is possibly a mistake. It should be investigated if there are significant differences in performance when different scalers are used.

The created server software does not use SSL, an unnecessary lack of security that must be fixed before the software is used in a consumer product.

8 Future Work

As discussed in *2 Related Work*, the room prediction of this software can be combined with inertial sensors using a Kalman filter to create a more accurate and reliable location prediction system.

Before further implementations of the software are performed, the concept of room (as discussed in *7 Limitations*) as well as the merit of room classification compared to the merit of real world coordinate regression should be researched or discussed.

Random forest performed very well in this context, it should be researched if this occurrence is an outlier or if random forest is an optimal algorithm in this problem space.

The results of k-fold cross validation and validation on an independent dataset were virtually identical during development of this software suite. It should be researched if k-fold cross validation is always sufficient in this problem context as it would save a lot of resources compared to having to collect a second dataset for verification of model generalizability.

Strategies to reduce the quadratic space complexity discussed earlier need to be found to let the software solve problems at real-world scale. Possible approaches are utilizing main component analysis or training multiple classifiers, one that classifies

buildings first and multiple ones that perform WiFi data classification for an individual building each.

The data collected during fingerprinting might be usable to determine quality of WiFi connection for each room by looking at values like the median maximum signal strength of all measurements taken in a single room, but it would have to be assessed first if RSSI is actually a good indicator for WiFi quality. Such information could then be displayed in UR-Walking or on digital campus maps.

Before using more resources on WiFi based indoor navigation for UR-Walking, a thorough review of the literature should be conducted, and backlog of system requirements should be created. A lot of the issues found during development of this software suite were necessitated by the initial task but were already known in the more recent literature. One big meta-review of the problem can save development resources in the future by preventing mistakes that were already made by others.

References

Accelerometer Wikipedia (2020a March 2). Retrieved from https://en.wikipedia.org/wiki/Accelerometer

Android API Build Documentation. Retrieved from https://developer.android.com/reference/android/os/Build

Android API ScanResult Documentation. Retrieved from https://developer.android.com/reference/android/net/wifi/ScanResult

Bozkurt, S., Elibol, G., Gunal, S., & Yayan, U. (2015). A comparative study on machine learning algorithms for indoor positioning. In *International Symposium on Innovations 02.09.2015* (pp. 1–8). https://doi.org/10.1109/INISTA.2015.7276725

C. Shah, & A. G. Jivani (2013). *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. [Place of publication not identified]: IEEE.

Chang, C.-C., & Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, *2*(3), 1–27.

Chug, A., & Dhall, S. (2013). Software defect prediction using supervised learning algorithm and unsupervised learning algorithm.

Cross-validation: evaluating estimator performance (2020, July 30). Retrieved from https://scikit-learn.org/stable/modules/cross_validation.html

D. W. Aha, D. Kibler, & M. K. Albert (2011). Instance-Based Learning. In *SpringerReference*. Berlin/Heidelberg: Springer-Verlag. https://doi.org/10.1007/SpringerReference_179121

Elbasiony, R., & Gomaa, W. (2014). WiFi localization for mobile robots based on random forests and GPLVM. In *2014 13th International Conference on Machine Learning and Applications.* Symposium conducted at the meeting of IEEE.

Freund, Y., & Schapire, R. E. (1996). Schapire R: Experiments with a new boosting algorithm. In *In: Thirteenth International Conference on ML.* Symposium conducted at the meeting of Citeseer.

Gyroscope Wikipedia (2020 March 1). Retrieved from https://en.wikipedia.org/wiki/Gyroscope

He, S., & Chan, S.-H. G. (2016). Wi-Fi Fingerprint-Based Indoor Positioning: Recent Advances and Comparisons. *IEEE Communications Surveys & Tutorials*, *18*(1), 466–490. https://doi.org/10.1109/COMST.2015.2464084

(2017). *Low-effort place recognition with WiFi fingerprints using deep learning*. *Advances in intelligent systems and computing: Vol. 550*. Cham: Springer.

John, G. H., & Langley, P. (2013). Estimating continuous distributions in Bayesian classifiers. *ArXiv Preprint ArXiv:1302.4964*.

Johnson, J. M., & Khoshgoftaar, T. M. (2019). Survey on deep learning with class imbalance. *Journal of Big Data*, *6*(1), 27. https://doi.org/10.1186/s40537-019-0192-5

Leppäkoski, H., Collin, J., & Takala, J. (2013). Pedestrian Navigation Based on Inertial Sensors, Indoor Map, and WLAN Signals. *Journal of Signal Processing Systems*, *71*(3), 287–296. https://doi.org/10.1007/s11265-012-0711-5

MAC address Wikipedia (2020b March 2). Retrieved from https://en.wikipedia.org/wiki/MAC_address

Magnetometer Wikipedia (2020 February 2). Retrieved from https://en.wikipedia.org/wiki/Magnetometer

Random Oversampling and Undersampling for Imbalanced Classification. Retrieved from https://machinelearningmastery.com/random-oversampling-and-undersampling-for-imbalanced-classification/

Roweis, S., & Ghahramani, Z. (1999). A unifying review of linear gaussian models. *Neural Computation*, *11*(2), 305–345. https://doi.org/10.1162/089976699300016674

Sanger, T. D. *Optimal unsupervised learning in a single-layer linear feedforward neural network.*

Shams, R., & Mercer, R. E. (2013). Classifying spam emails using text and readability features. In *2013 IEEE 13th international conference on data mining*. Symposium conducted at the meeting of IEEE.

Sharif, S. O., Kuncheva, L. I., & Mansoor, S. P. (2010). Classifying encryption algorithms using pattern recognition techniques. In *2010 IEEE International*

Conference on Information Theory and Information Security. Symposium conducted at the meeting of IEEE.

Sklearn documentation accuracy score. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

Sklearn documentation BaggingClassifier. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html

Sklearn documentation balanced accuracy score. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.balanced_accuracy_score.html

Sklearn documentation GradientBoostingClassifier. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

Sklearn documentation GridSearchCV. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Sklearn documentation KNeighborsClassifier. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Sklearn documentation MinMaxScaler. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

Sklearn documentation MLPClassifier. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

Sklearn documentation Model persistence. Retrieved from https://scikit-learn.org/stable/modules/model_persistence.html

Sklearn documentation MultinomialNB. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html

Sklearn documentation MultinomialNB. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html

Sklearn documentation RandomForestClassifier. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Sklearn documentation RandomizedSearchCV. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

Sklearn documentation SVC. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

Sklearn documentation Tuning the hyper-parameters of an estimator. Retrieved from https://scikit-learn.org/stable/modules/grid_search.html

Training, Validation, and Test sets Wikipedia (2020 February 19). Retrieved from https://en.wikipedia.org/wiki/Training,_validation,_and_test_sets

Wu, C.-L., Fu, L.-C., & Lian, F.-L. (2004). WLAN location determination in e-home via support vector classification. In *IEEE International Conference on Networking, Sensing and Control, 2004.* Symposium conducted at the meeting of IEEE.

Wu, C., Yang, Z., Liu, Y., & Xi, W. (2012). WILL: Wireless indoor localization without site survey. *IEEE Transactions on Parallel and Distributed Systems*, *24*(4), 839–848.

Yang, Z., Wu, C., Zhou, Z., Zhang, X., Wang, X., & Liu, Y. (2015). Mobility increases localizability: A survey on wireless indoor localization using inertial sensors. *ACM Computing Surveys (Csur)*, *47*(3), 1–34.

Yunhao Liu, Zheng Yang, Xiaoping Wang, Lirong Jian (2010 March). Location, localization, and localizability. *JOURNAL of COMPUTER SCIENCE and TECHNOLOGY*. (25(2)), 274–297. Retrieved from https://link.springer.com/content/pdf/10.1007/s11390-010-9324-2.pdf

Zhuang, Y., Syed, Z., Li, Y., & El-Sheimy, N. (2016). Evaluation of Two WiFi Positioning Systems Based on Autonomous Crowdsourcing of Handheld Devices for Indoor Navigation. *IEEE Transactions on Mobile Computing*, *15*(8), 1982–1995. https://doi.org/10.1109/TMC.2015.2451641