Praktikumsbericht Mobile Sensors

Jan-Hendrik Borth Lukas Härtel Thomas Kaspers
Björn Kreutz David Mebus Maximilian Meffert
Lukas Müller Alexander Stab Anna-Carina Strunk

8. Mai 2014

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Der fortschreitende Erfolg des Fahrrades im innerstädtischen Verkehr macht neue Wege der Navigation und Auswertung der individuellen Fahr-Erfahrung notwendig. Die Ausgangsidee beschreibt ein System zur Verkehrsflussanalyse von Fahrrädern, die mithilfe eines Smartphones erhoben werden kann.

Im Speziellen werden Fahrten auf Bodenbeschaffenheit, Bremsvorgängen und zu überquerenden Bordsteinkanten überprüft. Wenn eine solche Metrik im Kleinen von Einzelpersonen vorliegt, können Rückschlüsse über Straßenabschnitte oder sonstige Fahrradwege gezogen werden. Um die Statistiken für einzelne Anwender zu erheben, muss ein einfaches und einheitliches Verfahren bereitgestellt werden, mit dem die Benutzer die Vermessung betreiben können. Das Smartphone bietet mit seinen eingebauten Lagesensoren und Verortungsmöglichkeiten die ökonomischste und am leichtesten zugängliche Lösung an.

Das vorliegende Projekt stellt eine Applikation für Android-Smartphones vor, die einem Anwender die genannten Möglichkeiten geben soll. Dazu werden im Programm erfasste Sensordaten über Beschleunigung und Position an einen Server im Internet gegeben, wo sie anhand von maschinengelernten Algorithmen ausgewertet werden. Die Ergebnisse werden dann für die Anwender und Administratoren auf einer Webseite dargestellt. Das System besteht damit aus drei Komponenten mit dem Server im Mittelpunkt.

Die Auswertung der Einzelstatistiken im Verbund ist im endgültigen Entwicklungsstand nicht mehr implementiert. Die Komponenten können allerdings einfach erweitert und wiederverwendet werden. Die Schnittstellen sind Spezifiziert und Datenerhebung parametrisierbar. Im Ausblick steht also die Vollendung der Verkehrsflussanalyse mit Algorithmen auf Fahrstrecken und den aufgetretenen Ereignissen.

Im vorliegenden Bericht wird die Organisation, die Konzeption und die Implementation erläutert. Dazu wird im ersten Kapitel der wissenschaftliche Hintergrund erläutert, danach die Organisationsstruktur des Teams und im folgenden die Architektur der Komponenten. Deren Umsetzung wird in der Implementation beschrieben.

Da ein integraler Bestandteil des Programms das Versenden von Sensordaten ist, leitet sich ein Forschungszweig ab. In diesem wird in einer Fallstudie das Komprimieren und anschließende Versenden von Daten dem unkomprimierten Versand gegenübergestellt und die Frage beantwortet, welches der beiden Verfahren Energiesparender ist. Das Kapitel der erweiterten Aspekte berichtet und erklärt die Ergebnisse.

Das Fazit schließt den Bericht mit dem aktuellen Stand der Arbeiten und dem Ausblick ab und setzt das Projekt in Relation zu anderen Projekten. Dem Dokument nachgestellt ist das Anwenderhandbuch, in dem die Anwendung der entwickelten Komponenten als Endbenutzer oder als Dienstleister erklärt wird.

2. Wissenschaftlicher Hintegrund

In den folgenden Abschnitten werden die Kenntnisse aus der Vorbereitungsphase aufbereitet. Die behandelten Themen decken die notwendigen Fähigkeiten zur Entwicklung des mobilen Systems, der Ereigniserkennung und deren Anwendung auf Menschen ab. Außerdem wird eine Möglichkeit für die Bewertung von Straßenqualität aufgezeigt. Des weiteren werden Darstellungsmöglichkeiten von geografischen Informationen mithilfe von offenen Kartendiensten beschrieben.

2.1. Entwicklung für Android

Android ist ein auf dem Linux Kernel basierendes Betriebssystem, das von der *Open Handset Alliance* [1] entwickelt wird und für Mobilgeräte wie Smartphones und Tablets angedacht ist.

Da Mobilgeräte sowohl in ihrer Rechenleistung als auch ihrem Speicher limitiert sind und zudem akkubetrieben werden, wurde Android so entwickelt, dass es ressourcenschonend und schnell operiert. Dazu trägt die eigens für Android entwickelte *Dalvik Virtual Machine* bei, welche so konzipiert wurde, dass sie für jede Anwendung eine eigene *Virtual Machine* bereitstellen kann. Das führt dazu, dass jede Anwendung abgekapselt und somit sicher von anderen Anwendungen ausgeführt wird. Anwendungen für Android werden zwar in Java geschrieben, jedoch nicht von der *Java Virtual Machine* ausgeführt, sondern von der *Dalvik Virtual Machine*. Daraus ergibt sich, dass nicht alle Libraries unterstützt werden (z.B. Swing).

Ein weiterer Unterschied zu herkömmlichen Java-Projekten ist der Inhalt eben dieser. Zu den wichtigsten Bestandteilen eines Android-Projekts gehören sowohl die Ordner *src* und *res*, als auch die *AndroidManifest.xml*. Während sich im src-Ordner der Java Quellcode aufhält, befinden sich im res-Ordner alle notwendigen Anwendungsressourcen, wie beispielsweise Bilder in unterschiedlichen Auflösungen oder XML-Dateien die das Layout beschreiben. Die *AndroidManifest.xml* stellt dabei das Herzstück eines Android-Projektes dar. Hier müssen dem Projekt angehörige Packages aufgeführt werden, welche Android API für das Ausführen der Anwendung vorausgesetzt werden. Ein weiterer wichtiger Punkt ist die Festlegung aus welchen Komponenten sich die Anwendung zusammensetzt. Unter Komponenten versteht man *activities, services, broadcast receiver und content providers*.

Eine *Activity* bezeichnet das, was der Nutzer letztendlich sieht. Sie beinhaltet das User Interface (UI), mit dem der Nutzer interagiert. Jeder neue Bildschirm der angezeigt

[1] http://www.openhandsetalliance.com/ - 2007 gegründet von Google. Ein Konsortium bestehend aus Softwareunternehmen, Chipherstellern, Netzbetreibern und Mobiltelefonherstellern, das sich dafür einsetzt offene Standards für Mobilgeräte zu schaffen

wird, repräsentiert normalerweise eine *Activity*. *Activities* werden nach dem *last in, first out Prinzip* angeordnet. Wird eine *Activity* geschlossen, wird die zuletzt aufgerufene *Activity* angezeigt. Da Ressourcen gespart werden müssen, unterliegen *Activities* einem speziellen *Lifecycle* [Abb. 2.1][2]. Wie in der Abbildung zu sehen, sind dies die Phasen *onCreate, onStart, onResume, onPause, onStop, onDestroy und onRestart.*

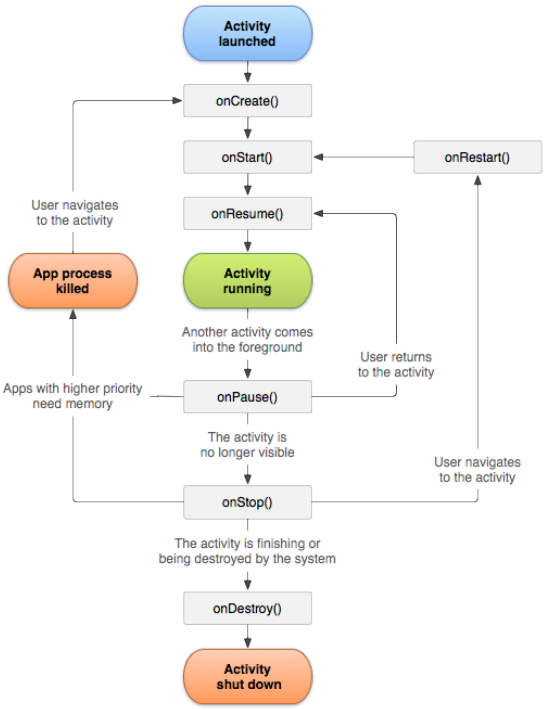

Abbildung 2.1.: Activity Lifecycle nach developer.android.com

Soll mehr als nur ein UI und kleinere Funktionen möglich sein, wird also eine Komponente benötigt, die außerhalb des Lifecycles arbeitet, damit der Prozess nicht wegen eines wechselnden Zustands unterbrochen wird. Eine solche Komponente nennt sich *Service*. Ein Service wird von einer Activity gestartet und läuft ab dann im Hintergrund. Selbst wenn die Applikation beendet wird, kann der Service weiter laufen, ist allerdings grundsätzlich nicht in der Lage, mit der aufrufenden Activity kommunizieren. Um das zu gewährleisten, muss der Service an die Activity gebunden werden. Ein so genannter *Bound Service* ermöglicht sogar Inter Process Communication (IPC), wird allerdings auch mit der Activity, an die er gebunden wurde, zerstört.

Grundsätzlich wird jeder Android-Anwendung ein Prozess zugeteilt, in dem der sogenannte *Main-Thread* jegliche Komponenten instanziiert. Der Main-Thread wird auch *User-Interface-Thread* genannt, da in ihm empfehlenswerterweise nur eine Activity laufen sollte, die sich um jegliche User Interface Manipulationen kümmert. Grundsätzlich

[2]http://developer.android.com/guide/components/activities.html

ist es zu vermeiden, den User-Interface-Thread zu blockieren, weshalb alle intensiven, bzw. längerandauernden Operationen in separate Threads oder eventuell auch Prozesse auszulagern sind. Dies ist möglich, da sich jede Komponente einem Prozess oder Thread zuweisen lässt, was in der *AndroidManifest.xml* geschieht. Dabei ist zu beachten, welche Komponenten in einem Prozess ausgeführt werden, da Android Prozesse entfernt, wenn der Akkustand gering ist oder Speicher für wichtigere Prozesse notwendig ist. Unter Android werden Prozesse in die *importance hierarchy* eingeteilt, bestehend aus fünf Prioritätsstufen. Prozesse, die als *foreground process* eingestuft werden, haben die höchste Priorität und werden nur entfernt, wenn der Speicher nicht ausreicht um diese weiterhin auszuführen. Darunter fallen Prozesse, die einen direkten Einfluss auf das haben, was der Benutzer gerade tut, beispielsweise *Activties* mit denen der Benutzer direkt interagiert. Ein *visible process* enthält Komponenten, die nicht direkt im Vordergrund der User-Interaktion stehen, aber dennoch im Hintergrund sichtbar sind. Ein Beispiel hierfür wäre, wenn ein Dialog aufgerufen wird, welcher dann im Vordergrund steht, aber im Hintergrund noch die vorherige Activity zu sehen ist. *Service processes* fallen in keine der gerade erwähnten Prioritätsstufen. Hierzu gehören Services, die im Hintergrund arbeiten, somit zwar keinen direkten Einfluss auf das haben, was der Benutzer auf seinem Bildschirm sieht, aber dennoch für ihn wichtig ist (z.B. Musikplayer). Prozesse die als *background processes* eingestuft werden, können zu jeder Zeit entfernt werden, da es hierbei um Activites handelt, die für den Benutzer nicht sichtbar sind. Diese werden in einer Liste verwaltet, in der Prozesse dieser Prioritätsstufe die am seltensten genutzt wurden, entfernt werden. Die niedrigste Priorität haben *empty processes*. Hierbei handelt es sich um Prozesse die keine aktiven Komponenten beinhalten und lediglich zum Cachen von Inhalten um Startzeiten der jeweiligen Anwendung zu verbessern dienen.

Um neue Activites oder Services zu starten, werden *Intents* benötigt. Dabei beschreibt der Intent die zu startende Activity bzw. den zu startenden Service und überträgt die benötigten Daten. Um die App-Komponenten zu aktivieren, werden *Intent Objects* verschickt. Diese Objekte tragen alle nötigen Informationen, die die Komponente benötigt. Dazu können unter anderem der Name der Komponente, die auszuführende Aktion oder benötigte Daten (bei einem Telefon zum Beispiel die anzurufende Telefonnummer) gehören. Mit einem Intent Filter kann zusätzlcih angegeben werden, welche Intents eine Activity oder ein Service empfangen kann oder darf.

2.2. OpenStreetMap

OpenStreetMap ist eine freie Weltkarte, an der jeder mitarbeiten kann. Die gesammelten Geodaten werden sowohl in Rohform, als auch in Form von gerenderten Kartenbildern angeboten. Die Daten werden größtenteils mit GPS-Geräten durch die OpenStreetMap-Community gesammelt. Eine einfache Meldung von Fehlern begünstigt die Korrektheit der Daten.

Die OpenStreetMap-Datenbank wird unter der Lizenz *Open Database License* verteilt. Von den Daten abgeleitete Datenbanken dürfen nur unter dieser Lizenz weiter verteilt werden, hergestellte Werke (z.B. ausgedruckte Karten) müssen lediglich mit einem

Quellenhinweis versehen werden. Solange die Lizenzbedingungen eingehalten werden, ist jegliche Art der Nutzung, einschließlich der gewerblichen, zulässig.

2.3. Human Activity Recogniton

HAR [3] behandelt das Themengebiet der Erfassung von menschlichen Verhaltensmustern mithilfe von technischen Verfahren, im allgemeinen anhand von Körpervermessung durch Sensoren, vorrangig auf Bewegungsabläufe bezogen. Exemplarisch dafür kann man Gehen, Sitzen, Rennen oder Fahrrad fahren nennen. Das informatische Konzept, das sich dahinter verbirgt, ist die Anwendung von Klassifikatoren auf Mengen von Sensordaten.

Die Klassifikatoren ordnen bestimmten Teilmengen dieser Daten kontinuierliche oder diskrete Werte zu. Mit kontinuierlichen Ausgaben kann angegeben werden, wie sehr eine bestimmte Klasse erfüllt wird. Mit diskreten Werten kann dagegen eindeutig zwischen Klassen unterschieden werden. Nimmt man etwa an, dass man herausfinden will, ob ein Individuum gerade geht, sitzt oder läuft, dann wendet man einen Klassifikator an, der aus den Eingabedaten einen Zahlenwert aus {1,2,3} zuordnet. Diese Zahlen sind dann homomorph auf {Gehen,Sitzen,Laufen} abbildbar, somit kann der Klassifikator erkennen, ob die Daten eine gehende, sitzende oder laufende Datenquelle, also vermessene Person beschreibt.

Man kann allerdings auch den Messwerten einer stehenden Person die reelle Zahl 0.0, einer rennenden die Zahl 1.0 zuordnen. Dazu wird der Klassifikator auf einen kontinuierlichen Bildbereich abgebildet. Wenn die ausgewertete Person geht, sollte der Klassifikator einen niedrigen Wert zwischen 0.0 und 1.0 ausgeben, wenn sie joggt einen höheren als den für das Gehen.

Solche Klassifikatoren können manuell entwickelt werden. Dazu schaut man sich die zu klassifizierenden Daten an und versucht, die Eigenschaften zu finden, die die Klassen voneinander unterscheiden oder zu ihrer Erfüllung beitragen.

Allerdings gibt es auch die Möglichkeit die Eigenschaften und sogar die Klassen von automatischen Verfahren entdecken zu lassen. Diese werden in Abschnitt 2.5 beschrieben.

Die Datenauswertung mit Klassifikatoren beschränkt sich jedoch nicht auf die Auswertung von Menschen, wie der Begriff *Human Activity Recognition* annehmen lässt - frei übersetzt mit Erkennung der Aktivität eines Menschen. Grundsätzlich sind alle Datensätze, wenn sie sich in Relation mit dem zu erkennenden Vorgang oder Zustand befinden, auf diese Auswertbar. Meistens muss man sich mit einer Dämpfung auseinandersetzen: trägt das Vermessene Subjekt die Sensoren nicht direkt an sich, wie es bei dem Vermessen eines Menschen mit einem Smartphone in einer Handtasche der Fall ist, so nimmt die Sicherheit, mit der die Klassifizierung richtig liegt, im Allgemeinen ab.

Für eine tiefergehende Einfürung in das Thema Human Activity Recognition sei an dieser Stelle auf den Artikel [LL13] von Lara und Labrador hingewiesen.

[3] Human Activity Recognition

2.4. Road Quality Assessment

Road Quality Assessment bezeichnet die Idee, von mobilen Sensordaten wie zum Beispiel Beschleunigungssensoren, Lagesensoren und GPS-Sensoren Rückschlüsse auf die Straßenqualität zu ziehen. Das unmittelbare Ziel dieser Datensammlung ist die automatische Erkennung, Evaluierung und Klassifizierung von Anomalien wie zum Beispiel Schlaglöchern. Hierzu müssen die Daten während der Fahrt aufgenommen werden, dann durch verschiedene Filterverfahren nachbearbeitet und bereinigt werden, bevor sie letztendlich der Klassifizierung übergeben werden können. Das Ziel dieser Unternehmungen ist eine objektive und flächendeckende Evaluation der Straßenqualität eines großflächigen Gebietes ohne zusätzlichen Personalaufwand.

Die Aufnahme der Sensordaten erfolgt durch ein oder mehrere Smartphones, die im betreffenden Fahrzeug angebracht werden. Dabei werden die Aufnahmegeräte fest im Fahrzeug fixiert, um möglichst direkte Ergebnisse zu erhalten, und mögliche Fehlerquellen wie Bewegung des Gerätes, oder Verfälschung der Daten durch menschliches Einwirken zu minimieren. Sobald das Fahrzeug auf eine Strassenanomalie trifft, schlägt sich dies in den aufgenommenen Sensordaten nieder.

Bevor die Daten klassifiziert werden können, müssen sie durch unterschiedliche Filterverfahren nachbearbeitet werden. Zu diesen Filterverfahren gehören Tiefpassfilter, um gegebenenfalls störende Vibrationen herauszufiltern, und das beseitigen von Fehlerwerten und kleinen Ausschlägen die das Ergebnis verfälschen könnten. Dies resultiert in einer Menge von Sensordaten, die dann abhängig von ihren Ausschlagsmustern und Ausschlagsstärke klassifiziert werden.

Die Klassifizierung der Events wurde in vergangenen Arbeiten [V.+12], [Eri+08] und [Med+11] mit verschiedenen Klassifizierungsalgorithmen durchgeführt, die auf die gefilterten Daten angewandt werden. Oft wird ein Grenzwert definiert der, sobald der Accelerometerwert diesen Wert überschreitet, als Event markiert wird. Abhängig von den Projekten wurden aber auch Varianzen in Intervallen, Ruckbewegungen, oder Schwebemomente betrachtet.

2.5. Machine Learning

Machine Learning beschreibt einen Prozess bei dem ein Computerprogramm eine bestimmte Aufgabe lernt und das vorhergesagte Ergebnis sich mit zunehmender Erfahrung verbessert [Mit97]. *Machine Learning* wird oft mit *Data Mining* gleichgesetzt. Beide Bereiche überlappen sich in vielen Punkten. In beiden Bereichen können gleiche mathematischen Methoden, beispielsweise zur Klassifizierung, verwendet werden. Beim *Data Mining* liegt der Fokus darauf neue, vorher unbekannte Eigenschaften zu finden. *Machine Learning* hingegen legt den Fokus darauf bekannte Eigenschaften anhand von gegebenen Trainingsdaten vorherzusagen.

Um gute Ergebnisse mittels *Machine Learning* zu erzielen, werden zunächst Trainingsdaten benötigt. Die Qualität dieser Daten hat starken Einfluss auf die spätere Qualität des Klassifikators. Eine große Menge an sauber gesammelten Trainingsdaten bedeuten

ausreichende Qualität. Liefert ein Klassifikator unzureichend gute Ergebnisse, kann dies an den Trainingsdaten liegen. Neue Trainingsdaten und ein daraus erzeugtes neues Modell können zu besseren Klassifizierung führen. Die Vorverarbeitung von Daten kann zusätzlich auch die Qualität und auch die Geschwindigkeit eines Klassifikators erhöhen, indem Korrelation zwischen einzelnen Attributen frühzeitig erkannt und entfernt werden können. Das daraus resultierende Modell kann dann mit weniger Attributen schneller zu equivalenten Ergebnissen führen. Beim Trainieren des Klassifikators mittels Trainingsdaten muss darauf geachtet werden, dass der Klassifikator nicht zu stark an die Trainingsdaten angepasst wird. Die als *Overfitting* bezeichnete Problematik sorgt dafür, dass der Klassifikator auf den Trainingsdaten bestmögliche Ergebnisse erzielt, aber auf neuen Daten zu Fehlklassifizierungen führt, da der Klassifikator zu stark auf die Trainingsdaten angepasst ist. Verschiedene Methoden ermöglichen diese Problematik zu umgehen. Indem ein zusätzlicher Testdatensatz verwendet wird kann der Klassifikator bestmöglich angepasst werden, sodass auf dem Testdatensatz eine möglichst hohe Genauigkeit erreicht wird. Sind die Datensätze nicht groß genug für einen eigenen Testdatensatz, kann man Kreuzvalidierung verwenden. Hierbei wird der Trainingsdatensatz in n gleichgroße Teile aufgeteilt. $N-1$ Datenteile werden jeweils als Trainingsdatensatz verwendet und einer als Testdatensatz. Dies wird n mal gemacht, mit jeweils einem neuen Testdatensatz. Im Anschluss daran wird das Model für den Klassifikator zusammengeführt. Kreuzvalidierung führt nicht zwingendermaßen zu guten Ergebnissen. Sind genug Daten vorhanden, wird immer die Verwendung eines eigenen Testdatensatzes empfohlen.

Beim *Machine Learning* unterscheidet man zwischen zwei Arten des Lernens: dem *überwachtem Lernen*, und dem *unüberwachtem Lernen*. Zusätzlich gibt es noch Abwandlungen, wie z.B. das *halbüberwachte Lernen* bei dem der Lernvorgang nicht explizit überwacht wird. Beim Überwachten Lernen wird der Klassifikator anhand von ausgewählten bereits klassifizierter Trainingsdaten erzeugt. Beim *unüberwachten Lernen* sind die Daten nicht klassifiziert. Hierbei versucht der Klassifikator anhand von Ähnlichkeiten Muster zu erkennen. Ein Beispiel für das *unüberwachte Lernen* ist die Segmentierung. Hierbei werden Daten anhand von erkannten Mustern in Gruppen aufgeteilt. Ein Beispiel für *überwachtes Lernen* sind Entscheidungsbäume, *Support Vector Machines* oder *Naive Bayes*.

Entscheidungsbäume

Mittels Entscheidungsbäumen lassen sich Daten anhand von einfachen Regeln in Klassen einteilen. Der große Vorteil eines Entscheidungsbaumes ist, dass dieser sich einfach und übersichtlich für einen Anwender visualisieren lässt Ziel ist es am Ende eines Entscheidungsbaumes möglichst reine Blätter mit nur einer Klasse zu haben. Um Overfitting zu vermeiden kannein komplett erzeugter Baum zurückgeschnitten werden. Zusätzlich kann auch bei Entscheidungsbäumen mittels seperatem Testdatensatz oder Kreuzvaldierung die Genauigkeit für neue Datensätze erhöht werden. Der bekannteste Algorithmus zum Erstellen eines Entscheidungsbaumes, der auch von WEKA in Form des J48 verwendet wird, ist der $ID3 Algorithmus$ von Ross und Quinlan aus dem Jahre 1986. Der ID3 Algorithmus ist ein iterativer Top-Down Greedy Algorithmus. Er ist schnell und liefert

ein gutes Ergebnis. Der Alogirthmus garantiert jedoch nicht, dass es keinem besseren Entscheidungsbaum für die Problemstellung gibt [Qui93].

Das Attribut, das durch einen Split den größten Informationsgewinn liefert, wird für den nächsten Split als Knoten ausgewählt. Für jedes restliche Attribut wird dieser Vorgang wiederholt, bis ein eindeutiges Blatt übrig bleibt, oder keine weiteren Attribute vorhanden sind. Sollte es keine weiteren Attribute mehr geben, wird die Klasse zugeordnet, die die Mehrheit im Blatt besitzt. Der exemplarische Entscheidungsbaum in Abbildung 2.2 beschreibt ein Beispiel, ob eine Person Tennis spielen gehen wird. Das Attribut *Outlook* liefert durch einen Split den höchsten Informationsgewinn und wird somit als Wurzelknoten verwendet. Alle Blätter des Baumes haben ein eindeutiges Ergebnis und liefern einen für Menschen gut lesbaren Baum.

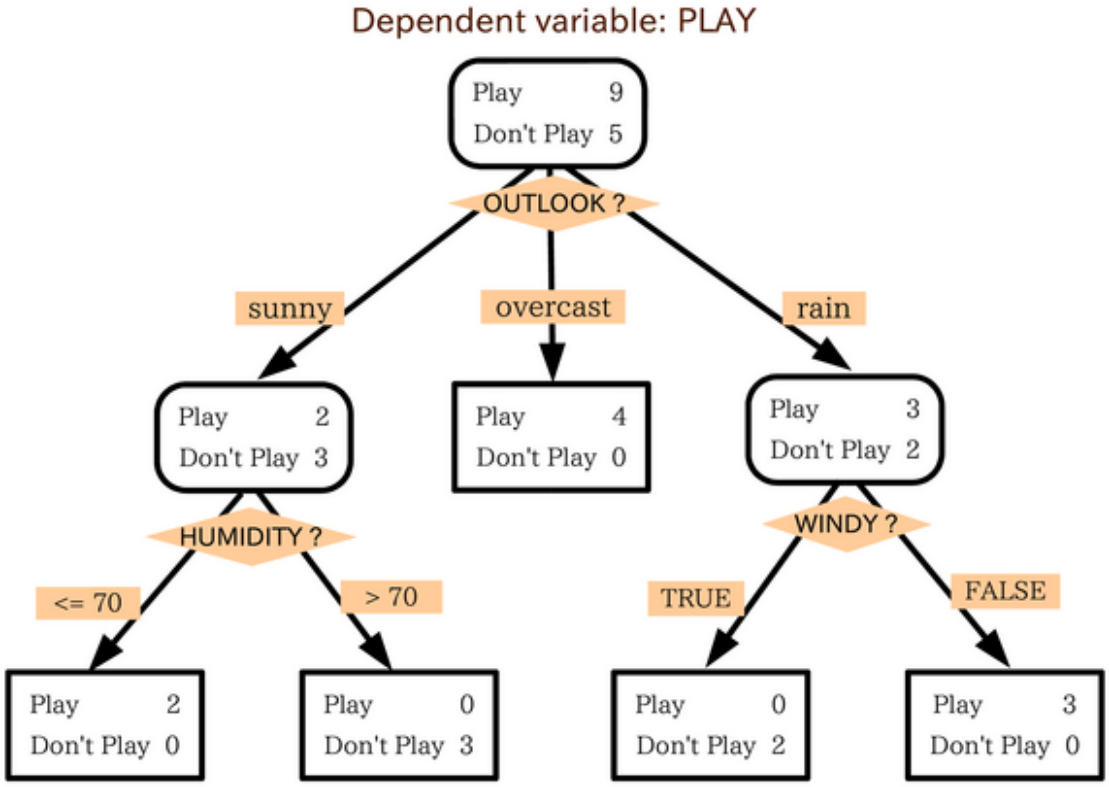

Abbildung 2.2.: Einfacher Entscheidungsbaum

3. Organisation

Im Folgenden wird näher auf die Organisation des Projekts eingegangen. So wird zum einen erläutert, wie das Team aufgeteilt wurde, um eine gerechte Aufgabenverteilung zu gewährleisten. Zum anderen werden die Werkzeuge zur Zusammenarbeit und Organisation des Projekts beschrieben.

3.1. Gruppenaufteilung und Treffen

Um die Vielfalt der unterschiedlichen Aufgaben zu bewältigen, war es von Nöten das Team in entsprechende Gruppen aufzuteilen. Dadurch haben sich drei Gruppen mit jeweils drei Mitgliedern ergeben, die sich jeweils auf eine der drei Komponenten - Mobile Client, Server und Klassifizierung - konzentriert haben. Darüber hinaus arbeiteten die Master-Studenten an dem Thema des Forschungspraktikums: Energieeffizienz-Analyse. Zusätzlich wurden zwei Projektleiter bestimmt, um das Projekt einerseits zu verwalten und andererseits den Überblick über das Projekt zu bewahren und es in die gewünschte Richtung zu lenken.

In den wöchentlichen Treffen mit den Betreuern wurden Fortschritte präsentiert, Fragen an die Betreuer gestellt und mit diesen über die präsentierten Fortschritte diskutiert. Darüber hinaus haben sich die einzelnen Gruppen in den, für sich selbst festgelegten, Intervallen getroffen, um an ihren Aufgaben zu arbeiten. Alle zwei Wochen fand ein Treffen der kompletten Gruppe ohne Betreuer statt, in denen die bisher erzielten Ergebnisse rekapituliert und gemeinsam neue Ziele/Meilensteine gesetzt wurden.

3.2. Werkzeuge

Das Problem einen Termin zu finden, an dem jedes Gruppenmitglied Zeit hat, ist für jede Gruppenarbeit eine kleine Herausforderung. Ebenso stellt das gemeinsame Arbeiten an Präsentationen oder Dokumenten eine Hürde dar. Im Folgenden werden Werkzeuge näher beleuchtet, die den Organisationsaufwand dezimiert und das gemeinsame Arbeiten effizienter gestaltet haben.

3.2.1. Trello

Trello[1] ist eine Webbasierte Plattform von Fog Creek Software zur Organisation von Projekten auf Kanban Basis. Die Website stellt die Möglichkeit bereit, Projekte in Boards zu unterteilen. Jedes Board stellt verschiedene Listen, für verschiedene Bereiche des

[1] http://www.trello.com

Entwicklungsprozesses zur Verfügung, durch die Aufgaben anhand von Karten geführt werden. Die reduzierteste Variante, die auch während des Projektes am häufigsten genutzt wurde, ist die Unterteilung in To Do, Doing und Done. Jede Gruppe hatte ein eigenes Board, in denen zusätzliche Termine vereinbart, Dokumente ausgetauscht, Probleme notiert und Ideen ausgestauscht werden konnten. Im Board Termine wurden für jeden einzuhaltende Termine vermerkt.

3.2.2. Google Drive

Google Drive ist ein webbasierter Dienst, der es erlaubt, Dokumente zu erstellen, die von mehreren Personen eingesehen und auch gleichzeitig bearbeitet werden können. Insbesondere die Möglichkeit sein Feedback sofort selber umzusetzen, stellte sich als besonders hilfreich heraus. Um den Dienst nutzen zu können muss ein Google+ Account bei dem Ersteller des Dokumentes vorhanden sein, alle weiteren Personen können über das Versenden eines Links uneingeschränkt darauf zugreifen.

3.2.3. Doodle

Für die Terminfindung wurde *Doodle* [2] verwendet. Doodle bietet die Möglichkeit mehrere Termine zur Auswahl zu stellen und die Nutzer tragen sich ein, an welchen Terminen sie können. Der Vorteil von Doodle, zugleich der Grund warum es für spontane Termine zum Einsatz kam, ist, dass eine Terminliste in kürzester Zeit erstellt und eine Registrierung nicht nötig ist.

3.2.4. GitHub

Eine Versionsverwaltung ist ein essenzielles Werkzeug der Softwareentwicklung und dient unteranderem der Protokollierung, Wiederherstellung und Archivierung von Projekten. Zum Anfang der Projektphase wurde SVN als das Versionsverwaltungssystem eingerichtet und verwendet, aufgrund der Vertrautheit durch den universitären Einsatz. Nach kürzester Zeit erfolgte der Umstieg auf git und damit *GitHub* [3]. Der Grund dafür liegt an den Organisationsmöglichkeiten die GitHub zur Verfügung stellt, z.B. Wiki, issue tracking, collaborative code review.

[2] http://doodle.com
[3] http://github.com

4. Architektur

In diesem Kapitel wird die Architektur des MobileSensors-Systems erklärt. Das System gliedert sich grob in drei Komponenten (Abbildung 4.1):

- einer Applikation für mobile Endgeräte, welche (Sensor-) Daten sammelt und Auswertungen anzeigt
- einer Server-Komponente zum verwalten und verarbeiten der Daten
- und einer Auswertungskomponente (`MobileSensors.jar`)

Abbildung 4.1.: MobileSensors Komponenten (UML)

In den folgenden drei Abschnitten wird der strukturelle Aufbau jeweils einer der genannten Komponenten erklärt. Dazu werden zunächst die *Stakeholder* [1] und *Anforderungen* [2] einer Komponente definert und deren jeweiligen Auswirkungen auf die Beziehungen und Hierarchien der einzelnen Bauelemente untereinander herausgearbeitet.

4.1. Mobile

In diesem Kapitel wird die Architektur der mobilen Komponente abgehandelt. Dazu wird auf die grundlegenden Funktionalitäten des Programms eingegangen, die verwendete Systemschnittstelle beschrieben und der Anschluss an die Serverkomponente besprochen. Da das Programm auch vom Endbenutzer gesteuert wird, erklärt das Kapitel auch die Verbindung einer Endbenutzeroberfläche mit dem System.

[1] Stakeholder: Gruppen von Nutzern, die in irgendeiner Form mit der Komponente konfrontiert werden und daher Interessen bezüglich Funktionalität und Qualität haben.

[2] Anforderungen: konkrete Interessen von Stakeholdern bezüglich Funktionalität und Qualität. Anforderungen werden daher oft in zwei Kategorien unterschieden:

- *funktionale* Anforderungen: Forderung eines bestimmten Funktionsumfangs oder einer bestimmten Funktionalität
- *nichtfunktionale* Anforderungen: Forderung einer messbaren Qualität einer Funktion oder eines Systems.

4.1.1. Komponenten des Kollektors

Der Zweck des Kollektors ist, die Sensordaten, die das Betriebssystem freigibt, aufzunehmen und zu versenden. Dazu werden die von der Android API bereitgestellten Sensorereignis-Hörer wie folgt angesteuert: die aus dem Kernel des Betriebssystems abgehörten Sensordaten werden von Android verpackt und dadurch den Programmen der Entwickler zugänglich gemacht. Um diese verpackten Datensätze zu erhalten, macht der Programmierer dem Betriebssystem eine Anlaufstelle bekannt, welche eine Ereignisfunktion bereitstellt. Die Anlaufstelle wird vom System dann verwendet, indem die definierte Funktion mit dem jeweiligen Sensor-Datum aufgerufen wird. Ab dann kann das Programm damit weiterarbeiten.

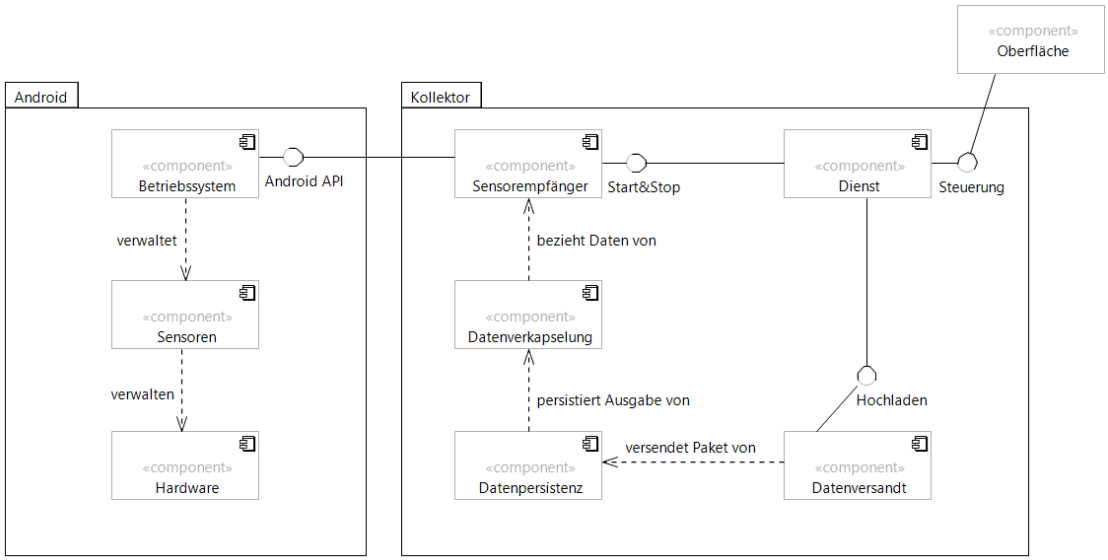

Abbildung 4.2.: Komponenten des Kollektors

Im Kollektor werden hierzu Anlaufstellen für alle von der API bereitgestellten Sensortypen aufgesetzt. Damit im Ruhezustand nicht unnötig Energie verbraucht wird, werden die Anlaufstellen vom Dienst je nach Bedarf gestartet und gestoppt. Im gestoppten Zustand werden keine Daten vom System für die Stellen produziert.

Da eine essentielle Funktionalität des Kollektors der Versand von Sensordaten ist, werden diese für den Transport zum Server verkapselt. Die Ausgabe der Verkapselung wird durch die Persistenzkomponente im Dateisystem des Smartphones gespeichert, um danach mithilfe des Datenversands an den Server weitergegeben zu werden. Auch der Vorgang des Hochladens wird durch den Dienst gesteuert, da nur der Dienst das Wissen über Anfang und Ende einer Aufnahme hat.

Die Anforderungen an die mobile Komponente lassen sich in die folgenden funktionalen und nichtfunktionalen Anforderungen aufteilen und den Stakeholdern zuordnen, welche diesen vorgestellt sind.

4.1.2. Stakeholder

Anwender

Die Anwender sollen das Programm auf einem Smartphone verwenden und die Daten für die Klassifikation und die Auswertung bereitstellen.

App-Entwickler

Die Entwickler der Applikation für das mobile Endgerät sind maßgebend für die Komponente. Sie setzen die Datenaufnahme und den Versand um und erstellen das vom Anwender benutzte Programm.

Server-Entwickler

Die Entwickler des Servers sind über die Relation der Komponenten *MobileClient* und *Server* aus der Abbildung 4.1 am Entwicklungsprozess beteiligt; im speziellen ist damit das Austauschformat der Daten gemeint.

Klassifikator-Entwickler

Da die Klassifikation Daten von ausgewählten Quellen in einer bestimmten Form und einem bestimmten Umfang brauchen, sind die Entwickler der Klassifikation als Anforderungssteller auch Stakeholder des Kollektors.

4.1.3. Anforderungen

Funktionale Anforderungen

1. **Datenaufnahme**
   Daten sollen aufgezeichnet werden.
   Stakeholder: Klassifikator-Entwickler, App-Entwickler
   1.1. **Aufnahme der aktuellen Position**
   GPS-Informationen sollen aufgezeichnet werden.
   1.2. **Aufnahme von Sensorendaten**
   Die Ausgabe der Umgebungssensoren soll gespeichert werden.
   1.3. **Annotieren von Daten**
   Aufgezeichnete Daten sollen an bestimmten Stellen Benutzereingaben als Text enthalten.
2. **Daten an Server senden**
   Die gesammelten Daten sollen über das Internet an den Server gesendet werden.
   Stakeholder: Server-Entwickler, App-Entwickler
3. **Informationen während der Fahrt anzeigen**
   Der Nutzer soll während der Fahrt Zugriff auf eine Karte seiner Umgebung haben,

sowie Geschwindigkeitsdaten und die Länge der gefahrenen Strecke.
Stakeholder: Anwender

Nichtfunktionale Anforderungen

1. **Einfache Bedienbarkeit**
   Schnelle und handliche Bedienbarkeit auch während der Fahrradfahrt.
   Stakeholder: Anwender, Klassifikator-Entwickler

2. **Anwender-Nutzen**
   Die Benutzung des Programms soll dem Anwender einen Vorteil bringen.
   Stakeholder: Anwender

3. **Dateneffizienz**
   Die Daten sollen effizient gespeichert und versandt werden.
   Stakeholder: Server-Entwickler

4.1.4. Architektur der Benutzerschnittstelle

Der Dienst, der die Funktionen des Programms realisiert, ist durch die Schnittstelle *Steuerung* der Oberfläche exponiert. Letztere muss sinnvolle Ausgaben an den Benutzer machen und die Eingaben ebendieses interpretieren.

Um den Rückweg von Dienst an Benutzerschnittstelle zu sparen, werden die Daten, die nach der Anforderung 3 angezeigt werden sollen, über sekundäre Wege aufgegriffen. Da die Daten nur erforderlich sind, wenn gleichzeitig die Aufnahme läuft, können die Ereignis-Hörer auch in der Oberfläche verwendet werden, ohne das Betriebssystem unnötige Sensorwerte produziert.

Somit entfällt auf die Architektur der Oberfläche nur die Instantiierung und Steuerung der Grafik. Die Architektur-Muster werden vom Android Development Kit in Form von Activities und XML-Layouts vorgegeben.

4.2. Server

4.2.1. Anforderungen and die Server Software

Der Server hat vielseitige Anforderungen. Im groben ist er zuständig für:

- Benutzersystem mit Registrierung, Authentifizierung und unterschiedlichen Rollen
- Bereitstellung eines Webservice zur Annahme von Sensordaten über authentifizierte Clients
- Persistierung von Sensordaten
- Darstellung von Sensordaten
- Schnittstelle für Daten-Export
- E-Mail Versand bei Registrierung

All diese Anforderungen benötigen festgelegte Schnittstellen und Konventionen aufgrund derer es ermöglicht wird, dass mehrere Entwickler die Software parallel umsetzen können. Um von der Erfahrung anderer Entwickler zu profitieren und um nicht die gesamte Architektur von Grund auf zu entwickeln, was den Rahmen des Pojektpraktikums schnell sprengen würde, fiel die Entscheidung ein Web Application Framework zu benutzen. Dabei wurde aufgrund des großen Umfangs an Paketen[3] und aufgrund der relativ leichten Erlernbarkeit der Sprache Ruby das Framework Ruby on Rails zur Umsetzung gewählt.

4.2.2. Architektur des Servers

Die Architektur der Server-Software ist in Abbildung 4.3 dargestellt. *Mobile Clients* und *Web Clients* kommunizieren über den Web Server mit der Rails Applikation. Die Architektur der Rails Applikation ist stark geprägt durch das *Model-View-Controller Pattern*. Über den Dispatcher wird die Route des Requests und dadurch sein zugehöriger Controller und Action bestimmt. Der Controller selbst entscheidet, was in dem Request geschieht. Es können über Active Record Daten gelesen, oder geändert werden. Je nachdem, ob der Webservice genutzt wird, oder einfach eine Webseite im Browser angezeigt werden soll, ruft der Controller eine entsprechende *View* auf, welche über die Response an die Endgeräte zurück geht.

[3] http://rubygems.org/

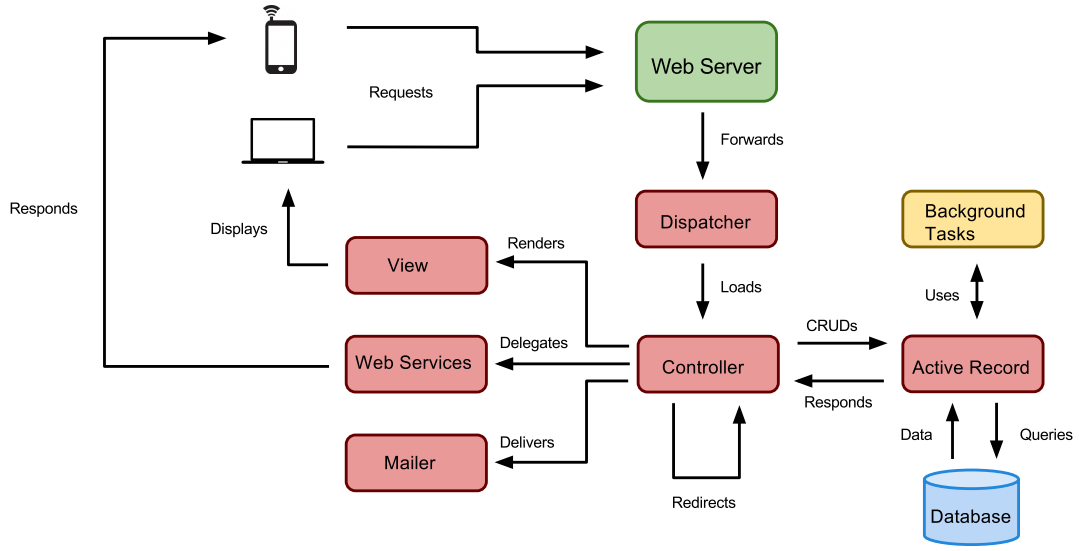

Abbildung 4.3.: Server Software Architektur

4.2.3. Ruby on Rails

Ruby on Rails [4] ist ein von David Heinemeier Hansson entwickeltes quelloffenes Web Application Framework. Die erste Release fand im Jahre 2004 statt. Die aktuelle Version lautet 4.1.0. Der Server des Projektpraktikums ist umgesetzt in der Versin 4.0.0. *Ruby on Rails* bedient sich mehrerer Maxime und Techniken der Softwareentwicklung. Die wichtigsten dabei sind:

- Konvention vor Konfiguration
- Don't repeat yourself (DRY)
- Model View Controller (MVC)
- Representational state transfer(RESTful)

Konvention vor Konfiguration

Konvention vor Konfiguration soll unnötige Konfigurationsdateien vermeiden und dem Entwickler Schreibarbeit und Verwaltungsaufwand abnehmen. So werden beispielsweise alle Tabellen in der Datenbank mit dem Primärschlüssel ID vom Typ Integer belegt. Eine weitere Konvention wäre, dass Template-Variablen den selben Bezeichner tragen wie ihre zugehörigen Bezeichner im Controller.

[4] http://rubyonrails.org/

Don't repeat yourself (DRY)

Die Idee, die hinter *DRY* steckt, ist, jegliche Redundanzen im Quellcode zu vermeiden. Dies erhöht die Lesbarkeit und die Wartbarkeit des Quellcodes ungemein. *Ruby on Rails* bietet hierzu bereits mehrere Lösungen an. So werden *Helper* für *Controller* und *View* angeboten, die immer wiederkehrende Aufgaben in kurzen Funktionsausrufen unterbringen. In der *View* gibt es weiterhin die Möglichkeit *Partials* zu benutzen, um sich wiederholende HTML-Strukturen zu vermeiden. Bei den *Models* dienen *Mixins* zur Vermeidung von Redundanzen. Hier können Funktionen sich ähnelnder *Models* in ein *Mixin* zusammengezogen werden.

4.2.4. MVC

Das Softwarepattern *Model View Controller* ist eine grundlegende Schichtentrennung. Sie ist der Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit des Quellcodes dienlich. Den einzelnen Schichten kommen dabei unterschiedliche Aufgaben zu, welche im folgenden erklärt werden.

Model

Das *Model* stellt das Rückgrat einer jeden *Ruby on Rails* Applikation dar. Der Zweck des Models ist die Persistierung sowie die Abfrage der durch die Applikation erzeugten Daten. In *Ruby on Rails* gibt es mehrere Möglichkeiten, Daten zu speichern. Für die Anforderungen des Projektpraktikums kam die Datenbanksoftware *PostgreSQL* in Frage. Dabei wird der objektrelationale *Mapper Active Record* für die Kommunikation mit der Datenbank verwendet. Datenbank-Tabellen entsprechen hier Klassen, Tabellen-Spalten entsprechen Klassen-Attributen. Eine Zeile einer Datenbank-Tabelle entspricht hier einer Instanz der Klasse. So hantiert man im Quellcode statt mit Arrays oder einzelnen Variablen mit Listen von Objekten, welche die Daten repräsentieren. Auch Datenbank-Abfragen werden erleichtert, leserlicher, und sicherer, da man *SQL* nur noch in Ausnahmefällen selbst schreibt. Dies verhindert SQL-Injections und ermöglicht es, die Datenbank (welche beispielsweise einen anderen SQL-Dialekt besitzt) im Hintergrund bei Bedarf auszutauschen. Relationen und Beziehungen werden im *Model* definiert, wodurch die Daten bei der Erzeugung direkt validiert werden können. Darüber hinaus werden Joins direkt über die Benutzung von Objekten realisiert. Die Löschung von Daten geschieht durch die festgelegten Beziehungen auf Wunsch tabellenübergreifend. Die Arbeit für alle Aufgaben Queries zu schreiben bleibt dem Entwickler somit erspart.

Das Datenbank-Schema wird über Migrations gewartet. So werden Tabellen und Spalten über auto-generierte Migrations-Dateien erzeugt, umbenannt oder gelöscht. Dies vereinfacht das Deployment von den Entwicklungs-Umgebungen in die Produktiv-Umgebung. Des weiteren ist es möglich, den Entwicklungsstand der Datenbank zu jedem Zeitpunkt wieder herzustellen. Die Benutzung von Migrations ermöglicht es weiterhin auf einfache Art und Weise mehrere Entwicklungszweige zu erstellen sowie diese wieder zusammenzuführen.

View

Die *View* ist in *Ruby on Rails* zuständig für das Aussehen des HTTP-Bodys. Dabei werden unter anderem folgende Ausgabeformate unterstützt:

- HTML
- XML
- JSON
- CSV
- JavaScript
- ...

Für das Rendern der verschiedenen Ausgabeformate dienen unter anderen folgende Templatesysteme:

- ERB
- HAML
- SASS
- CoffeeScript
- ...

Die verschiedenen Templatesysteme bieten unterschiedliche Möglichkeiten, die Ausgabeformate zu rendern. So können vereinfachte Auszeichnungssprachen wie *HAML*, *SASS* oder *CoffeeScript* verwendet werden, um die *View* zu rendern. Die Sprachen sind darauf ausgelegt, die Ausgabelogik übersichtlich zu gestalten und Syntaxfehler zu vermeiden. Es gibt allerdings auch deskrtiptive Sprachen, die automatisiert *XML* oder *JSON* Ausgaben erzeugen können.

Controller

Der *Controller* ist in *Ruby on Rails* die Steuerungsschicht. Der Controller dient dazu, das *Model* und die *View* zu verbinden. In ihm werden Datenbankoperationen ausgeführt, Parameter weitergereicht, die Session verwaltet, Redirects durchgeführt und die *View* gerendert. Es steuert somit den Ablauf der Sofware und kann verschiedene *Actions* enthalten. Eine *Action* wird aufgrund von in der Konfiguration festgelegten Routen vom *Dispatcher* aufgerufen. Per Konvention besitzt jede *Action* eine eigene *View*. Sie kann aber durchaus mehrere *Views* besitzen, wenn sie unterschiedliche Ausgabeformate unterstützt.

4.2.5. RESTful

Ruby on Rails hält sich an die Prinzipien des Representational State Transfer. REST steht die Idee zu Grunde, dass jede URL einer Anwendung eine feste Repräsentation von Objekten der gehaltenen Daten wiederspiegelt. Einen solchen Dienst nennt man eine Ressource. Somit ist jede Ressource addressierbar. Die Repräsentation der Ressource ist frei wählbar. So kann Darstellung beispielsweise in *HTML, JSON*, oder *XML* geschehen. Des weiteren sollte jede Aktion zustandslos sein. Das heißt der Client und der Server finden alle benötigten Informationen im Request beziehungsweise in der Response wieder. Weiterhin ist jeder HTTP-Request gebunden an eine der vier Operationen

- GET
- POST
- PUT
- DELETE

Durch die Festlegung der Operation wird angegeben, was der Request in der Anwendung bewirkt. Soll eine Ressource angezeigt werden, wird die Operation GET benutzt. Soll eine Ressource angelegt werden POST. Änderungen werden mit PUT realisiert und für das Löschen von Ressourcen gibt es die Operation DELETE.

4.2.6. Background-Tasks

Die Rails Applikation läuft produktiv über einen Apache[5] mit Phusion Passenger[6]. Technisch bedingt ist jeder Request über einen 30 Sekunden Timeout limitiert. Werden über einen Request aufwändigere Aufgaben erledigt und somit der Timeout überschritten, wird die Aufgabe nicht zu Ende gebracht. Diese Problematik kommt schnell bei dem Datenimport zu tragen. Hier ist es notwendig, die Aufgabe im Request asynchron aufzurufen. Hier wurde sich für die Library Delayed::Job[7] entschieden. Parallel zu dem *Apache Web Server* läuft ein *Daemon* im *Rails Environment*, welcher aufwändigere Aufgaben erledigen kann. Solche asynchronen Aufrufe geschehen nahtlos direkt aus dem *Model* oder dem *Controller*.

[5] http://httpd.apache.org/
[6] https://www.phusionpassenger.com
[7] https://github.com/collectiveidea/delayed_job

4.3. Klassifizierung

Im folgenden Abschnitt soll die Architektur der Java-Komponente zur Ereigniserkennung/-klassifizierung beschrieben werden.

Dazu werden zunächst Stakeholder und Anforderungen definiert und anschließend aus den Perspektiven der Stakeholder interpretiert, um so Entwurfs- und Design-Entscheidungen zu generieren und zu erläutern.

4.3.1. Stakeholder

Externe Entwickler

Externe Entwickler sind Endnutzer der Klassifizerungskomponente. Sie verwenden sie zur Entwicklung anderer Kompontent. Sie betrachten die Komponente als Blackbox und müssen daher nur einige ausgesuchte, nach außen sichtbare Funktionalitäten und deren Spezifikation kennen. Interne Abläufe interessieren sie nicht.

Interne Entwickler

Interne Entwickler warten und erweitern die Komponente selbst. Sie müssen den kompletten Aufbau und alle Abläufe der Klassifizerungskomponente kennen und verstehen.

4.3.2. Anforderungen

Funktionale Anforderungen

1. **Verwendung von Java**
   Es muss Java als Plattform verwendet werden.
   Stakeholder: Interne & Externe Entwickler

2. **Ereignisserkennung via Machine-Learning**
   Ereignisse muss mit Mitteln des maschinellen Lernens erkannt werden.
   Stakeholder: Interne Entwickler

   2.1. **Verwendung von Weka**
   Zur Umsetzung von Machine-Learning muss auf Komponenten der Weka-API zurückgegriffen werden.
   Stakeholder: Interne Entwickler

   2.2. **Ereignisserkennung auf Intervallen**
   Auf einem Intervall von Sensordaten muss ein Klassifikator entscheiden ob ein Ereigniss eingetreten ist oder nicht.
   Stakeholder: Interne & Externe Entwickler

   2.3. **Klassifikatoren trainieren**
   Klassifikatoren muss auf gegebenen Trainingsdaten trainiert werden können.
   Stakeholder: Interne & Externe Entwickler

2.4. **Klassifikatoren anwenden**
Klassifikatoren muss auf Intervallen von Sensordaten angewendet werden können.
Stakeholder: Interne & Externe Entwickler

2.5. **Klassifikatoren serialisieren**
Klassifikatoren muss serialisiert und extern gespeichert werden können.
Stakeholder: Interne & Externe Entwickler

2.6. **Klassifikatoren deserialiseren**
Gespeicherte Klassifikatoren muss deserialisiert und angewendet werden können.
Stakeholder: Interne & Externe Entwickler

Nichtfunktionale Anforderungen

1. **Einfache Erweiterbarkeit bezüglich Ereignissen**
Neu definierte Ereignisse muss sich einfach in die Komponente einpflegen lassen.
Stakeholder: Interne Entwickler

2. **Einfache Verwendbarkeit**
Die Komponente muss in anderen Komponenten einfach verwendbar sein.
Stakeholder: Externe Entwickler

4.3.3. Externe Perspektive

Hier sollen Anforderungen aus der Perspektive eines externen Entwicklers betrachtet und aus den gezogenen Schlüssen Entwurfsmuster abgeleitet werden. Wichtig durch das Einnehmen dieser Perspektive ist insbesondere die nichtfunktionale Anfoderung 2 und deren Auswirkung auf die funktionalen Anforderungen 2.2. bis 2.6., sowie die Interpretation des Wortes ”einfach”.

Allgemeiner Aufbau

Nach Anforderung 1 muss die Komponente in Java geschrieben werden, daher macht es Sinn, den Aufbau als Paket-Diagramm (Abbildung 4.4) darzustellen. Aus externer Perspektive existieren allerdings nur der oberste Namespace *MobileSensors* und dessen einziges Mitglied *MobSens*. Dies liegt an der Rolle, die externe Entwickler hier einnehmen:

Externe Entwickler müssen nur die Spezifikation der nach außen sichtbaren Funktionen kennen. Die funktionalen Anforderungen 2.2. bis 2.6. dienen aus dieser Perspektive lediglich als Korrektheitskriterien für diese Spezifikation. Die qualitative Anforderung 2 bestärkt hier die Position der externen Programmierer. An dieser Stelle soll ”einfach” so verstanden werden, dass man sich nicht mit den technischen Details aufhalten muss, sondern dass stattdessen die Bedienung der Klassifizierungskomponente durch eine vereinfachte Schnittstelle, einer vorgeschalteten *Fassade* ermöglicht wird.

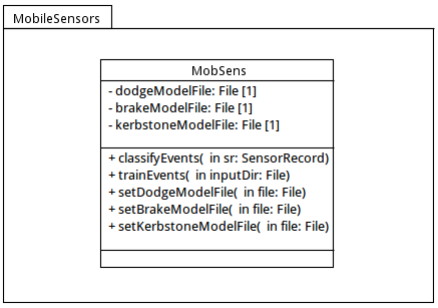

Abbildung 4.4.: Paket-Aufbau aus externer Sicht (UML)

Entwurfsmuster: Fassade

Die Fassade ist ein Strukturmuster zum erstellen vereinfachter Schnittstellen zu Systemen mit stark technischen Klassen, die von Außen eigentlich nie verwendet werden müssen. Im Fall der *MobSens-Fassade* (Abbildung 4.5) werden neben optionalen Settern die Methoden *classifyEvents* und *trainEvents* zur Verfügung gestellt. Der konkrete Ablauf des Trainierens und Klassifizieren von Ereignissen, bzw. des Serialisieren und Deserialisieren von Klassifikatoren bleibt dabei unbekannt (Anforderung 2).

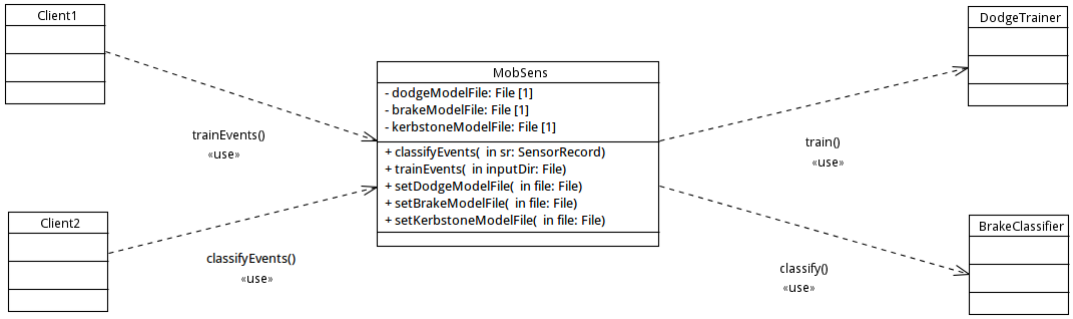

Abbildung 4.5.: MobSens-Fassade aus externer Sicht (UML)

Das Fassadenmuster hat nicht nur Vorteile bezüglich der vereinfachten programmatischen Nutzung (vergleiche hierzu das UML-Diagramm [Abbildung 4.5] mit dem reduzierten Code-Beispiel [Abbildung 4.6]), sondern auch gegenüber einer langfristigen Pflege der Komponente. Berücksichtigt man die Spezifikation der einzelnen Fassaden-Methoden, kann deren Implementation im extrem Fall komplett ausgetauscht werden, ohne die Funktionsweise der Nutzerkomponenten zu gefährden.

4.3.4. Interne Perspektive

Zur Erinnerung: Interne Entwickler und Programmierer sollen die Klassifizierungskomponente warten und erweitern. Sie müssen daher den kompletten Aufbau und alle Abläu-

```
//Ereignisse trainieren:

import MobileSensors.MobSens;

...

File fileToBrakeModel = ...      /* optional */
File fileToDodgeModel = ...      /* optional */
File fileToKerbstoneModel = ... /* optional */
File inputDir = ...

MobSens mobs = new MobSens();

mobs.setBrakeModelFile(fileToBrakeModel);          /* optional */
mobs.setDodgeModelFile(fileToDodgeModel);          /* optional */
mobs.setKerbstoneModelFile(fileToKerbstoneModel); /* optional */
mobs.trainEvents(inputDir);
```

```
//Ereignisse klassifizieren:

import MobileSensors.MobSens;

...

File fileToBrakeModel = ...      /* optional */
File fileToDodgeModel = ...      /* optional */
File fileToKerbstoneModel = ... /* optional */
SensorRecord sr = ...
ArrayList<Event> result = ...

MobSens mobs = new MobSens();

mobs.setBrakeModelFile(fileToBrakeModel);          /* optional */
mobs.setDodgeModelFile(fileToDodgeModel);          /* optional */
mobs.setKerbstoneModelFile(fileToKerbstoneModel); /* optional */

result = mobs.classifyEvents(sr);
```

Abbildung 4.6.: MobSens-Fassade (Code)

fe kennen. Bezogen auf die Interpretation der Anforderungen aus ihrer Perspektive hat das zur Folge, dass hier alle Anforderungen, funktionale und nichtfunktionale, bedacht werden müssen. Allerdings existieren auch aus dieser Sicht zwei Anforderung, die besonderen Einfluss auf Designentscheidungen haben. Einerseits die qualitative Forderung der einfachen Erweiterbarkeit (Anforderung 1), andererseits die funktionale Forderung der Verwendung von *Weka* [8] (Anforderung 2.1.). Da es sich hierbei nicht nur um ein Data-Mining-Werkzeug mit Benutzeroberfläche, sondern ebenfalls um eine eigenständige Data-Mining Java-API mit eigenen Architekturmustern handelt, die wiederum Einfluss auf die Architektur der Klassifizierungskomponente nehmen.

Allgemeiner Aufbau

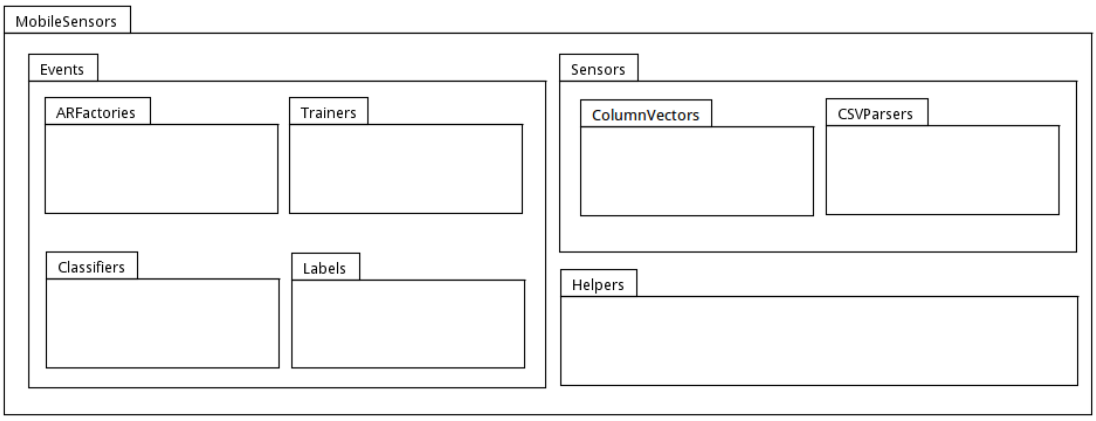

Abbildung 4.7.: Paket-Aufbau aus interner Sicht (UML)

Auch aus der internen Perspektive fordert Anforderung 1 die Nutzung von Java als Laufzeit-Umgebung, daher wird auch hier Aufbau der Klassifizierungskomponente als Paket-Diagramm (Abbildung 4.7) dargestellt. Allerdings wird hier zunächst nur die Aufgliederung in Subpakete erklärt.

Die Komponente zerlgegt sich in drei Hauptpakete:

- *Helpers*
  Enthält Helferklassen für statistische Berechnungen und zur Fensterbildung.

- *Sensors*
  Enthält Klassen zur Darstellung und Verarbeitung von Sensordaten (z.B. Accelerometer). Zusätzlich existieren noch die Subpakete *CSVParsers* und *ColumnVectors*, die das Verarbeiten von Sensordaten in CSV- und "Matrix"-From [9] ermöglichen.

[8] `http://www.cs.waikato.ac.nz/ml/weka/`

[9] Listen von Java-Objekten, insbesondere Instazen von *ArrayList*, können als Matrizen interpretiert werden, wobei ein Element in der Liste einer Zeile in der Matrix entspricht. Diese sind nativ auf den

- *Events*
  Enhält Klassen zur Darstellung, Erkennung und zum Trainieren von Ereignissen, daher auch die Aufteilung in die Subpakete: *ARFactories*, *Classifiers*, *Trainers* und *Labels*.

Im folgenden soll hauptsächlich die Architektur der Subpakete *Events.ARFactories*, *Events.Classifiers* und *Events.Trainers* erläutert werden, da dort mit Weka und maschinellem Lernen gearbeitet wird. Zudem werden in den Paketen *Sensors* und *Helpers* keine komplexeren Entwurfsmuster als einfache Datenkapselung mit einfacher Vererbung (mit und ohne generische Typen) verwendet.

Entwurfsmuster: Fassade

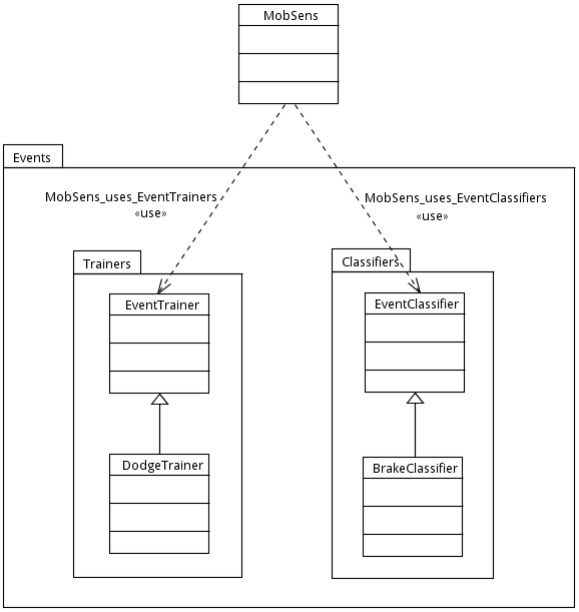

Abbildung 4.8.: MobSens-Fassade aus interner Sicht (UML)

Das Fassaden-Entwurfsmuster ist bereits aus der externen Perspektive bekannt. In Abbildung 4.5 wird beispielhaft gezeigt, wie die *MobSens*-Fassade die Klassen *DodgeTrainer* und *BrakeClassifier* verwendet, diese sind jedoch nur Erben der Klassen *EventTrainer* bzw. *EventClassifier*. Um eine höhere *Trennung der Belange* zu erreichen werden die Anforderungen 2.3. und 2.5., sowie 2.4. und 2.6. in die Subpakete *Trainers* und *Classifiers* aufgeteilt. Subklassen vom Typ *EventTrainer* kümmern sich dabei um das Trainieren und Serialisieren von Ereignisklassifikatoren, Erben vom Typ *EventClassifier* um das Deserialisieren und Anwenden von Klassifikatoren.

Zugriff auf Zeilenvektoren optimiert, im Kontext des maschinellen Lernenes wird jedoch häufig der Zugriff auf Spaltenvektoren benötigt.

Exkurs: Weka

An dieser Stelle soll kurz das Data-Mining-Werkzeug *Weka* und dessen für die Klassifizierungskomponente wichtigen Klassen vorgestellt werden:

Weka steht für *Waikato Environment for Knowledge Analysis* und ist, wie der Name schon sagt, die freie Software-Umgebung für Wissensanalyse der Universität Waikato in Neuseeland. Im Kern handelt es sich um eine Sammlung von Java-Algorithmen und -Datenstrukturen spezialisiert auf Data-Mining und maschinelles Lernen, Weka wird jedoch standardmäßig auch mit einer Benutzerobfläche für Datenanalysen zur Verfügung gestellt.

Datenstrukturen. Wichtige Datenstrukturen der Weka Bibliothek sind:

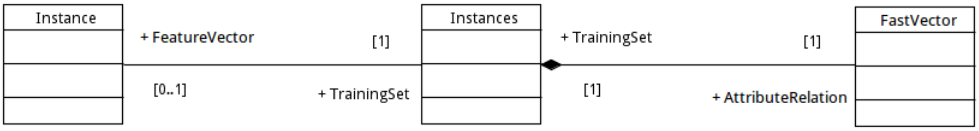

Abbildung 4.9.: Weka Datenstrukturen (UML)

- **FastVector:**
  Optimierte *Vector*-Klasse. Allgemeine Repräsentation von Tupeln aus Java-Objekten. Dient im Kontext des maschinellen Lernens als abstrakte Definition von Merkmalsvektoren (engl.: *feature vector*) als Attributrelationen.

- **Instance:**
  Repräsentiert eine konkrete Instanz eines vorher definierten Merkmalsvektors.

- **Instances:**
  Repräsentiert eine Menge konkreter Instanzen eines vorher definierten Merkmalsvektors.

Abbildung 4.10 zeigt eine übliche Verwendungsweise der Weka-Datenstrukturen *FastVector*, *Instance* und *Instances* im Machine-Learning-Kontext. Hier wird eine einfache Attributrelation definiert, instanziiert und schließlich einer Trainingsmenge hinzugefügt.

Wichtig an diesem Beispiel ist besonders die Anweisung `fv.setDataSet(trainingSet)`, denn sie verdeutlicht die bidirektionale Assoziation zwischen *Instance* und *Instances* (Abbildung 4.9). Allgemein können Werte eines Merkmalsvektors auch über einen Attributindex gesetzt werden, allerdings würden diese Indizes in größeren Applikation die Auftrittswahrscheinlichkeit von sogenannten magischen Zahlen [10] im Quelltext deutlich erhöhen. Daher ist es oft sinnvoller den Attributwert mit einer statischen Attributvariable als Index zu setzen. Hierbei wird intern ein Objektabbgleich vorgenommen, welcher überprüft

[10] Im Quelltext auftretende numerische Werte, deren Herkunft unklar (”magisch”) ist.

```
//=============================================================
//define attributes:

String label01 = "label01";
String label02 = "label02";

FastVector labels = new FastVector();
labels.addElement(label01);
labels.addElement(label02);

Attribute label = new Attribute("label", labels);

Attribute a = new Attribute("a");
Attribute b = new Attribute("b");
Attribute c = new Attribute("c");

//=============================================================
//define attribute relation:

FastVector ar = new FastVector();
ar.addElement(a);
ar.addElement(b);
ar.addElement(c);
ar.addElement(label);

//=============================================================
//define empty training set:

Instances trainingSet = new Instances("myRelation", ar);
trainingSet.setClass(label);

//=============================================================
//define concrete feature vector:

Instance fv = new Instance();
fv.setDataSet(trainingSet); //links fv to ar over trainingSet!
fv.setValue(a, 1);
fv.setValue(b, 2);
fv.setValue(c, 3);
fv.setValue(label, label01);

//=============================================================
//add feature vector to training set:

trainingSet.add(fv);
```

Abbildung 4.10.: Übliche Verwendung der Weka-Datenstrukturen

ob sich dieses Attribut in der Attributrelation befindet. Die Attributrelation wird einer *Instance*-Instanz jedoch nur über den Umweg einer *Instances*-Instanz zur Verfügung gestellt.

Algorithmen. Wichtige (konkrete) Algorithmen in Weka gibt es per se nicht, da es sehr vom Einsatzgebiet abhängt welcher Algorithmus geeignet ist. Allderings muss man wissen, dass alle Algorithmen im Kontext des maschinellen Lernens in Erben der abstrakten Klasse *Classifier* gekapselt werden:

- **Classifier:**
  Abstrakte Schnittstelle für alle numerische und nominale Vorhersagemodelle. Bietet Methoden zum Trainieren und Anwenden von Klassifikatoren. Gelernete Modelle sind Teil des Zustands einer Kindklasse.
- **J48:**
  Erbe von *Classifier*. Implementierung von des C4.5 Entscheidungsbaumgenerators in Java.

Entwurfsmuster: Abstrakte Fabrik

Abstrakte Fabrik ist ein Erzeugungsmuster, dessen Hauptzweck es ist, Herstellung verschiedenartiger Objekte zu zentralisieren. Das bedeutet der konkrete Prozess der Klasseninstanziierung wird vor Nutzerklassen der konkreten Fabrik versteckt [11] und diese müssen sich nicht mehr um die oftmals problematische Konstruktion von Objekten kümmern. Abbildung 4.11 zeigt eine vereinfachte Darstellung des *Gang of Four*-Abstrakte-Fabrik-Musters.

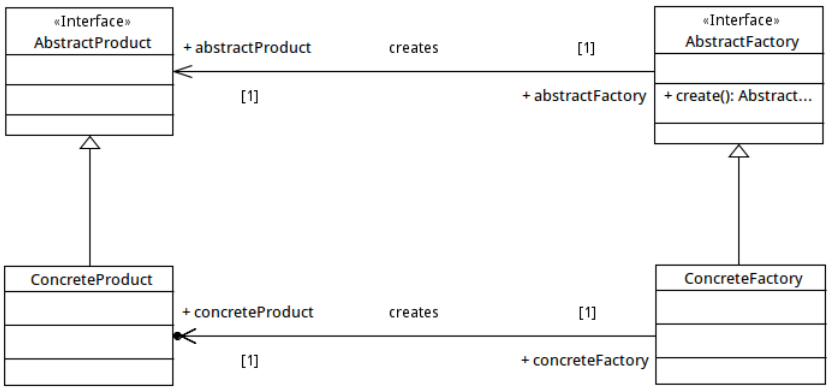

Abbildung 4.11.: Go4 Fabrik (UML)

Hierbei wird der Fabrik eine zu implementierende Schnittstelle vorgegeben, die Methoden zur Erzeugung von möglicherweise auch mehreren Produktfamilien (*AbstractProduct*) enthält. Eine konkrete Fabrik wird dann mit der Herstellung von konkreten

[11] *Information Hiding*

Produkten aus den Produktfamilien beauftragt. Klienten können nun mit den erzeugten Produkten arbeiten ohne deren Typ zu kennen, es genügt allein das abstrakte Produkt. Dies wiederum macht das ausgelagerte Erzeugungsverhalten vollkommen austauschbar, was zu einer höheren Wart- und Erweiterbarkeit (Anforderung 1) führt.

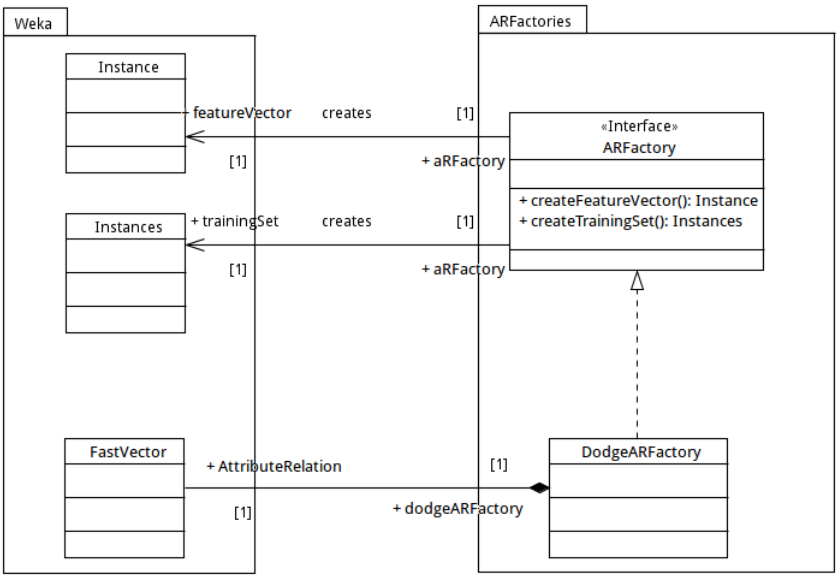

Abbildung 4.12.: Attribute-Relation-Factory (UML)

Abbildung 4.12 zeigt die abstrakte Fabrik der Klassifizierungskomponente, hier *Attribute-Relation-Factory* genannt. Der Bezug zum Muster der abstrakten Fabrik wird jedoch erst auf Objektebene klar und ist der Allgemeinheit der Weka-Datenstrukturen *Instance*, *Instances* und *FastVector* geschuldet. In Abbildung 4.10 wurde die übliche Nutzung dieser Strukturen im Zusammenhang mit machinellem Lernen vorgestellt. Zunächst wird dort das Schema eines Merkmalsvektors als Attributrelation definiert. Diese Definition könnte sich allerdings über einen längeren Produktionszeitraum, zum Beispiel aufgrund von Optimierungen, ändern. Daher möchte man diese Attributrelation möglichst zentral, hier in einer konkreten Fabrik, definieren. Ein solches Schema stellt zwar keinen Typ auf der Ebene des Java-Typsystems dar, kann aber doch als Typ, um den nur die konkrete Fabrik weiß, interpretiert werden. Die Klientseite arbeitet dabei nur mit Objekten vom Typ *Instance* für Merkmalsvektoren und *Instances* für Trainingsmengen, ohne um das konkrete Schema der Attributrelation zu wissen.

Entwurfsmuster: Strategie

Strategie ist ein Verhaltensmuster, das die Auswahl von Algorithmen (Verhalten/Strategien) zur Laufzeit ermöglicht. Abbildung 4.13 zeigt die klassische Darstellung des *Gang of Four* Strategiemusters.

Verhalten wird bei diesem Muster in Klassen gekapselt und in sogenannten Familien

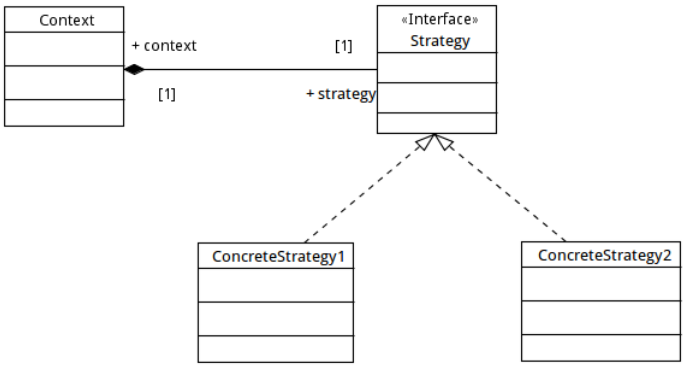

Abbildung 4.13.: Go4 Strategie (UML)

strukturiert. Diese Familien werden üblicherweise als Schnittstellen dargestellt, damit eine konkrete Kontextklasse gegen diese implementiert werden kann, um sie so mit variablen Verhalten auszustatten. Dies führt einerseits dazu, dass Wissen um konkretes Verhalten vor Kontextklassen versteckt wird, andererseits macht es sie aber auch zukünftig bezüglich ihres Verhaltens erweiterbar.

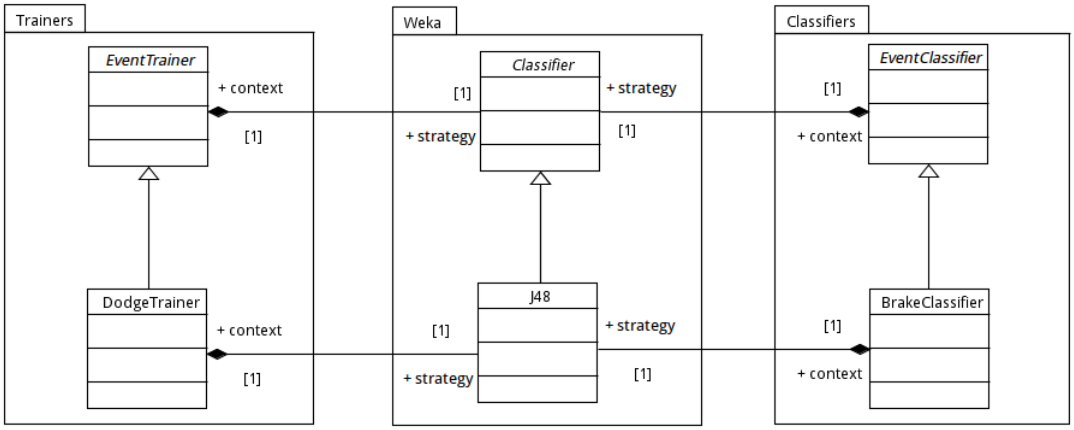

Abbildung 4.14.: MobSens Strategie (UML)

Betrachtet man nun die Weka-Algorithmen für Vorhersagemodelle, stellt man fest, dass sie eine *Classifier*-Familie darstellen. Daher liegt es nahe, für die (Kontext-)Klassen zum Trainieren und Anwenden von Klassifikatoren das Strategie-Entwurfsmuster anzuwenden (Abbildung 4.14). Dabei wird das Verhalten (Trainieren bzw. Klassifizieren) in die Strategie-Klassen ausgelagert (Anforderungen 2.3. und 2.4.). Da gelernte Klassifizierungsmodelle Teil des Zustands von *Classifier*-Erben sind, ist Serialisierungs- und Deserialiserungsverhalten schon inherent über die Java *Object*-Klasse gegeben (Anforderungen 2.5. und 2.6.).

Hier wird nur Infrastruktur zur Erweiterbarkeit bezüglich Ereignissen gelegt. Um

ein neues Ereigniss der Komponente hinzuzufügen, müssen lediglich Klassen von *EventTrainer* und *EventClassifier* abgeleitet werden (Anforderung 1). Bei den eigentlichen Machine-Learning-Algorithmen kann komplett auf die Weka-API zurückgegriffen werden.

Labels und Generics

Um maschinelles Lernen zu ermöglichen, müssen Trainingsdaten bereits klassifiziert bzw. *markiert* sein. Diese werden im Subpaket *Events.Labels* für jedes Ereignis als Enumeration als Subtypen von *EventLabel* definiert (Abbildung 4.15).

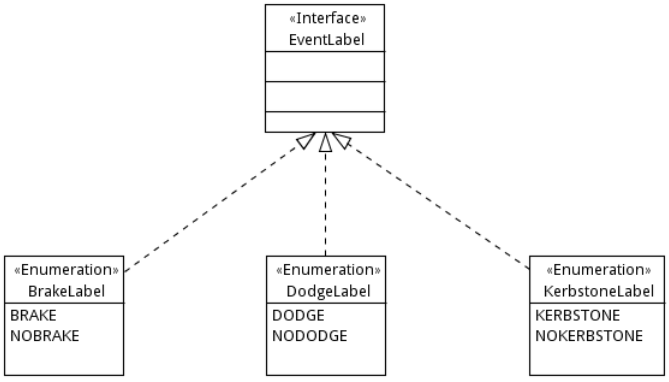

Abbildung 4.15.: Ereignis-Label (UML)

Zusätzlich sind die abstrakten Klassen bzw. Schnittstellen *ARFactory*, *EventClassifier* und *EventTrainer* mit einem generischen Typparameter versehen, der wiederrum auf *EventLabel* eingeschränkt ist, um so einen zentralen und eindeutigen Bezug zwischen den konkreten Klassen und einem Ereignis herzustellen.

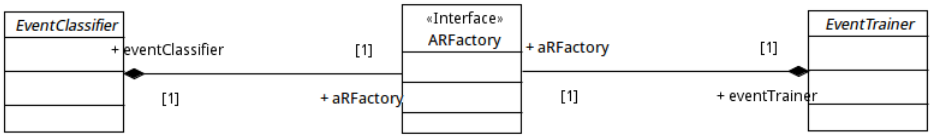

Abbildung 4.16.: MobSens Factory Strategy (UML)

Abbildung 4.16 zeigt die Beziehungen zwischen *ARFactory*, *EventClassifier* und *EventTrainer*. Hier wurde wieder das Strategiemuster angewand (vgl. dazu Abbildungen 4.14 und 4.13), um die Funktionalität der Fabrik variabel verwendbar zu halten. Zudem konnte so redundanter Quelltext in die abstrakten Klassen *EventClassifier* und *EventTrainer* hochgezogen werden.

5. Implementierung

Die Implementation befasst sich mit der Umsetzung der Architektur. Zuerst wird der mobile Part besprochen, gefolgt von der Gestaltung des Servers. Abschließend geht das Kapitel auf die Klassifizierung ein. Hierfür werden Probleme identifiziert, deren Hintergründe aufgezeigt und Lösungen besprochen. Um den Quelltext auf das Wichtigste zu reduzieren und die Programmierung effizient zu gestalten, greifen die Implementationen auf existierende Bibliotheken und Werkzeuge zurück. Deren herausstechende Anwendungsfälle erläutert dieses Kapitel auch.

5.1. Mobile

Die Entwicklung der mobilen Komponente wurde in drei Phasen unterteilt. In der Abbildung 5.1 werden die Phasen als Pakete, die Programme als Artefakte und die Beziehungen zwischen diesen als Abhängigkeiten beschrieben.

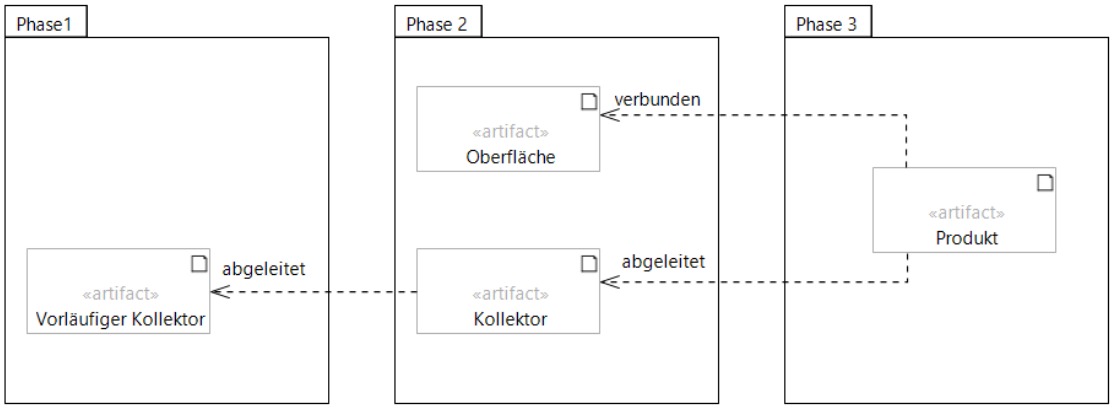

Abbildung 5.1.: Phasen der Kollektor-Entwicklung

Die erste Phase kam vor Beginn der Architektur zu Ende und mündete im Artefakt des *Vorläufigen Kollektors*, einem Programm zur Veranschaulichung des Konzeptes.

Die zweite Phase umfasste zwei parallele Projekte. Die Phase wurde aufgeteilt, um der Teamstruktur in der Untergruppe für mobile Entwicklung gerecht zu werden. Die Erfahrung aus der vorhergehenden Phase wurde hier in einer verbesserte Variante der mobilen Komponente eingesetzt. Diese wird als *Kollektor* bezeichnet. Währenddessen ist Vorwissen über die Entwicklung von Android Oberflächen, welches in 2.1 beschrieben ist,

in einem weiteren Programm umgesetzt worden. Diese wird als *Oberfläche* bezeichnet. Der Zweck dieses Artefakts war die Konzeption der Endbenutzer-Schnittstelle.

In der dritten und letzten Entwicklungsphase wurden die beiden individuellen Komponenten aus der zweiten Phase zu einem funktionalen Programm zusammengesetzt. Dazu wurden die Funktionsaufrufe, die der Anwender in der Oberfläche tätigt, mit den Implementierungen im Kollektor verbunden. Im Diagramm ist das Ergebnis der letzten Phase als *Produkt* gekennzeichnet.

5.1.1. Implementierung des Collectors

Zu Anfang wurde, wie im Architektur-Teil, eine Vorabversion geschrieben. In dieser Stufe des Programms sind bereits die Grundelemente der folgenden Implementierung vorweggenommen. Unter anderem sind die Komponenten des HTTP-Transports, der Verkapselung der Daten mit dem JSON-Format, die Realisierung von Hintergrundprozessen mithilfe von Intent-Services sowie Interprozesskommunikation durch AIDL im Vorläufer vorhanden. Auch die Aufnahme der Sensordaten, die Grundlage für das Projekt, ist hier bereits realisiert worden.

Obwohl diese nicht vollends in ihrem vorgesehenen Zweck verwendet wurden, bot das Entwickeln der Komponenten dem Entwicklerteam eine Einführung in die erforderlichen Konzepte. Außerdem wurde schnell klar, dass für die Verbindung der Komponenten ein Modell, welches auf Datenfluss ausgerichtet ist, einem Modell von Klassen und Feldern und Referenzen vorzuziehen ist. So ist das System von verschalteten Flusstransformationen und Flussfiltern entstanden.

Da die Anwendung erfordert, dass ein Programm im Hintergrund auf dem Smartphone abläuft, musste eine Instanz der Android Service-Klasse verwendet werden. Dieser ermöglicht eine kontinuierliche Ausführung, auch wenn das Gerät kein aktiviertes Display hat oder eine andere Anwendung im Vordergrund läuft.

Die Trennung von Steuerung und Programmverhalten, die auch dem Model-View-Controller Muster entspricht, machte es auch möglich, die Steuerung von einer für den Endbenutzer optimierten Oberfläche übernehmen zu lassen.

Auf diese Aspekte soll in den folgenden Seiten eingegangen werden. Der Quellcode hierzu ist im Projekt *MobileSensingCollector*.

Kommunikation mit den Services

Wie bereits erwähnt, wurde die Sensordatenaufnahme und deren Steuerung getrennt. Daraus sind die Komponenten des Collectors und des Controllers entstanden. Der Collector ist hier der Dienst, der im Hintergrund ausgeführt wird und der Controller die Benutzerschittstelle. Um zwischen den Komponenten zu kommunizieren, wird in der Implementierung der Austausch von Intent-Objekten eingesetzt. Die allgemein gefasste Bestückung der Objekte mit Schlüssel-Wert Paaren erfordert eine Eingrenzung vom Entwickler, da keine Signatur mehr vorhanden ist.

Üblicherweise werden Schnittstellen anhand von Signaturen bereitgestellt, die über die Anzahl und Form der Parameter Auskunft geben. Die Parameter tragen meistens

symbolische Namen und grenzen die Auswahl der Werte durch Parametertypen ein. In der Abbildung 5.2 ist die Signatur für eine Methode `onStartCollector` dargestellt. Sie hat den Rückgabetyp `void`, was sie als Prozedur ohne Rückgabewert kennzeichnet und einen Parameter vom Typ `String` mit dem Namen `title`. Es handelt sich hier also um eine Zeichenkette für einen Titel. Frei übersetzt bedeutet dieser *Beim Starten des Kollektors*. Es handelt sich um einen Zustands-Hörer, der über das Starten einer Aufnahme informiert wird. Auf den Zugriffsmodifikator `protected` sowie die Implementierungsvorgabe `abstract` für ableitende Klassen soll hier nicht weiter eingegangen werden.

```
protected abstract void onStartCollector(String title);
```

Abbildung 5.2.: Beispiel einer Signatur in Java

Im Gegensatz dazu wird die Signatur von Intents nur in der Dokumentation beschrieben. Der Entwickler muss sich um deren Beschaffenheit selbst sorgen. Dadurch können Probleme entstehen. Einerseits kann man solch ein Objekt abschicken und nicht alle Notwendigen Parameter belegen. Beim Entgegennehmen entstehen dann bei nicht defensiven Implementierungen Probleme. Außerdem ist es möglich, dem Parameter einen falschen Typen mitzugeben.

```
Date time;
double latitude;
double longitude;

Intent intent = new Intent("Location␣Update");

intent.putExtra("Time", time);
intent.putExtra("Latitude", latitude);
intent.putExtra("Longitude", longitude);
```

Abbildung 5.3.: Beispiel eines signaturlosen Intents[1]

Dieser Umstand wird dadurch unterstützt, dass sich die Signaturen der Intent-Beladung für verschiedene Typen den Namen teilen - man spricht von einer Überladung. In der Abbildung 5.3 sieht man, dass `time` einen anderen Typ hat als `latitude` und `longitude`, jedoch beide dem selben Methodenaufruf, namentlich `putExtra`, übergeben werden. Ändert man den Typen einer Variable im Programm, so wählt der Compiler automatisch einen anderen Typen für die Belegung im Intent.

Die Lesemethoden für diese sind jedoch mit einem festen Typen annotiert, somit folgt ein Rücksprung auf den Standardwert, da im Aufruf nicht mehr der erforderte

[1]Die Variablen für Zeit und Ort sind hier nicht belegt, was unter normalen Umständen kein gültiger Java-Code wäre. Im echten Quelltext sind den Variablen Werte zugewiesen. Desweiteren sind die echten Namen in den Zeichenketten durch Symbolische Namen ersetzt worden.

Typ verwendet wird. Das Programm kann den Aufruf abarbeiten, jedoch nicht in dem beabsichtigten Ablauf.

Um dieser unübersichtlichen Fehlerquelle entgegenzuwirken wurde sowohl das Absenden als auch das Empfangen von Intents strukturiert. Dazu wurde das Versenden in einem Methodenaufruf zusammengefasst, womit die fehlende Signatur wieder hergestellt ist. Im Empfang stellen Zusicherungen die Typensicherheit und Parameterpräsenz fest, danach wird eine zum Absender passende Methode aufgerufen.

Alternativ zu den Intents gibt es die Möglichkeit zwischen zwei Prozessen eine Schnittstelle auf zumachen. Im Vorläuferprogramm wurde IPC - die Interprozesskommunikation - im Vergleich zum Endprodukt ausgiebiger benutzt. Wo sich jetzt nur eine Zustandsabfrage befindet, hat sich vorher die gesamte Steuerung abgespielt. Aufnahmekontrolle so wie das Herunterfahren des Dienstes waren als Methoden im AIDL-Interface enthalten. Der Dienst stellte diese Bereit und verarbeitete die Anfragen.

Um die Kommunikation mit IPC zu vereinfachen, wurde der sehr technische Vorgang des Verbindens zweier Prozesse, der über Klassennamen, Binder, Aufrufe sowie benachrichtigende Verbindungspunkte stattfindet, zusammengefasst. Mit Bindern sind die Implementierungen der Schnittstelle beschrieben. Die Verbindungspunkte beschreiben den Verbindungsstatus der Prozesse und können Änderungen derer mitteilen.

Als Muster der Vereinfachung nahmen wir an, dass ein Dienst von vielen Benutzerschnittstellen verwendet wird und diese beliefert. Die Benutzeroberflächen hingegen kennen genau ihren Bedarf an Funktionalität und können konkrete Dienste benennen, die jene Bereitstellen; die Verbindung von mehr als einem Dienst wurde vernachlässigt, da in der vorliegenden Applikation nur ein Dienst benötigt wird. Daraus leiteten sich dann die Klassen der *Connecting Activity* und des *Connected Service* ab. Beide können über den Verbindungsstatus berichten. Die Aktivität greift auf existierende Dienste zurück und startet sie, wenn noch nicht vorhanden. Für den Anwender der Klassen muss nur der Name der Klasse und der Binder bekannt sein. Aufrufe werden im Lebenszyklus der Android-Konstrukte vollzogen und so verborgen, was den Code strukturiert.

Allerdings sind Interprozessschnittstellen nicht für die Steuerung eines autonom arbeitenden Dienstes erforderlich. Sie bieten die Möglichkeit, dem Aufrufer einen Rückgabewert zu liefern. Außerdem pausiert das Programm des Aufrufenden solange, bis der Verarbeitende die Anfrage abgearbeitet hat. Dieses System, auch wenn es für verschiedene Anwendungsbereiche sehr sinnvoll ist, schränkt die Performanz unnötig ein. Da die Steuerungsnachrichten allesamt asynchron abgearbeitet werden können, muss der Absender nicht auf dessen Ankunft und Behandlung warten. Letztendlich waren daher Intent-Objekte die bessere Alternative, da sie dem Programm erlauben weiterzuarbeiten, nachdem der Befehl verschickt wurde. Die Bindung der Aktivitäten und Dienste sind, obwohl die IPC-Schnittstelle nur noch eine Zustandsinformation enthält, weiterhin vorhanden. Durch diese konnte die Existenz des aufnehmenden Dienstes sichergestellt werden.

Der relevante Code für die Kommunikation ist in den Packages
`mobsens.collector.communications`,
`mobsens.collector.intents` und
`mobsens.collector.drivers.messaging` verortet.

Diese Konzepte könnte man auch durch Intents realisieren, sie bieten auch die Möglichkeit den behandelnden Dienst zu starten, falls er nicht existiert. Zum Zeitpunkt der Implementierung waren die Mechanismen dem Entwicklerteam nicht bekannt.

Sensordatenaufnahme

Die Voraussetzung für das Erkennen der Ereignisse ist die Aufnahme von Sensordaten. Die Schnittstelle des Android Betriebssystems für Sensoren ist die Android Hardware Sensors API, die über ein verwaltendes Objekt geregelt wird. Der sogenannte Sensor Manager ist sowohl in den Diensten als auch in den Aktivitäten erreichbar. Er lässt sich dort als System-Dienst abfragen und bietet unterschiedliche Methoden, mithilfe derer man auf Änderung der Sensoren eingehen kann.

Die Daten werden im Sinne des Listener-Patterns an angeschlossene, vom Anwendungsentwickler bereit gestellte Anlaufstellen - den Zuhörern - weitergegeben. Die Quelle der Sensordaten findet sich im Betriebssystem Linux. Dort werden die Sensoren eines Smartphones nach der Hardwareabstraktion als Devices aufgelistet. Die Android API bedient sich dieser und bildet aus den Rohdaten dann Objekte, die an die Anlaufstellen gegeben werden.

Diese Objekte sind üblicherweise Speziell an die Sensorausgabe angepasst und haben bis auf *Object* keine gemeinsame Superklasse. Dabei lässt sich für alle aus der natürlichen Umgebung abgeleiteten Ereignisse mindestens deren Zeitpunkt als Gemeinsamkeit anführen. Um diesen Umstand zu korrigieren wurden die Daten neu verkapselt, jedem verarbeiteten Sensor ist folglich jeweils ein Treiber und ein Objekt zugeordnet worden. Wo in frühen Versionen die gemeinsamen Eigenschaften im Vordergrund standen, sind in der abschließenden Applikation Methoden für Serialisierung entstanden. Auf diese wird im Abschnitt über das Format und den Upload an den Server eingegangen.

Die vorhin erwähnten Sensoren-Treiber vereinfachen den Zugriff auf die API des Systems, indem sie die hoch parametrisierte Registrierung für Zuhörer und deren Trennung reduziert. Durch Aufruf von *Start* und *Stopp* werden alle notwendigen Maßnahmen ergriffen. Außerdem gliedern sich die Treiber in die Pipeline ein, die in der folgenden Sektion beschrieben steht.

Dem abschließenden Programm sind die folgenden Sensoren bekannt:

- Netzwerkstatus des Zellulären Telefonnetz
- Ortung
- Umgebungssensoren
  - Beschleunigung
  - Axiale Beschleunigung
  - Magnetfeld

Aus dem Umgebungssensor für Beschleunigung leitet die Android API Sensoren für die Gravitation sowie die gravitationsbereinigte Beschleunigung ab. Dazu wird ein Tiefpass-

Filter auf die Beschleunigung angewendet und die Richtungsänderung durch das Gyroskop, dem Sensor für die Rotationsbeschleunigung, vorausgesehen [ABH12]. Die gravitationsbereinigte Beschleunigung, die in der API *Lineare Beschleunigung* genannt wird, errechnet sich daraus folgend als Abzug der Gravitation von der real gemessenen Beschleunigung.

Die Aufnahme der Sensoren ist in den Packages `mobsens.collector.drivers .sensors` geregelt, die GPS-Aufnahme wird in `mobsens.collector.drivers.locations` gesteuert.

Pipeline

Der Entschluss, die Daten in einer Pipeline zu verwalten ergründet sich unter anderem in der Tatsache, dass in der Applikation Daten von den Sensoren durch die Verkapselung in Dateien fließen.

In diesem Paradigma lassen sich aber auch die Probleme umgehen, die durch Multithreading entstehen. Der Begriff bezeichnet Aufgaben, die in sogenannten Threads gleichzeitig abgearbeitet werden. Diese teilen sich, im Gegensatz zu von einander getrennten Prozessen, ihre Variablen und Daten.

Multithreading wird in der mobilen Applikation benötigt, da eine Diskrepanz zwischen den entstehenden Sensordaten und deren Serialisierung und Sicherung besteht. Sowohl das Abhören der Sensordaten als auch das Schreiben der Sicherungsdatei wird vom Betriebssystem übernommen. Wenn man die Sensordaten direkt nach ihrer Entgegennahme speichern würde, wären deren Empfangs-rate und die Schreib-rate des Dateisystems aneinander gebunden. Das jeweils langsamste Glied würde den Durchsatz limitieren. Um dem entgegenzuwirken wird ein geteilter Speicher für die Daten eingerichtet. Dieser wird vom ersten Thread mit eingehenden Sensordaten gefüllt. Ein zweiter liest den Speicher aus und schreibt den Inhalt in die Sicherungsdatei. Unterschiedliche Geschwindigkeiten wirken sich in einem *voller werdenden* Speicher aus, wenn das abspeichern länger braucht, und in einem *leer laufenden* Speicher aus, wenn die Sensoren nicht so viele Datensätze erzeugen.

Jedoch kann man Threads nicht ohne Risiken einsetzen. Da man üblicherweise nicht sagen kann, welcher der Threads zuerst an einem bestimmten Punkt angelangt ist, können Probleme mit dem Lesen und Schreiben von Variablen auftreten. Je nachdem welcher Thread schneller ist, können verschiedene Zustände entstehen, man spricht von *Race-Conditions*. Um dem Problem entgegenzuwirken, sind in der Informatik verschiedene Konzepte zur Synchronisierung bekannt. Dazu zählen Semaphoren, atomare Operationen und Resourcenbelegung. Diese gehen sehr stark auf die technischen Hintergründe von Programmabläufen ein.

Um in der Applikation die Semantik in den Vordergrund zu stellen, wurde der erwähnte geteilte Speicher als ein Element der Pipeline abstrahiert. Der *WorkerCache* reiht eingehende Daten in eine threadsichere Schlange ein. Das in der Android API enthaltene Konzept des *Worker* kümmert sich dann um die Weiterleitung des Inhalts der Schlange.

Nachdem die Objekte aus der Schlange entnommen wurden, fließen sie durch wei-

tere Knoten. In Transformationen wird auf die eingehenden Elemente eine Funktion angewandt und deren Rückgabewert als ausgehendes Datum verwendet. Filter hingegen sehen die ankommenden Daten und entscheiden anhand eines Kriteriums, ob sie diese durchlassen oder zurückhalten. Bis auf Spezialfälle wie den *WorkerCache* lassen sich mit diesen Grundkonzepten ausdrucksstarke Flussnetze bilden.

Die Grundstruktur wird in `mobsens.collector.pipeline` vorgegeben,

`mobsens.collector.pipeline.basics` enthält erweiterte Implementierungen. Das Paket `mobsens.collector.consumers` enthält die Ziele der Pipeline.

Binäre Serialisierung

Um die von den Sensoren erfassten Daten auf dem Gerät vorzuhalten müssen sie in einem serialisierbaren Datenformat vorliegen. Die direkte Herangehensweise besteht darin, die Zahlenwerte und Texte im selben Format zu speichern, wie sie im Prozessor und Datenspeicher gehalten werden: als Bytes [Che14].

Bei den zu serialisierenden Daten handelt es sich um komponierende Strukturen. Diese unterscheiden sich in ihrem Typ und den komponierten Elementen. Das Datum des Accelerometer ist beschrieben durch die Komponenten der X-, Y- und Z-Achse: die Bewegungsrichtungen parallel zum Display sowie die lotrechte Bewegung. Außerdem ist die Zeit enthalten. Das Datenstruktur für Anmerkungen enthält die Anmerkungszeit und den angemerkten Text.

Um diese Daten jetzt im binären Format zu kodieren müssen den Komponenten jeweils Länge und Position zugeordnet werden. Die Länge spiegelt die Größe eines Wertes im Prozessor wieder. Die Position lässt sich aus der Addition der vorhergehenden Längen errechnen. Mithilfe der Abfolge kann eine Struktur genau so hergestellt werden, wie sie gespeichert wurde. Dem vorher erwähnten Accelerometer würde der Aufteilung in Tabelle 5.4 entsprechen:

Komponente	Position	Länge
Zeit	0	8
X-Achse	8	8
Y-Achse	16	8
Z-Achse	24	8

Abbildung 5.4.: Komponenten der Accelerometer-Daten

Für Daten fester Größe ist dies kein Problem. Wenn man jedoch Daten mit variabler Größe speichert, wie es zum Beispiel Texte sind, hilft die Spezifikation der Länge nicht. Um nicht festgesetzte Größen zu erlauben, setzt man ein der Komponente vorgestelltes Feld ein. Dieses gibt die Länge des Nachfolgers an. Die Anmerkung bietet dafür das Beispiel, wie in der Abbildung 5.5 zu erkennen ist.

Eine andere Möglichkeit dafür ist der Einsatz eines Terminators. Damit wird ein Symbol beschrieben, das nicht in den Daten der variablen Länge vorkommt. Der Dekodierende liest so lange, bis er dieses Zeichen sieht. Im Falle von Text wird oft das

Komponente	Position	Länge
Zeit	0	8
Länge des folgenden Textes	8	4
Text	12	*Wie zuvor angegeben*

Abbildung 5.5.: Komponenten der Anmerkungs-Daten

`NUL`-Zeichen aus dem ASCII-Alphabet verwendet. Dieses ist ein Unsichtbares Zeichen, was dem Abschluss einer Zeichenkette dient. Für andere Datentypen lässt sich so ein Terminator ebenso einführen.

Da man nicht nur eine Art von Daten speichern will, sondern mehrere verschiedene - wie anhand von Accelerometer und Anmerkung zu erkennen - muss ein Verfahren angewendet werden, mit dem verschiedenartige Daten aneinandergereiht werden können. Für die verschiedenen Datenklassen wird jeweils ein Präfix gewählt. Dieser wird dem Datum vorangestellt. Liest man jetzt im Datenstrom, so beginnt man zuerst beim Präfix. Anhand des Präfixes kann dann festgestellt werden, wie weiter vorgegangen werden muss. Ist ein Datum abgeschlossen, so befindet man sich vor dem nächsten Präfix. In Abbildung 5.6 ist dieses Verfahren angewendet worden. Daten des Typs *Anmerkung* wird eine 0 vorangestellt, *Accelerometer* ist durch den Präfix 1 beschrieben.

0	*Anmerkung*	1	*Accelerometer*	1	*Accelerometer*	0	*Anmerkung*	...

Abbildung 5.6.: Präfixkodierung

Aus dem Zusammenspiel der genannten Methoden entsteht ein Sehr kompaktes Datenformat, das keine redundanten Informationen enthält. Jedoch sind Dateien in der binären Kodierung nicht vom Menschen ohne Hilfsprogramme lesbar. Daraus können Probleme bei der Fehlerbehebung sowie beim *Reverse Engineering* entstehen. Ersteres meint, dass man bei Problemen auch berücksichtigen muss, dass der Fehler im Kompakten, dafür aber instabilen Format liegt. Das Zweite beschreibt die Wiederherstellung von strukturellen Informationen ohne explizite Dokumentation. Aus binären Informationen sind die Code-Strukturen nicht so einfach ersichtlich wie aus textuellen. Außerdem ist die Kodierung sehr rigide, man kann also schwer Änderungen durchführen, die auch rückwärtskompatibel sind.

Nach gründlicher Abwägung wurde als Datenformat daher eine andere Repräsentation gewählt. Diese soll im folgenden Abschnitt beschrieben werden.

Serialisierung als Text

Um den im vorhergehenden Abschnitt angesprochenen Problemen aus dem Weg zu gehen fiel die Entscheidung, auf ein bereits existierendes Datenformat zu setzen. Angesprochen wurden CSV und JSON. In CSV wird eine feste Datenstruktur als Tabelle abgespeichert

[Sha05]. Dieses Format ist für ein Text-basiertes sehr Kompakt. Die Informationen über die Zuordnung von Feldern zu Werten wird nur einmal im Tabellenkopf gespeichert. Jedoch müssen für unterschiedliche Datentypen jeweils eigene Tabellen angelegt werden. Man kommt also in die Verlegenheit, verschiedene Dateien anzulegen, was mit einem Mehraufwand in der Verwaltung einhergeht. In den Abbildungen 5.7 und 5.8 ist ein Beispieldatensatz angeführt.

Zeit	X-Achse	Y-Achse	Z-Achse
12:08:01.001	6.2	0.1	-1.5
12:08:01.107	1.2	2.7	4.1
12:08:01.149	3.1	3.2	-2.1
⋮	⋮	⋮	⋮

Abbildung 5.7.: Accelerometer im Format CSV

Zeit	Anmerkung
12:08:01:000	Ein Text
⋮	⋮

Abbildung 5.8.: Anmerkungsdaten im Format CSV

Die Alternative zu CSV, das JSON-Format, bildet die Werte als Zuordnung von Schlüsseln zu Werten oder als Listen von Werten ab [Bra14]. Damit sind genestete Strukturen mit Daten variabler Länge sehr einfach zu realisieren, die Darstellung ist zudem durch Verschachtelung mit Klammern äußerst verständlich. Für das Format sind Implementierungen in den gängigen Programmiersprachen vorhanden. Daher konnten sowohl die mobile Anwendung als auch der Server auf getestete Bibliotheken zurückgreifen. Die Daten aus den Abbildungen 5.7 und 5.8 sind in 5.9 im JSON-Format aufgeschrieben. `ACC` soll ab hier für Accelerometerdaten und `TAG` für Anmerkungen stehen.

Wie man sehen kann, dürfen auch unterschiedliche Daten in einer Struktur enthalten sein. Die Anmerkung, die in CSV in einer separaten Tabelle gespeichert werden muss, ist in der Alternative zwischen den Accelerometer-Daten eingeflochten. Der zusätzliche Aufwand für das Zusammenfassen mehrerer Dateien und deren Versand an den Server fällt damit weg.

In der ersten Entwicklungsphase wurde die Javascript Object Notation jedoch anders als in der vorhergehenden Abbildung verwendet. Die verflochtenen Datenstrukturen wurden zugunsten einer direkten Zuordnung zu Objekten und Relationen vernachlässigt. Stattdessen sind die unterschiedlichen Datenwerte über einen Schlüssel innerhalb einer weiteren Schale erreichbar.

Obwohl dieses Format sehr komfortabel zu erstellen und auszulesen ist, wurde es in der endgültigen Fassung noch abgeändert. Das Problem bei der Komposition von JSON-

```
[
    {
        "Typ":  "ACC",
        "Zeit": "12:08:01.001",
        "Wert": [6.2, 0.1, -1.5]
    },
    {
        "Typ":  "ACC",
        "Zeit": "12:08:01.107",
        "Wert": [1.2, 2.7, 4.1]
    },
    {
        "Typ":  "TAG",
        "Zeit": "12:08:01:000",
        "Wert": "Ein Text"
    },
    {
        "Typ":  "ACC",
        "Zeit": "12:08:01.149",
        "Wert": [3.1, 3.2, -2.1]
    }
]
```

Abbildung 5.9.: Beispieldaten im Format JSON

Dateien ist, dass die alle Daten bekannt sein müssen, bevor sie Serialisiert werden können. Das führt zu einem Datenstau und einer sehr hohen Speicherauslastung. Wenn die Datei geschrieben wird, ist das System sehr lange mit dem Umwandeln in das serialisierte Format und das Schreiben beschäftigt. Um dieser Art von Problem vorzubeugen, bieten die Java-Bibliotheken einen direkten Dateizugriff an. Man kann Dateien inkrementell - also nacheinander in kleineren Einheiten - schreiben, Anstatt den gesamten Inhalt vorher zu kennen und auf einen Schlag zu speichern.

Da bei JSON eine Struktur vollständig erstellt sein muss, um gültig zu sein, wurde darauf verzichtet, die zu sichernden Daten in einer Liste oder in einem Objekt zu speichern. Stattdessen wurden die Daten direkt geschrieben. Da die Zeichenkette für einen Zeilenumbruch in JSON keine Bedeutung hat, kann diese als Terminator verwendet werden. Somit ist je einem Objekt eine Zeile zugeordnet. Mit dieser Methode kann man das inkrementelle Erstellen von Dateien benutzen und umgeht den Kollaps durch große Datenmengen. Außerdem bleiben die Dateien für Menschen verständlich, da man anhand der Zeilennummer jedem Datum auch seine Position im Datenstrom zuordnen kann und die einzelnen Zeilen wiederum gültige Objekte sind. In der Abbildung 5.10 sind die vorhergehenden Beispieldaten im zeilenweisen JSON angegeben. Die Formatierung muss zu großen Teilen aufgehoben werden, da hierfür der Zeilenumbruch verwendet wurde.

```
{ "Typ": "ACC", "Zeit": "12:08:01.001", "Wert": [6.2, 0.1, -1.5] }
{ "Typ": "ACC", "Zeit": "12:08:01.107", "Wert": [1.2, 2.7, 4.1] }
{ "Typ": "TAG", "Zeit": "12:08:01:000", "Wert": "Ein Text" }
{ "Typ": "ACC", "Zeit": "12:08:01.149", "Wert": [3.1, 3.2, -2.1] }
```

Abbildung 5.10.: Beispieldaten im Format des zeilenweisen JSON

Um die Daten auszulesen, muss der Server auch nicht die gesamte Datei in den Arbeitsspeicher laden, sondern kann so lange lesen, bis er einen Zeilenumbruch findet, und den bis dahin gelesenen Text in ein JSON-Objekt umformen. Dieses kann weiterverarbeitet werden während man den neuen Zeilenumbruch erwartet. Lesen und verarbeiten lässt sich somit einfach trennen. Wie die Dateien an den Server geschickt werden, wird im folgenden Abschnitt besprochen.

Datentransfer vom Gerät zum Server

Um die gesammelten Messwerte des Smartphone-Programms dem Klassifizierer und der Web-Oberfläche bereitzustellen, müssen diese in ihrer serialisierten Form dem Server übergeben werden. Es gibt viele alternative Protokolle, die auf die Transportschichten IP sowie TCP und UDP aufsetzen. Da es sich bei den Daten aber um Dateien im Klartext handelt, bietet sich das Hypertext Transfere Protokoll - kurz HTTP - an. In der gängigen Version bietet dieses den Versand von Dateien, da es auch auf Webseiten in Hochlade-Formularen verwendet wird [Fie+99]. Außerdem sind alle modernen Server-Plattformen in der Lage, HTTP zu verstehen und anzunehmen. Dadurch wurden Fehler in Transportprotokollen vermieden.

Des weiteren ist im HTTP Protokoll die Möglichkeit gegeben, eine Verpackung vorzunehmen, das Zipping. Mit diesem Verfahren können Dateien, in denen oft gleiche Zeichenfolgen vorkommen, effizient verkleinert werden. Dies bietet sich an, da durch die reguläre Datenstruktur mit den wiederkehrenden Schlüsseln oft redundante Texte entstehen. Im Kapitel 6, das sich mit der Energieeffizienz-Analyse befasst, wird auf die Verpackung genauer eingegangen.

Die Schnittstelle für die Serialisierung ist in `mobsens.collector.wfj` enthalten. Die Datenobjekte der Sensoren implementieren diese.

5.1.2. Implementierung des UI

Insgesamt 5 Activities wurden benötigt, die alle via XML-Layout verwirklicht wurden. Es wurde sich hierbei für ein relatives Layout entschieden, da dies gut dafür genutzt werden kann, um voneinander unabhängige Elemente dennoch übersichtlich anzuordnen und sie zu einer Relation zueinander zu stellen, ohne dabei darauf achten zu müssen, wie groß der Bildschirm des Nutzers letztendlich ist.

Die 5 Aktivitäten sind wie folgt benannt. `activity_main.xml` fungiert als Startbildschirm und Intentionswahl[2]. Die Applikation startet in dieser Ansicht falls keine Aufnahme stattfindet. `activity_login.xml` enthält die Anmeldung und Defaultintentionswahl. `activity_map.xml` zeigt während der Fahrt die OpenStreetMap mit Overlays an, und zeigt die momentane Geschwindigkeit und zurückgelegte Strecke. `activity_rating.xml` stellt nach der Fahrt Fahrtstatistiken zur Verfügung. `activity_fasttag.xml` gibt Entwicklern die Chance, manuell Ereignisse zu taggen, also zu markieren. Dies ist nützlich um den Klassifizierer zu trainieren.

[2]Ermöglicht die Spezifizierung der Art der Fahrt. Die Unterteilung erfolgt grob in Sport, Alltag und Freizeit.

Die Implementierung der Aktivitäten ist im Projekt im Paket `mobsens.collector.activities` enthalten.

Um die Steuerung des Hintergrund-Dienstes zu realisieren, sind die Programmaufrufe der Knöpfe mit den Steuerungsaufrufen aus 5.1.1 verbunden. Das Prozessdiagramm 5.11 zeigt den Aktivitätsverlauf der mobilen Benutzeroberfläche. In dieser Grafik sind die Aktivitäten 'Start', 'Einstellung', 'Karte' und 'Zusammenfassung' äquivalent mit den implementierten activities. 'Laufender Datenversand' ist kein einzelner Bildschirm sondern ein Hintergrunddienst der den Upload der Daten steuert.

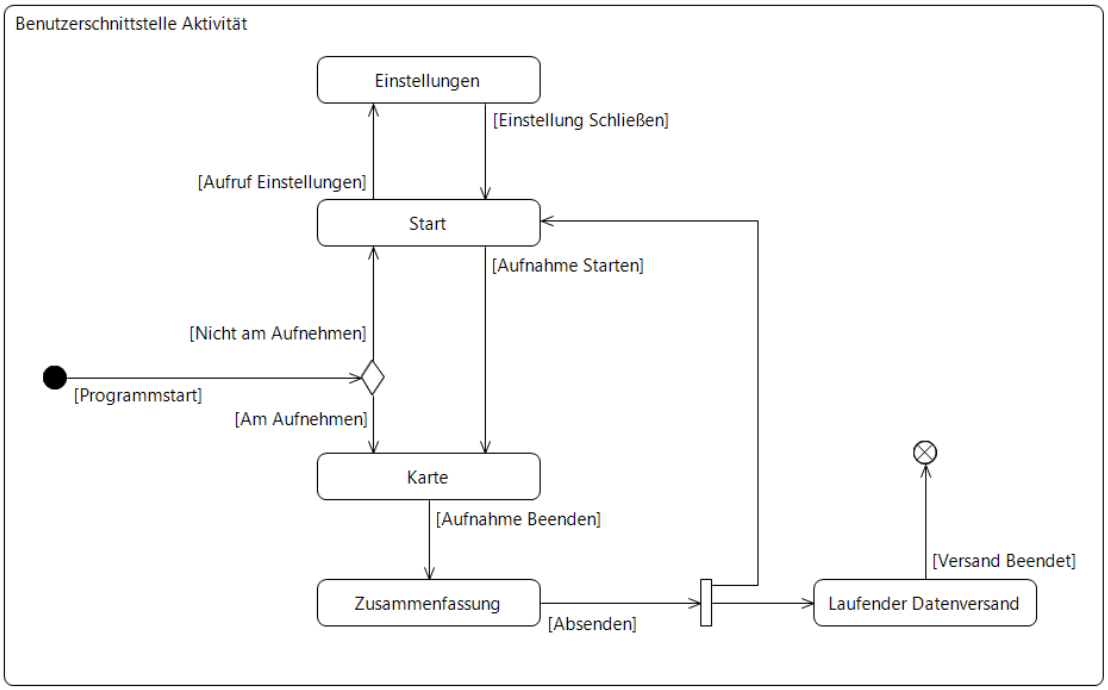

Abbildung 5.11.: Aktivitätsverlauf der Oberfläche

OSM

Die Darstellung der Karte ist in der Applikation mit der Oberflächenkomponente *osmdroid*-Bibliothek realisiert, welche sich auf das *OpenStreetMap*[3]-Projekt stützt, das im Abschnitt 2.2 benannt wurde. Die Markierung der aktuellen Position und des zurückgelegten Weges kann, mithilfe der von der Bibliothek bereit gestellten API, leicht umgesetzt werden. Diese unterstützt verschiedene Möglichkeiten Markierungen, Bewegungsdaten und Grafiken auf der Karte in sogenannten Overlays darzustellen. Das Overlay erhält die Positionsdaten und trägt sie entsprechend auf der Karte ein.

[3] `http://www.openstreetmap.org`

Die Klasse *ResourceOverlay* leitet sich von *Overlay* ab und wird benutzt, um einzelne Symbole oder Grafiken auf dem Kartenbildschirm darzustellen. In der Applikation wird so während der Sensordatenaufnahme eine blaue Raute an der momentanen Position des Gerätes dargestellt, welche fortlaufend durch den GPS-Sender ermittelt wird. Dieses Overlay wird jedes mal aktualisiert, sobald sich die momentane Position des Gerätes ändert.

Die bereits in *osmdroid* enthaltene Klasse *PathOverlay* erhält eine Liste von Koordinatenpunkten, die dann sequentiell mit Geraden verbunden werden und so die Möglichkeit bieten, Pfade auf der Karte einzuzeichnen. In diesem Projekt wird ein Path Overlay genutzt, um die bisher gefahrene Strecke zu markieren, indem regelmäßig die momentane Position des Geräts an die Liste der Wegpunkte angehängt wird. Dieses Overlay aktualisiert sich nur nach festgelegten Zeitabständen, um zu verhindern dass der Pfad zu schnell mit zu ähnlichen, unnötigen Koordinaten gefüllt wird. Das Wegpunktelimit im Path Overlay liegt bei circa 10.000 Punkten, was zwar die Länge der gezeichneten Strecke begrenzt, mit der zusätzlichen Wegpunktefrequenzbegrenzung für dieses Projekt allerdings ausreicht.

5.1.3. Design des UI

Ziel des Applikations-Designs war, dass der Nutzer die Applikation problemlos auf dem Fahrrad steuern kann. Wichtig ist deswegen vor allem ein schlankes Design, ohne viele Bedienelemente und die Aufnahme, mit so wenig Schritten wie möglich, starten zu können. Durch die Option eine Standartintention wählen zu können ist es gelungen, die Applikation mit nur 2 Taps komplett zu starten: Das Öffnen der Applikation und das Starten der Aufnahme.

Die Applikation ist in 4 Screens unterteilt, die im folgenden Abschnitt hinsichtlich der Designentscheidung näher erläutert werden.

Startbildschirm

Der Startbildschirm der Applikation ist für die Intentionswahl gedacht. Auch wenn diese Funktion aktuell noch keine direkte Berücksichtigung in der Datenverwertung findet, zeigt sich hier noch Entwicklungspotential.

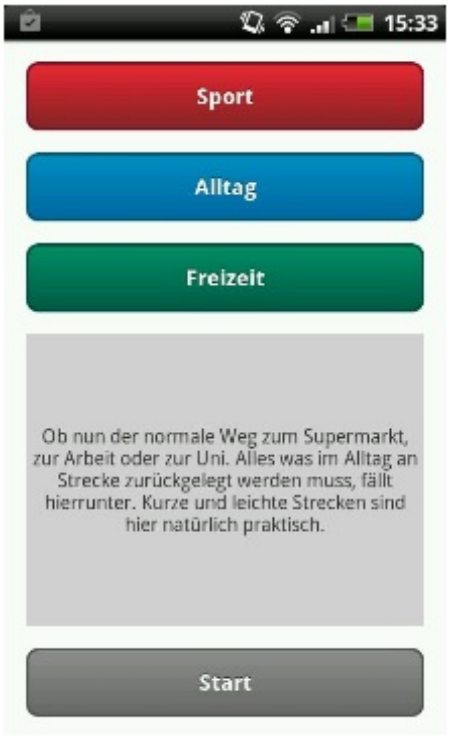

Abbildung 5.12.: Startbildschirm

Die Buttons sind groß und füllen einen Großteil der Bildfläche aus. So ist das Starten der Applikation auch vom Fahrrad aus ohne Probleme möglich (Abbildung 5.12). Dadurch ist alles übersichtlich und leicht zu erfassen, was unter anderem auch an den Farben der Intentionsbuttons liegt. Die Einfärbung bewirkt, dass auf einen Blick erkannt werden kann, um welche Intention es sich handelt. Dabei wurden die Farben an die Begriffe angepasst.

Grün z.B. steht für Natur und lässt sich dadurch gut mit dem Thema Freizeit verbinden. Um das Auge nicht zu überlasten sind sie entsättigt worden. Das große Textfeld erleichtert nicht nur das Lesen erheblich, sondern auch das Bedienen. Die Genauigkeit eines Taps kann so auch auf Kopfsteinpflaster gewährleistet werden. Dank einer Einstellungsmöglichkeit in den Optionen, kann eine Intention angegeben werden, die Standartmäßig beim Öffnen der Applikation ausgewählt ist So muss nur noch der Start-Button betätigt werden.

Optionen und Einloggen

Um die Bedienung in den Optionen zu vereinfachen, wird der Bildschirm in zwei Teile unterteilt: Das Einloggen und die Intentionsvorauswahl (Abbildung 5.13). Mehr wird für die Applikation nicht benötigt.

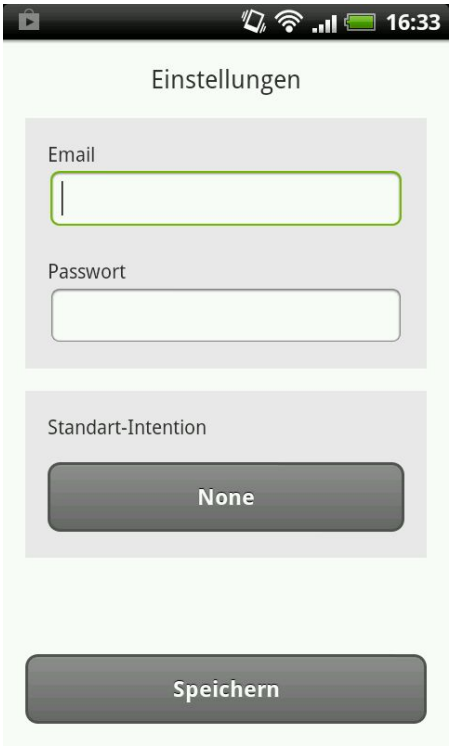

Abbildung 5.13.: Optionen und Einloggen

Die graue Hinterlegung dient dabei zur übersichtlichen Abgrenzung , ohne dabei aufdringlich zu sein. Die meistgenutzte Funktion, das Einloggen, befindet sich in der oberen Hälfte des Screens und zieht sich bis leicht über die Mitte. Durch die graue Markierung, die die beiden Eingabefelder zu einer Einheit verbindet, steht dieser Bereich direkt im Fokus des Users.

Um die Auswahl der Standardintention zu vereinfachen, sind die Buttons in der jeweiligen Farbe der Intention eingefärbt.

Kartenbildschirm

Die Karte ist möglichst groß und ohne störende Zusatzinformationen. Nur eine kleine Geschwindigkeitsanzeige und die zurückgelegte Strecke erscheinen am unteren Rand (Abbildung 5.14). So soll gewährleistet sein, dass der Fahrende von der Applikation nicht abgelenkt wird und nur das zu sehen bekommt, was ihn auch tatsächlich in diesem Moment interessiert.

Abbildung 5.14.: Kartenbildschirm

Der große Stopp-Button ist ebenfalls leicht zu erreichen und zu bedienen, selbst wenn noch gefahren wird. Um Verwirrungen zu vermeiden, dreht die Karte sich automatisch in Fahrtrichtung mit. So ist der Nutzer in der Lage, sich schnell zu orientieren, was besonders wichtig ist, da er während der Fahrt sonst den Blick zu lange von der Fahrbahn abwenden müsste.

Endbildschirm

Ist die Tour beendet, werden noch einmal einige Daten angezeigt (Abbildung 5.15). Hierbei ist alles in einfacher Schrift gehalten. So wird das Lesen erleichtert und die Schrift wirkt deutlicher und sachlicher. Der Absenden-Button ist auch hier groß gehalten und nachdem er betätigt wurde, gelangt man auf den Startbildschirm zurück.

Abbildung 5.15.: Endbildschirm

5.2. Server

5.2.1. Installation des Servers

Grundsätzlich ist die Server-Software so geschrieben, dass sie auf allen gängigen Plattformen läuft, welche einen Ruby-Interpreter besitzen. Bei dem Projektpraktikum kam Ubuntu Linux zum Einsatz. Die Softwarekonfiguration des Servers sieht wie folgt aus:

- Ubuntu 13.10 GNU/Linux 3.11.0-12-generic
- Ruby 2.0.0.299-2
- Postgresql 9.1
- Apache 2.4.6
- ruby-passenger 3.0.13debian-1.2

5.2.2. Datenbank-Schema

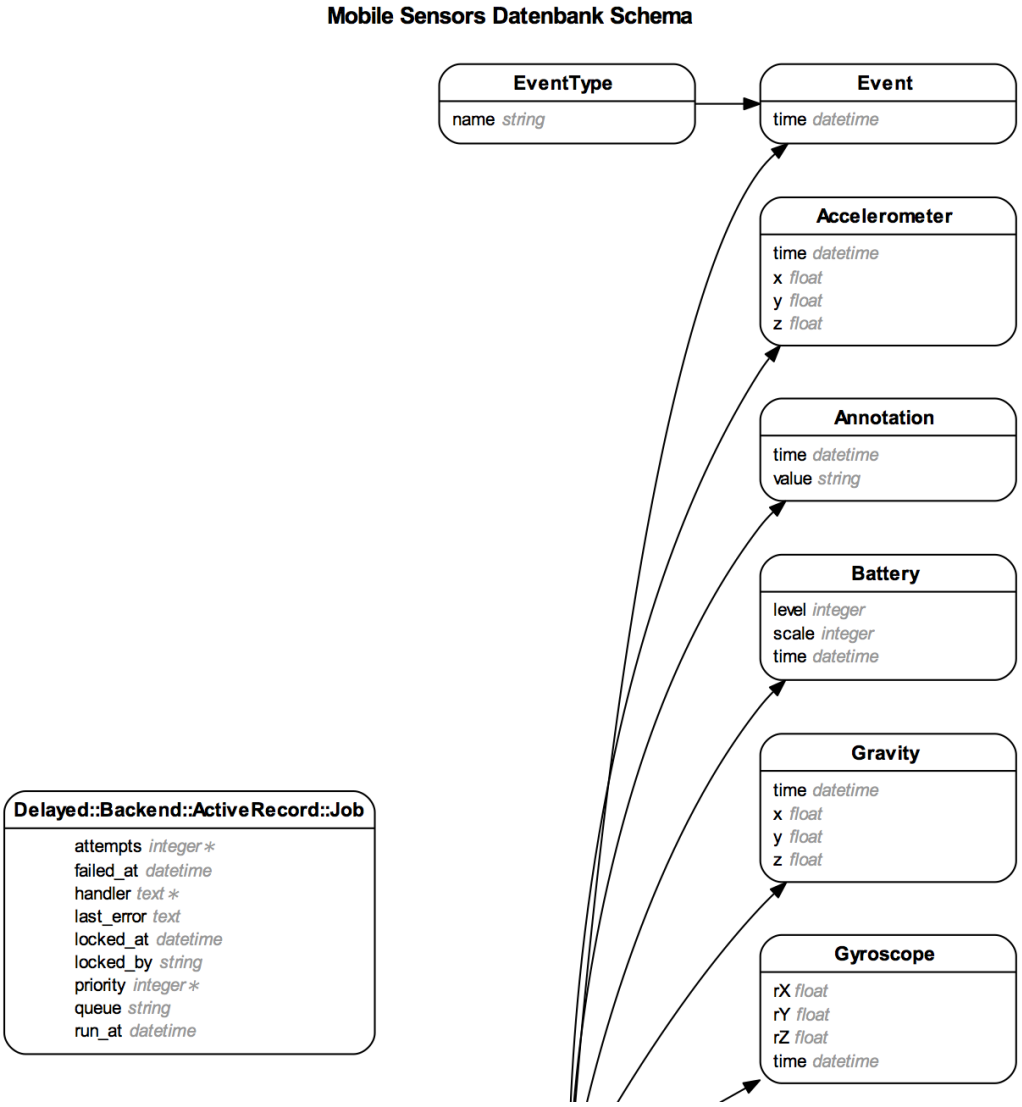

Abbildung 5.16.: Datenbank-Schema

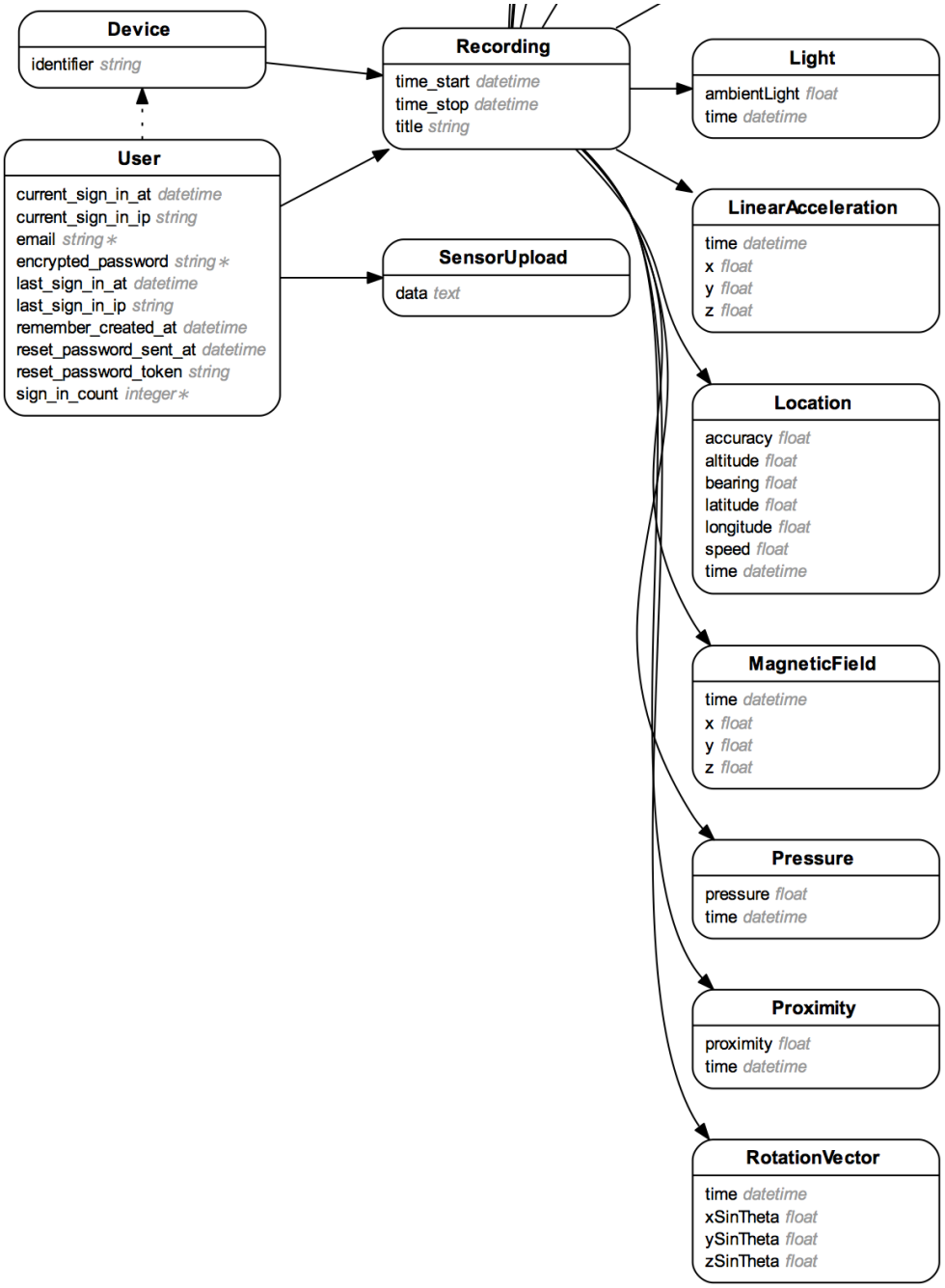

Abbildung 5.17.: Datenbank-Schema

5.2.3. Bootstrap

Bootstrap[4] ist ein von Twitter entwickeltes Front-End Framework, das 2011 unter einer Open-Source-Lizenz veröffentlicht wurde. Es enthält eine Vielzahl von auf HTML und CSS basierenden Gestaltungsvorlagen und optionale JavaScript-Erweiterungen.

Bootstrap unterstützt HTML5 und CSS3 und funktioniert dank dem Prinzip der progressiven Verbesserung[5] auch bei älteren Browsern. Durch den Einsatz von LESS-Stylesheets[6] wird die Wartbarkeit der in Bootstrap enthaltenen Stylesheets erhöht und Code-Wiederholungen werden vermieden. Seit Version 2.0 wird zudem das Erstellen von "responsiven" Webseiten unterstützt, d.h. der grafische Aufbau einer Seite erfolgt dynamisch anhand der Anforderungen des Gerätes, mit dem sie betrachtet wird.

Die einzelnen Komponenten von Bootstrap können in eingeschränktem Maße verändert werden. Um tiefgreifendere Änderungen vorzunehmen, muss die LESS-Deklaration überschrieben werden. Hieraus ergab sich für die Gestaltung der Mobile Sensing Webseite ein Nachteil, da detaillierte Änderungen der Benutzeroberfläche mit sehr hohem Aufwand verbunden waren.

5.2.4. Maps

Google Maps

Google Maps[7] stellt APIs zur Verwendung von Kartenmaterial und -funktionalität in Webseiten und mobilen Anwendungen zur Verfügung. Um eine der APIs verwenden zu dürfen, muss zunächst ein API Schlüssel angefordert werden, welcher anschließend in die Webseite oder mobile Anwendung eingebunden werden muss. Die Webseite oder mobile Anwendung, in der der Google-Dienst kostenlos genutzt wird, muss zudem frei zugänglich sein, d.h. er muss für Endnutzer kostenfrei und öffentlich erreichbar sein.

OpenCycleMap

OpenCycleMap[8] ist eine Weltkarte für Radfahrer, welche auf den Geodaten von OpenStreetMap basiert. Der Karte können die Standorte von Rad- und Fußwegen, Abstellmöglichkeiten für Fahrräder, Fahrradshops und Toiletten entnommen werden.

Das Projekt wurde im Jahre 2007 vom digitalen Kartographen Andy Allen ins Leben gerufen und auf seiner Thunderforest-Plattform[9] zusammen mit anderen Kartenstilen für die Nutzung in mobilen Anwendungen und Webseiten angeboten. Die Verwendung

[4] http://getbootstrap.com/

[5] Eine Methode im Webdesign, um eine Webseite auch für Geräte nutzbar zu machen, die nicht alle implementierten Funktionalitäten unterstützen. Dies wird erreicht, indem die grundlegendsten Informationen der Webseite für alle Geräte zugänglich gemacht werden, erweiterte Funktionalität jedoch nur dann zur Verfügung gestellt wird, wenn diese auch von dem vom Aufrufer verwendeten Gerät unterstützt wird.

[6] http://www.lesscss.de/

[7] https://maps.google.de/

[8] http://www.opencyclemap.org/

[9] http://www.thunderforest.com/

in Hobby-Projekten mit höchstens 150,000 geladenen Kartenkacheln im Monat ist kostenfrei. Für größere Projekte fällt eine monatliche Nutzungsgebühr an. Hierbei kann der Entwickler oder das Unternehmen zwischen 3 Tarifen wählen: Solo Developer, Small Business und Large Business.

Abwägung

Bei der Wahl des richtigen Kartendienstes lag das Augenmerk besonders auf kostenfreien und möglichst unkomplizierten Lösungen. Im Bereich der kostenfreien Kartendiensten waren vor allem Google Maps und OpenStreetMap zu untersuchen.

Während man für die Verwendung der Google Maps API einen API Schlüssel benötigt, wird für die Verwendung von OpenStreetMap keine Authentifizierung benötigt. Somit ist mithilfe von OpenStreetMap ein schnelleres und einfacheres Einbinden von Karten möglich.

Zudem gibt es eine Vielzahl von spezialisierten Karten, die auf den Rohdaten von OpenStreetMap basieren, darunter auch Karten für Radfahrer, wie z.B. die OpenCycleMap. Da sich diese Karte nicht nur schnell, einfach und kostenfrei einbinden lässt, sondern auch noch zusätzliche Informationen für Radfahrer enthält, war sie die erste Wahl für die Mobile Sensing Webseite.

5.2.5. Web Service

Der Web Service ist *RESTful* und im JSON Dialekt gehalten. Eine Ausnahme bilden Sensordaten, welche zusätzlich zu JSON als CSV exportiert werden können. Der Client muss sich dem Server gegenüber authentifizieren, bevor er Daten mit ihm austauschen kann.

5.2.6. Ruby-Java-Bridge

Wie bereits aus Abschnitt 5.1.1 bekannt, werden die vom Benutzer durch das Smartphone gesammelten Sensordaten an den Server übertragen, der diese wiederrum in der Datenbank speichert. Wie man Abschnitt 4.3.2 entnehmen kann, ist es notwendig die Sensordaten aus der Datenbank zu extrahieren und dem Klassifikator ein Intervall der Sensordaten zur Verfügung zu stellen, damit dieser entscheiden kann, ob ein Ereignis eingetreten ist oder nicht. Zusätzlich wird Java als Plattform gefordert.

Hinsichtlich der gerade erwähnten Punkte, war es notwendig eine Pipeline zu entwickeln, die kontinuierlich eine Aufnahme aus der Datenbank entnimmt, deren relevante Sensordaten in Fensterabschnitte unterteilt und an den Klassifikator zu Verarbeitung weiterleitet. Ursprünglich war es angedacht, diese Pipeline in Java zu entwickeln. Dieser Gedanke wurde jedoch verworfen, um eine Redundanz der Datenbanklogik zu vermeiden und zugleich das Einhalten des DRY-Prinzips zu gewährleisten.

Stattdessen wurde nach einer Schnittstelle zwischen Ruby und Java gesucht. Die *Ruby-Java-Bridge* [10] bietet eben eine solche Verbindung zwischen Ruby und Java. Die

[10] http://rjb.rubyforge.org/

```
$ curl -v -H "Accept: application/json" -H "Content-type:
    application/json" -c cookies.txt -X POST --data '{"user":{"
    email":"your@ema.il","password":"yourpassword"}}' 'http://
    localhost:3000/users/sign_in.json'


$ curl -v -H "Accept: application/text" -H "Content-type:
    application/text" -b cookies.txt -X POST --data @data.txt '
    http://localhost:3000/recordings/upload'

$ curl -v -H "Accept: application/json" -b cookies.txt -X GET
    http://localhost:3000/recordings.json

$ curl -v -H "Accept: application/json" -b cookies.txt -X GET
    http://localhost:3000/recordings/1.json

$ curl -v -H "Accept: application/csv" -b cookies.txt -X GET
    http://localhost:3000/recording/1/accelerometers.csv
```

Abbildung 5.18.: Curl-Anfragen

Implementierung einer Pipeline zur Übergabe der Sensordaten aus der Datenbank an den Klassifikator anhand der Ruby-Java-Bridge ist dennoch aus Zeitgründen ausgeblieben. Der Aufwand der Umsetzung ist hierbei höher ausgefallen, als erwartet.

5.3. Klassifizierung

Die ursprüngliche Entwicklung, der Klassifizierung von bestimmten Ereignissen, ist in der aktuellen Version kaum von Bedeutung. Jedoch wurde das Projekt in zwei Teilprojekte untergliedert. Das Ziel des ersten Projektes bestand in der Entwicklung einer leicht zugänglichen Klassifizierungs-Komponente für funktionierende Klassifizierer. Dieses Projekt wurde im späteren Entwicklungsprozess vollständig durch die Klassifizierung mittels Weka ersetzt. Das zweite Projekt (im Folgenden als *Testumgebung* bezeichnet) soll dem Entwickler von Klassifizierern jede mögliche Unterstützung für die Entwicklung und für das Testen bereits geschriebener Klassifizierer ermöglichen. In diesem Kapitel wird auf beide Teilprojekte eingegangen.

5.3.1. Implementierung des manuellen Klassifikators

Zu Beginn wurde versucht, die zuvor festgelegten Ereignisse, mittels selbst programmierter Funktionen zu erkennen. Bei dieser Herangehensweise stellten sich mehrere Nachteile heraus. Deshalb wurde im späteren Verlauf entschieden, dass diese Erkennung durch die Machine-Learning-API `Weka` ersetzt werden soll. Die Nachteile der selbst programmierten Funktionen sind:

1. komplizierte Erarbeitung von Erkennungsmerkmalen
2. schlechte Erweiterbarkeit von neuen Ereignissen
3. schlechte Erweiterbarkeit von neuen Endgeräten
4. ungenaue Ergebnisse bei der Erkennung von komplexen Bewegungen

Um Ereignisse durch Algorithmen bestimmen zu können, müssen die Messdaten, die während einem Ereignis aufgenommen werden, von den restlichen Messwerten, anhand bestimmter mathematischen Merkmalen, unterschieden werden. Diese Merkmale zu erkennen ist oft nicht trivial. Daraus ergibt sich der in Punkt 1. beschriebene Nachteil. Die in Punkt 2. und 3. erwähnten Nachteile, bezüglich der Erweiterbarkeit, sind damit zu begründen, dass für jedes neue Ereignis und für jedes neue Endgerät die Messwerte von verschiedenen Sensoren von dem Entwickler begutachtet und analysiert werden müssen. Der zuletzt genannte Punkt ergibt sich aus den drei vorhergehenden. Durch die komplexe Erarbeitung solcher Klassifizierer ergeben sich nicht vermeidbare Ungenauigkeiten. Alle genannten Nachteile dieser Vorhergehensweise sind bei der Realisierung von `Weka` nicht oder nur in einem tolerierbaren Maße vorhanden.

Anpassung der Messdaten

Um die Entwicklung von Erkennungs-Algorithmen zu ermöglichen und vereinfachen, können die Messdaten mit Methoden aus der Signalverarbeitung angepasst werden. Die Anpassung wird in zwei Schritten erläutert. Zuerst wird auf fehlerhafte Messwerte eingegangen, welche von manchen Testgeräten geliefert wurden. Anschließend wird erklärt, wie Messreihen angepasst werden, um eine Ereignis-Erkennung zu realisieren.

Fehlerhafte Messwerte entfernen. In Abbildung 5.19 werden fehlerhafte Messwerte in einem Diagramm dargestellt. Dabei wird der Messwert des Sensors durch die *Y-Achse* und der Aufnahmezeitpunkt durch die *X-Achse* dargestellt. Die blauen Punkte zeigen die ursprünglichen Messwerte, welche von einem Endgerät stammen. Die roten Punkte zeigen die korrigierten Messwerte. Dabei ist zu erwähnen, dass auf jedem roten Punkt auch ein Blauer ist. Dies ist in dem Diagramm leider nicht ersichtlich. Es wurden lediglich fehlerhafte Werte gelöscht. Der verwendete Algorithmus wurde empirisch ermittelt und basiert auf der Differenz von aufeinanderfolgenden Werten. Zusätzlich wurden alle Messwerte entfernt, die unter gegebenen Umständen nicht erreicht werden.

Abbildung 5.19.: Entfernung fehlerhafter Messwerte

Messwerte glätten. Wie bereits erwähnt, werden Messdaten angepasst, bevor sie für die Erkennung von Ereignissen eingesetzt werden. Der wichtigste Schritt ist hierbei das Glätten des Signals. Das liegt zum einen daran, dass der Entwickler schneller besonde-

re Merkmale eines Ereignisses aus zum Beispiel einem geplotteten Diagramm erkennen kann. Des Weiteren gibt es, besonders bei hochfrequenten gemessenen Sensoren (z.B. Accelerometer), ein hohes Maß an Rauschen. Um das Signal zu glätten, wurde sich hier für den *zentrierten gleitenden Mittelwert* entschieden, da die Implementierung einfach und das Ergebnis ausreichend für die weitere Verarbeitung ist [Smi02]. Die Abbildung 5.20 zeigt eine Aufnahme eines Accelerometers. Dabei werden die Messwerte durch die *Y-Achse* und der Aufnahmezeitpunkt durch die *X-Achse* dargestellt. Die blauen Punkte zeigen die Messwerte, nachdem fehlerhafte Messwerte entfernt wurden. Die roten Stellen den zentrierten, gleitenden Mittelwert der Ordnung 41 dar. Die Ordnung besagt, wie viele Werte für die Mittelwertberechnung genutzt werden. Die Eigenschaft *zentriert* besagt, dass der Mittelwert $\overline{y_i} := \frac{1}{n+1} \sum_{k=-\frac{n}{2}}^{n-\frac{n}{2}} y_{i+k}$ eines Messpunkt y_i durch vorherige und nachfolgende, sowie dem Messwert selbst berechnet wird. Die Formel wurde aus [KS10] entnommen.

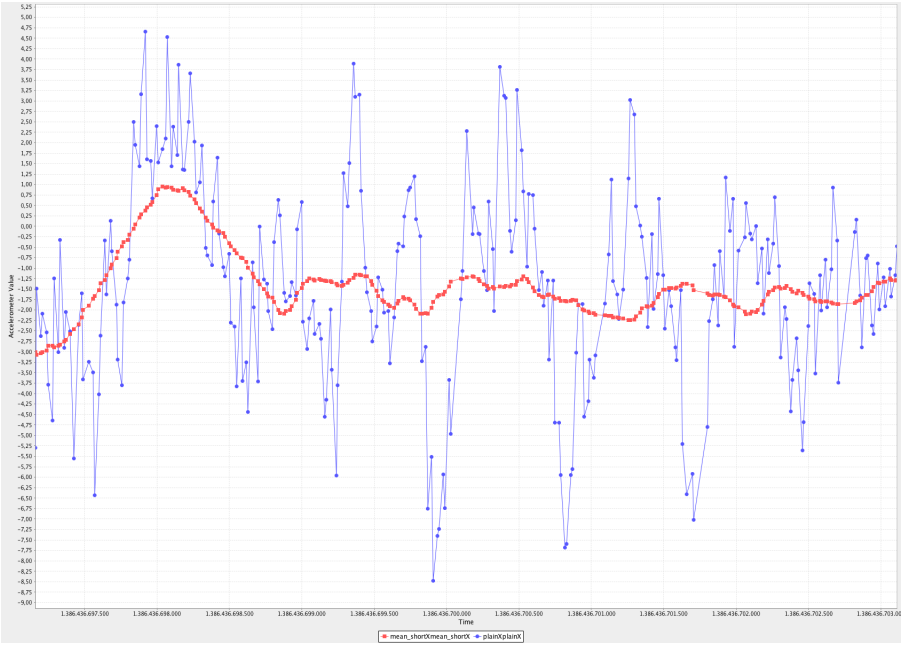

Abbildung 5.20.: Messwerte und zentrierter Gleitender Mittelwert im Vergleich

Erkennung eines Bremsvorganges

Für die Erkennung eines Bremsvorganges werden die Messwerte des GPS-Sensors verwendet. Der GPS-Sensor liefert die aktuelle Geschwindigkeit in Meter pro Sekunde. Die mobile Komponente liefert einen Messwert pro Sekunde. Um einen Bremsvorgang zu erkennen müssen, aufgrund der geringen Messfrequenz maximal drei aufeinanderfolgende Messwerte untersucht werden. Die Erkennung basiert darauf, dass an einem Messpunkt x_i entweder der Messwert des Messpunktes x_{i+1} oder x_{x+2} verhältnismäßig niedriger ist. Die Erkennung funktioniert sehr stabil und weist nur in sehr geringem Maße falsch-positive und falsch-negative Werte auf.

Erkennung eines Ausweichmanövers

Für die Erkennung von Ausweichmanövern werden Messwerte der X-Achse des Accelerometers untersucht. Das Accelerometer gibt die Beschleunigung des Gerätes zu einem Zeitpunkt je Raumachse an. Die Abbildung 5.21 zeigt die Messwerte des Accelerometer bezüglich der X-Achse. Diese Achse des Accelerometer ist orthogonal zur Längsseite des Endgerätes. Der Erkennungsalgorithmus benutzt ein *Sliding-Window-Verfahren*. Dabei

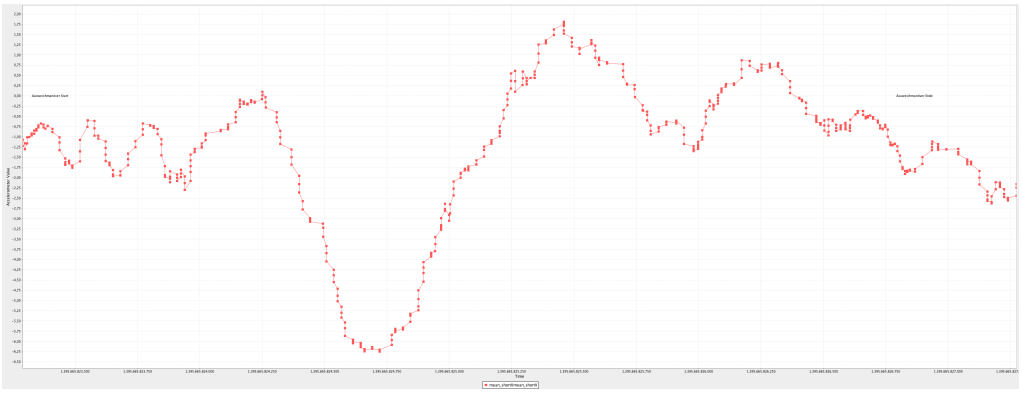

Abbildung 5.21.: Accelerometer-Messwerte der X-Achse während eines Ausweichmanövers

wird eine bestimmte Menge von Messwerten nach einem bestimmten Messpunkt untersucht. Alle Messwerte in einem *Window* müssen eine vordefinierte Eigenschaft aufweisen. Es müssen Beispielsweise alle Messwerte kleiner sein, als der aktuelle Messpunkt. Das Verfahren ist die Grundlage des Algorithmus' und versucht die aufeinanderfolgenden Bewegungen, welche bei einem Ausweichmanöver auftreten, zu erkennen. Jedoch besteht die Schwierigkeit darin, dass unterschiedliche Endgeräte existieren und die Messwerte und Messfrequenz des Accelerometers sehr unterschiedlich ist. Deshalb wurde die Implementierung dieses Klassifikators nicht perfektioniert, weil auf *Machine-Learning* umgestiegen wurde.

5.3.2. Implementierung der Testumgebung für Klassifikatoren

Die Testumgebung hilft einem Entwickler bei Erstellung von manuellen Klassifikatoren. Außerdem werden Werkzeuge für die Klassifizierung mittels Weka bereitgestellt.

Anforderungen

1. **Einlesen von Sensordaten**
   1.1. **Einlesen von CSV-Dateien**
   Die Dateien müssen im CSV-Format eingelesen werden können
   1.2. **Kommunikation mit der Server-Komponente**
   Die Daten müssen direkt von der Server-Komponente bezogen werden können
2. **Deserialisieren der Eingabedaten**
   Die eingelesenen Sensordaten müssen in eine geeignete Objektstruktur übertragen werden.
3. **Ausgabe in geeigneten Formaten**
   Die deserialisierten Sensordaten müssen in ein geeignetes Ausgabeformat gebracht werden.
   3.1. **Ausgabe als Chart-Bild**
   Um die Sensordaten für den Entwickler in einer gut lesbaren Form darzustellen, müssen Charts aus den Sensordaten gerendert werden können.
   3.2. **Ausgabe von getaggten Ereignissen als CSV-Dateien**
   Die Klassifizierung benötigt für jedes vom Benutzer markierte Ereignis eine CSV-Datei, damit das markierte Ereignisse gelernt werden kann. Die Testumgebung soll alle markierten Ereignisse aus einer Aufnahme extrahieren.

Realisierung

Die Testumgebung erfüllt alle Anforderungen. Im weiteren Verlauf wird auf die Realisierung von den wichtigsten Bestandteilen eingegangen.

Kommunikation mit der Server-Komponente Wie bereits in Kapitel 5.2.5 beschrieben, wird mit dem Server über einen *RESTful-Webservice* kommuniziert. Daraus ergeben sich mehrere Vorteile.

1. **Einfach gestaltete Kommunikation**
   Kommunikation über *HTTP-Requests* und *HTTP-Responses*.
2. **Einfache Deserialisierung mittels CSV**
   Der *Webservices* bietet die Sensordaten im *CSV-Format* an. Eine vom Server empfangene CSV-Datei kann durch einen *CSV-Parser* geparst und deserialisiert werden.

3. **Datenschutz**
   Der *Webservice* erfordert einen Login bestehend aus einem Benutzernamen und Passwort, sodass ein autorisierter Zugang sichergestellt ist.

Um eine leichte Bedienung für Entwickler des *Webservices* sicherzustellen, stellt die Testumgebung Methoden für die Kommunikation mit dem Server bereit. So ist es Möglich mit einem Methodenaufruf sich auf dem *Webservice* einzuloggen oder deserialisierte Sensorobjekte zu beziehen.

Ausgabe von CSV-Dateien für die Weka-Klassifizierung Für die Klassifizierung mittels *Weka* ist es notwendig, dass die Messwerte einzelner Ereignisse in einer CSV-Dateien vorliegen. So wird bei der Aufzeichnung ein Ereignis durch den Benutzer mittels Annotationen eingegrenzt. Das bedeutet, dass auf der Mobilen-Komponente vor und nach jedem Ereignis ein Knopf gedrückt werden muss. Dabei wird je eine Annotation mit einem Zeitstempel versehen und gespeichert. Durch die Zeitstempel der Annotationen werden die, während des Ereignisses aufgenommenen, Sensordaten extrahiert und als Ereignisaufnahme deklariert. Für jedes gekennzeichnete Ereignis in einer Aufnahme wird eine CSV-Datei gespeichert. Die Sensordaten, welche nicht während eines Ereignisses aufgezeichnet wurden, können auch extrahiert werden und sind für die Klassifizierung in *Weka* relevant. So werden die Sensordaten einer Aufnahme als *Trainingsdaten* klassifiziert und in *Ereignisse* und *Nicht-Ereignisse* markiert.

Nutzung der Testumgebung Da die Testumgebung Methoden zur Http-Kommunikation zum Server beinhaltet, konnte die Klassifierungs-Komponente mit bereits gelernten Modellen gegen neue Aufnahmen getestet werden. Dabei wurden Aufnahmen deserialisiert und in gleich große Intervalle (im weiteren Verlauf *Window* genannt) geteilt. Diese Windows wurden mittels der Fassade der Klassfizierungs-Komponente auf Ereignisse untersucht. Dadurch ist es möglich die Klassifikations-Komponente zu evaluieren.

6. Erweiterte Aspekte: Energieeffizienz-Analyse

Dies ist die schriftliche Ausarbeitung einer Untersuchung zum Energiebedarf von mobilen Geräten. Es wird im Speziellen der Energiebedarf eines Komprimierungsvorgangs ermittelt. Ihm wird der Energiebedarf eines Sendevorgangs von Daten über ein Funknetzwerk gegenübergestellt. In der Untersuchung zeigt sich, dass sich bei mobilen Geräten in Bezug auf den Energiebedarf nahezu immer lohnt, Daten vor dem Versenden zu komprimieren.

6.1. Einleitung

Bei der Untersuchung des Energiebedarfs waren grundsätzlich verschiedene Aufgaben zu bewältigen:

- Ermittlung der Parameter, welche maßgeblich den Energiebedarf für das Versenden und Komprimieren von Daten auf einem mobilen Gerät bestimmen
- Bereitstellung von Testdaten
- Entwicklung jeweils einer Applikation, welche Daten komprimiert und versendet
- Bereitstellung einer Methode, um den Energiebedarf einer Applikation zu messen
- Durchführung des Experiments
- Auswertung und Interpretation der erhobenen Daten

In den einzelnen Abschnitten wird im Detail auf die einzelnen Aufgaben eingegangen und die allgemeine Herangehensweise an die Problemstellung beschrieben.

6.2. Testdaten

Die Fragestellung gehandelt den Energiebedarf von komprimierten und nicht komprimierten Daten. Daher ist eine heterogene Grundmenge an Testdaten nötig, um objektive Betrachtungen zu ermöglichen.

6.2.1. Anforderungen

Annahmen:

- Der Energiebedarf eines Programms ist abhängig von der Anzahl und Art der Rechenschritte, welche es durchzuführen hat
- Je mehr Daten ein Komprimierungsalgorithmus abzuarbeiten hat, desto mehr Rechenschritte benötigt er
- Je kleiner die Entropie der zu komprimierenden Daten ist, desto weniger Speicherplatz benötigen die entsprechenden komprimierten Daten im Verhältnis
- Je mehr Daten über ein Funknetzwerk versendet werden, desto mehr Energie benötigt der Sendevorgang

Aus den Annahmen folgt, dass für die Testdurchläufe sowohl Daten mit unterschiedlichem Speicherbedarf als auch mit unterschiedlicher Entropie bereitzustellen sind.

6.2.2. Begriff der Entropie

Der Begriff der Entropie geht zurück auf Claude Elwood Shannon. Nach ihm ist die Entropie H über dem Alphabet $Z = \{z_1, z_2, ..., z_m\}$ definiert durch:

1. Für ein einzelnes Zeichen

$$H_1 = -\sum_{i=1}^{m} p_i \cdot log_2 p_i$$

   p_i entspricht der Auftrittswahrscheinlichkeit des Zeichens p_i

2. Für ein Wort der Länge n

$$H_n = -\sum_{w \in Z^n} p_w \cdot log_2 p_w$$

   p_w entspricht der Auftrittswahrscheinlichkeit des Wortes w

Die Entropie ist interpretierbar als ein Maß für den Informationsgehalt eines gegebenen Datensatzes. Besitzt ein Datensatz eine große Entropie, so hat er seinem Speicherbedarf entsprechend einen großen Informationsgehalt. Komprimierte Daten weisen also im Allgemeinen eine höhere Entropie auf als unkomprimierte Daten. Besitzen Daten ein hohes Maß von sich wiederholenden Elementen, so ist ihre Entropie entsprechend geringer.

6.2.3. Messung der Entropie

Wir berechnen die Entropie der Daten mit dem gemeinfreien Unix-Programm *ent* von John Walker.

6.2.4. Erzeugung der Daten

Die Testdaten werden byteweise generiert. Jedes Byte setzt sich aus 8 unabhängig erzeugten Zufalls-Bits zusammen. Dabei dient ein Float-Wert zwischen 0 und 1, der einmal für den gesamten Testdatensatz festgelegt wird, als Moderator. Dieser dient dazu, die Entropie der Testdaten zu regulieren. Für jedes Bit wird ein zufälliger Float-Wert zwischen 0 und 1 mit dem Moderator verglichen. Ist der Moderator größer als die Zufallszahl, wird das Bit auf 1 gesetzt, sonst auf 0. Nachfolgend ist der Algorithmus dargestellt, welcher die Testdaten erzeugt.

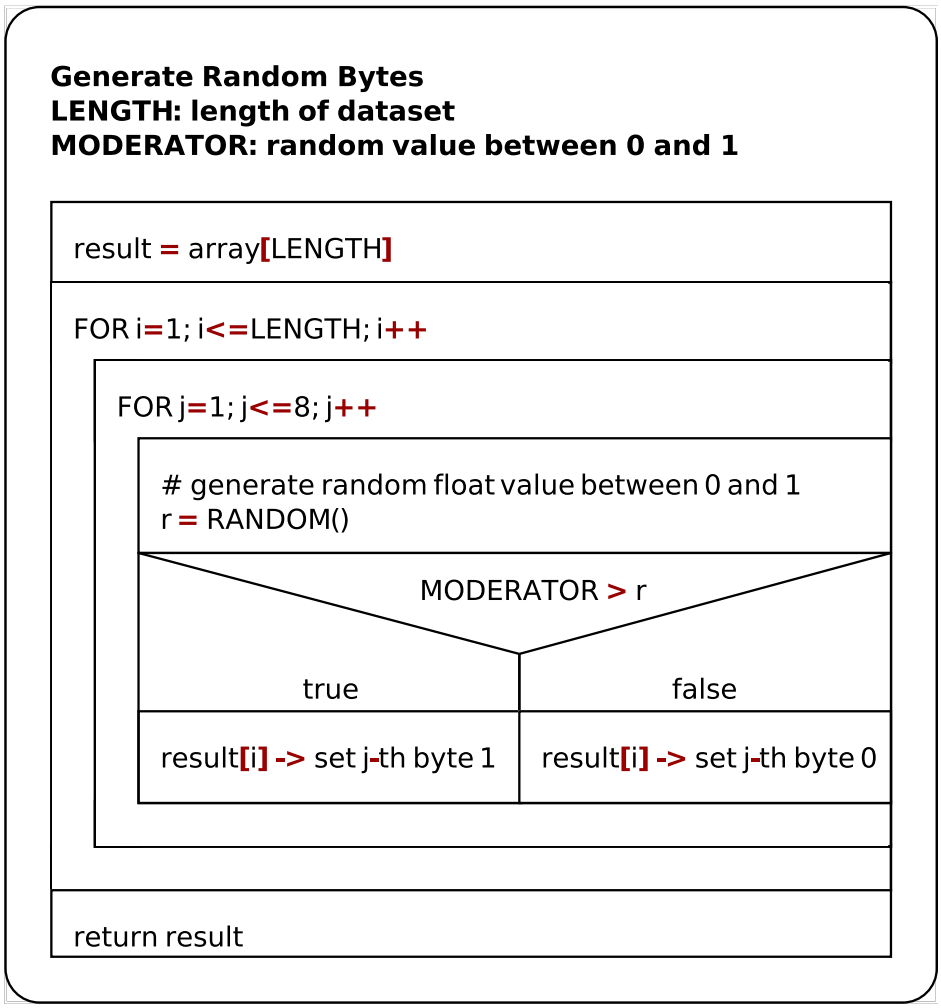

Abbildung 6.1.: Nassi-Shneidermann-Diagramm für Datenerzeugung

Entscheidet sich der Wert jedes Bits nach diesem Verfahren, lässt sich nach dem *Gesetz der großen Zahlen* die Entropie des Datensatzes aus dem Wert des Moderators herleiten. Wir setzen p als Zufallsvariable für die Wahrscheinlichkeit, dass das Bit den Wert 1 annimmt und q dafür, dass das Bit den Wert 0 annimmt. Die Wahrscheinlichkeit dieser beiden Ereignisse ist in der Summe immer 1. Es gilt: $p + q = 1$. Setzen wir p und q in die Definition der Entropie ein und setzen wir für $q = p - 1$, so erhalten wir:

$$H(p) = -8 \cdot (p \cdot log_2 p + (1 - p) \cdot log_2(1 - p))$$

Der Faktor 8 gibt die Entropie statt in Bit/Bit in Bit/Byte an.

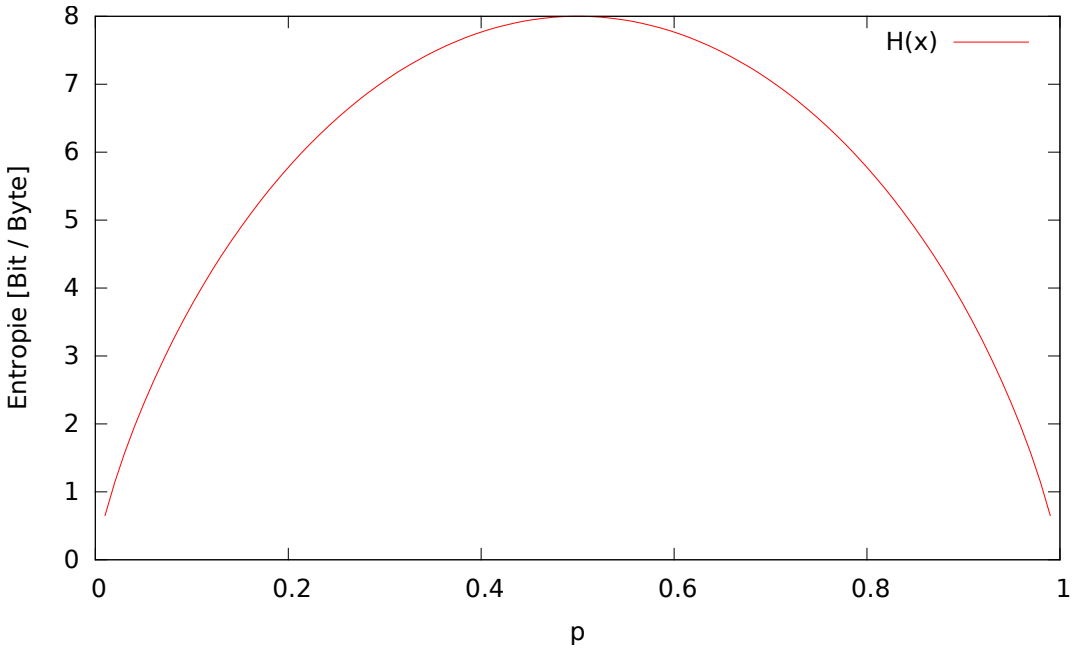

Abbildung 6.2.: Abhängigkeit zwischen Zufallsvariable und Entropie

Es zeigt sich, dass die Zufallsvariable bei dem Wert 0,5 ihr Maximum von 8 Bit/Byte erreicht. Bei 0 liegt die Entropie bei 0. Die Funktion ist achsensymmetrisch bei $p = 0,5$. Es genügt also, für eine Entropie von 0 bis 8 Bit/Byte die Zufallsvariable von 0 bis 0,5 laufen zu lassen.

Es ist sinnvoll, die Entropie der verschiedenen Testdaten linear festzugelgen. So ergeben sich für die Zufallsvariable p folgende Werte:

Entropie [Bit/Byte]	p	Verprobung mit *ent* [Bit/-Byte]
0	0	0.000000
1	0.0171286550767266	1.000176
2	0.0416926902736567	2.000160
3	0.0724497922261490	3.005329
4	0.110027864438360	4.001996
5	0.156142171816185	5.001030
6	0.214501744859829	5.999288
7	0.294926193599964	6.999386
8	0.5	7.999812

Die Zufallsvariable p dient dem Algorithmus in Abbildung 6.1 als Moderator. Mit ihm wurden Testdaten mit einem Umfang von 1.000.000 Bytes erstellt. Anschließend wurde mit *ent* verprobt.

6.3. Messung des Energieverbrauchs

Für die Messung des Energieverbrauchs wurden Anforderungen erhoben. Mit deren Hilfe wurde eine Software gesucht, die deren Ansprüchen genügt.

6.3.1. Anforderungen

Es ergaben sich folgende Anforderungen an die Software:

- Erhebung des Energieverbrauchs pro PID [1]
- Energieverbrauch in der Einheit: mJ
- Trennung des Energieverbrauchs in: CPU, 3G, WIFI
- Speicherung der erhobenen Daten
- Verlässlichkeit der erhobenen Daten

6.3.2. Powertutor

Powertutor[2] basiert auf einem Energiemodell [LZ11], welches den Energieverbrauch verschiedener Hardware unterschiedlich berücksichtigt.

- CPU: Auslastung und Frequenz-Level
- OLED/LCD: Helligkeit und ggf. Pixel Informationen

[1] Prozess ID
[2] http://www.powertutor.org

- Wifi: Uplink-Durchsatz, Uplink-Daten-Rate und Pakete/Sekunde
- 3G: Pakete/Sekunde und Power-State
- GPS: Anzahl der entdeckten Satelliten, Power-State
- Audio: Power-State

Eine Recherche ergab, dass Powertutor allen gestellten Anforderungen genügt. Powertutor ist eine Applikation für Android, welche von der Universität von Michigan entwickelt wurde. Die offizielle Weiterentwicklung von Powertutor wurde im Jahre 2013 eingestellt. Deshalb war es nötig einen Patch zu schreiben, der einen Fehler bei der Energieverbrauchsermittlung des WLAN-Moduls behebt. Die notwendige Neukompilierung erlaubte zusätzlich eine Anpassung der erstellten Log-Datei. Darin wurde der Stromverbrauch der einzelnen Hardwarekomponenten von jeder ausgeführten Applikationgespeichert. Durch hinzugefügte Timestamps wurde die Ermittlung des Energieverbrauchs innerhalb eines bestimmten Zeitfensters realisiert. Die gepatchte Version ist beziehbar über den Google-Play-Store[3].

[3]https://play.google.com/store/apps/details?id=com.henny.PowerTutor2

Abbildung 6.3.: Screenshot von Power Tutor 2

6.4. Durchführung

Für die Durchführung der Messvorgänge wurden Testdaten mit dem in Abschnitt 6.2 beschriebenen Verfahren erzeugt. Es wurden Dateien der Entropien 1 bis 8 Bit/Byte und der Dateigrößen 1,2,4,8,16 MB erstellt. Als Testgerät wurde ein Nexus 5[4] genutzt.

[4]http://www.google.de/nexus/5

6.4.1. Powertutor

Powertutor 2 lief während der Datensammlung im Hintergrund, um den Energieverbrauch in einer Log-Datei zu speichern. Powertutor 2 speichert jede Sekunde folgende Daten in einer Log-Datei:

```
time 1396787072113
begin 8
total-power 515
LCD-TOTAL-0-Kernel 0
LCD-TOTAL-1017-sys_1017 0
LCD-TOTAL-10105-ZipTest 290
...
CPU-TOTAL-0-Kernel 487
CPU-TOTAL-1017-sys_1017 0
CPU-TOTAL-10105-ZipTest 772
...
WIFI-TOTAL-0-Kernel 0
WIFI-TOTAL-1017-sys_1017 0
WIFI-TOTAL-10105-ZipTest 0
...
3G-TOTAL-0-Kernel 0
3G-TOTAL-1017-sys_1017 0
3G-TOTAL-10105-ZipTest 0
```

In der ursprünglichen Version von Powertutor wird der Zeitpunkt und nur der gesamte Stromverbrauch einer Applikationgespeichert. Die unternommenen Anpassungen ermöglichen eine Berechnung des Stromverbrauchs jeder einzelnen Komponente.

6.4.2. Test-Applikationen

Zum Messen des Energieverbrauchs, welcher durch das Komprimieren oder Versenden der Daten entsteht, wurden je eine Applikation entwickelt. Dabei wurden vor und nach jedem Testlauf folgende Daten in eine Log-Datei geschrieben:

```
startSending 1396621263821 /storage/emulated/0/20971_1entropy.data
finishedSending 1396621590402 /storage/emulated/0/20971_1entropy.data
startSending 1396621590404 /storage/emulated/0/10485_1entropy.data
finishedSending 1396621779307 /storage/emulated/0/10485_1entropy.data
...
```

Der grundsätzliche Ablauf einer Test-Applikation wird in Abbildung 6.4 dargestellt. Wie bereits erwähnt , wurde ein Verzeichnis erstellt, welche alle Testdateien enthält. Für jede Datei in diesem Verzeichnis wird eine Methode `execute` aufgerufen. Der Parameter der Methode enthält eine Datei aus dem Verzeichnis. In der Methode `execute` wird, je nach Applikation, eine Datei gepackt oder über das Internet versendet. Bei der Auswertung der Messwerte wurden die Dateinamen der Test-Daten verwendet, um die Attribute der Test-Dateien zu bestimmen. Die erste Zahl im Dateinamen gibt die Größe der Datei in Byte an. Die zweite Zahl steht für die Entropie in Bit pro Byte.

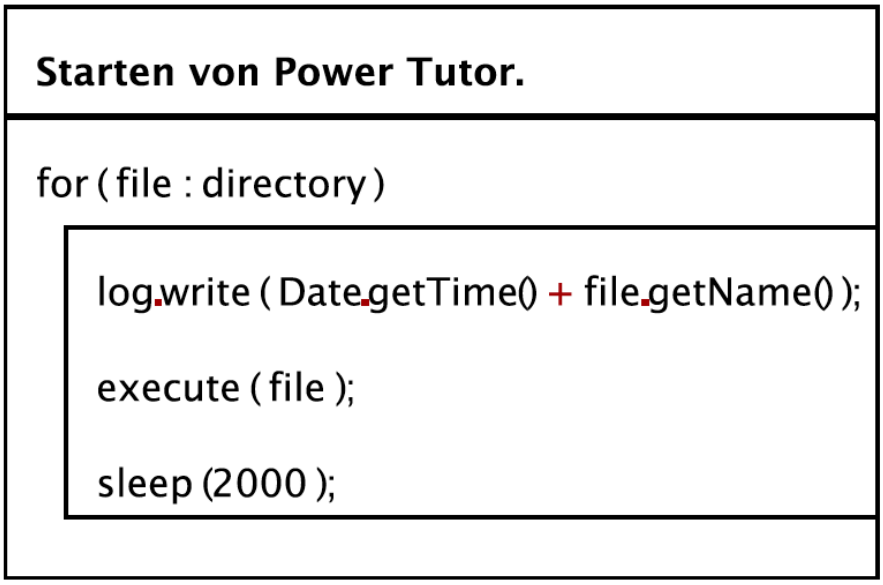

Abbildung 6.4.: Ablauf einer Test-App.

6.4.3. Parsen der Log-Daten

Um die aus den Test-Applikationen entstandenen Log-Dateien auszuwerten, wurde ein Programm entwickelt, welches mit Hilfe der Log-Dateien von Powertutor 2 den Stromverbrauch jedes Testlaufs errechnet. Dabei werden die Start- und Endzeitpunkte der Applikationen ausgelesen und der Stromverbrauch in dieser Zeitspanne mit Hilfe der Log-Datei von Powertutor 2 errechnet. Die Ausgabe des Programms ist eine CSV-Datei, welche alle Informationen enthält, die für eine weitere Auswertung in Excel benötigt werden. Eine CSV-Datei für die Auswertung des Stromverbrauch der Komprimierungsapplikation ist wie folgt aufgebaut:

`Dateigröße in Byte, Entropie in Bit pro Byte, Komprimierungs-Stufe, Dateigröße nach Komprimierungsvorgang, WIFI-Verbrauch, CPU-Verbrauch, LCD-Verbrauch, Ausführungsdauer`.

Beispiel einer Ausgabe-Datei:

```
...
16777216 , 3 , 7 , 7614 , 0.0 , 13391.0 , 11875.0 , 25497
16777216 , 4 , 7 , 9634 , 0.0 , 20320.0 , 15675.0 , 32701
...
```

6.5. Auswertung

Die in den Tests gesammelten Daten zeigen, dass der Energieverbauch in den einzelnen Tests abhängig von unterschiedlichen Faktoren ist. Für den Komprimierungstest spielt

die Größe und die Entropie der Datei, sowie der benutzte Komprimierungsalgorithmus eine Rolle. Das Verschicken hingegen ist von der Entropie unabhängig. Beim Verschicken hängt der Energieverbrauch von Dateigröße und Übertragungsgeschwindigkeit ab, die sich auf die Dauer des Sendevorgangs auswirken.

6.5.1. Komprimierungstest

Bei dem Komprimierungstest wurde der Energieverbauch in Abhängigkeit von Größe und Entropie der Datei, sowie in verschiedenen Komprimierungstufen analysiert. Die Effizienz der unterschiedlichen Verfahren unterscheidet sich kaum. Bei einer Entropie von 1 Bit pro Byte liegt die Kompressionsrate bei der schnellsten Kompression bei 77%. Für die beste Kompression liegt die Kompressionsrate bei 83%. Je höher die Entropie der Ausgangsdatei ist, desto geringer sind die Unterschiede in der Kompressionsrate bei unterschiedlichen Kompressionsstufen. Bei einer Entropie von 7 Bit pro Byte liegt die Differenz bei lediglich 1%. Für eine Entropie von 8 Bit pro Byte kann die Ausgangsdatei nicht weiter komprimiert werden, wodurch für alle Kompressionsstufen die Kompressionsrate bei 0% liegt (Abbildung 6.5).

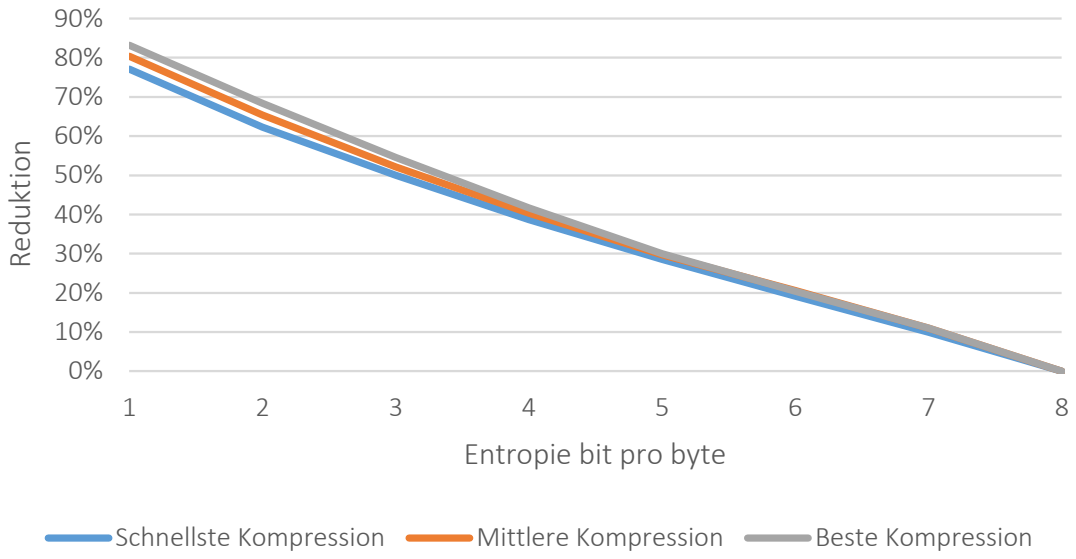

Abbildung 6.5.: Kompressionsraten für unterschiedliche Kompressionsmethoden

Bei einer 16MB großen Ausgangsdatei liegt der maximale Unterschied, der bei einer Entropie von 1 erreicht wird bei 0.96MB. Demnach würde sich die Methode mit der besten Kompression für das Komprimieren besonders bei größeren Dateien lohnen. Ein weiterer Aspekt, der betrachtet werden muss, ist die Dauer, die der Prozess für die

Komprimierung einer Datei benötigt. Der Unterschied zwischen dem schnellsten Kompressionsmethode und der Methode mit der besten Kompressionsrate ist hier gerade bei mittleren Entropiewerten deutlich. Bei einer 16MB Datei benötigt die schnellste Methode bei einer mittleren Entropie von 3 Bit pro Byte rund vier Sekunden. Die Methode mit der besten Komprimierungsrate benötigt mehr als zwei Minuten länger und spart dabei 4% ein (Abbildung 6.6). Die Dauer der schnellsten und der mittleren Komprimiermungsmethode liegen recht dicht beieinander und unterscheiden sich deutlich von der Rechendauer der besten Komprimierungsmethode. Dies lässt sich mit der Paretoregel beschreiben, die besagt, dass 80% der Ergebnisse in 20% der Zeit geschafft werden [Par]. Eine mittlere Kompressionsstufe benötigt nur vier Sekunden länger als die schnellste Methode und spart knapp 2% ein (Abbildung 6.7).

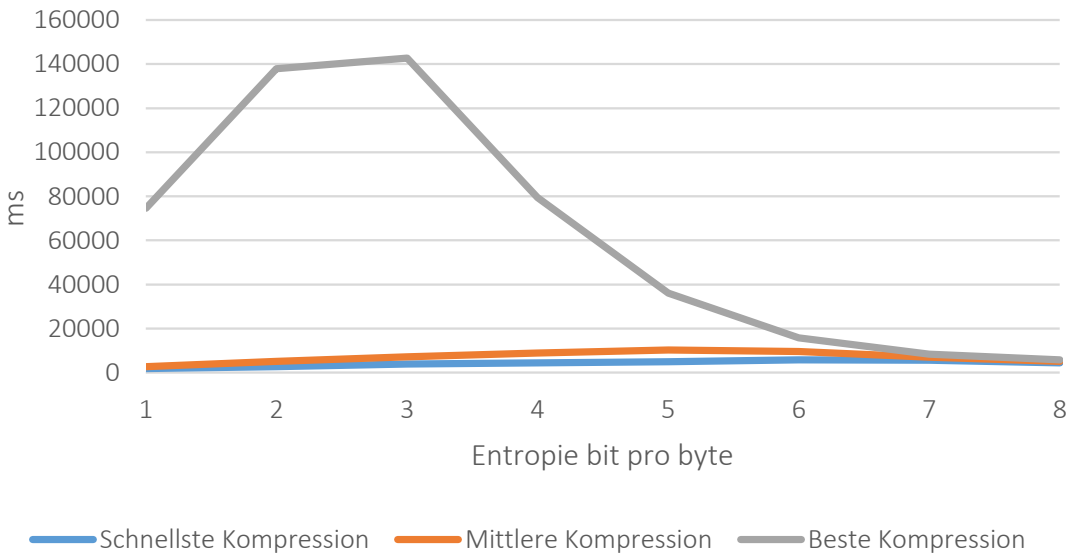

Abbildung 6.6.: Dauer für Kompressionsvorgang einer 16MB großen Datei

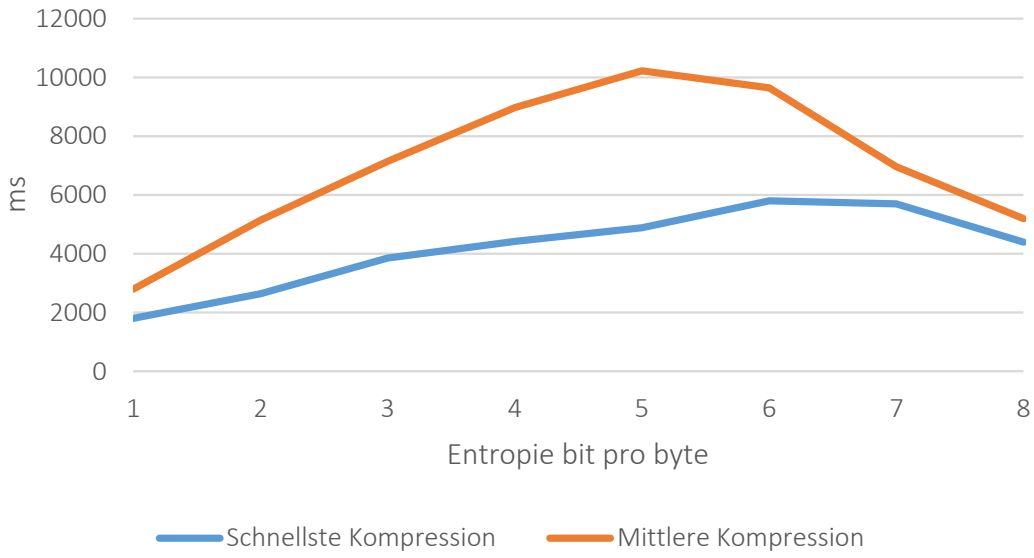

Abbildung 6.7.: Dauer für Kompressionsvorgang einer 16MB großen Datei (Vergrößert)

Bei einer hohen Entropie von 7 oder 8 Bit pro Byte sind alle Kompressionsmethoden schneller als 10 Sekunden. Jedoch wird hier kein Unterschied in der Kompressionsrate durch die unterschiedlichen Methoden erreicht. Die Gesamtdauer des Komprimierungsvorganges hat Einfluss auf den Energieverbrauch des Prozesses. Die beste Komprimierungsmethode benötigt sogar knapp ein Watt mehr als die schnellste oder eine mittlere Komprimierungsmethode, welche recht ähnlich bei 275mW liegen (Abbildung 6.8). Der Energieverbrauch ist unabhängig von der Entropie und wird nur von der gewählten Komprimierungsmethode beeinflusst. Die beste Komprimierungsmethode verbraucht konstant 1300mW, unabhängig von der Entropie und von der zur Komprimierung gewählten Dateigröße der Ausgangsdatei (Abbildung 6.9).

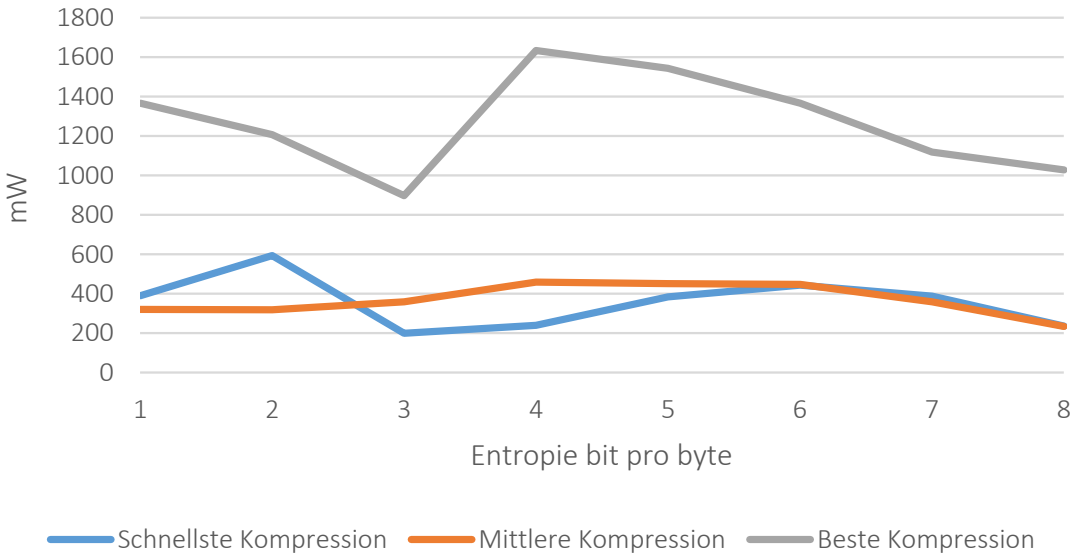

Abbildung 6.8.: Energieverbauch bei Komprimierung einer 16MB großen Datei

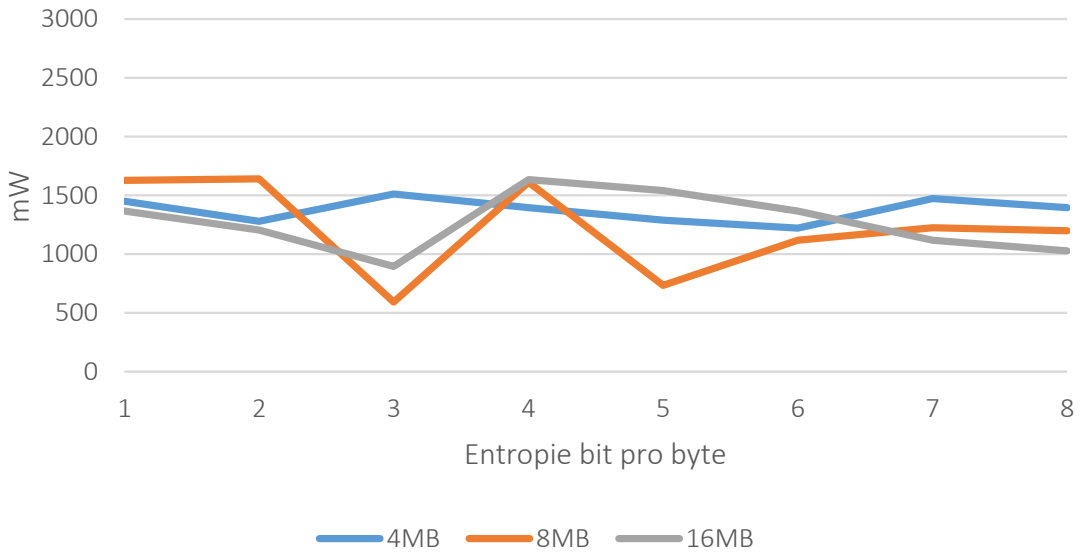

Abbildung 6.9.: Energieverbauch bei unterschiedlichen Dateigrößen (Beste Kompression)

Anhand dieser Ergebnisse ist die schnellste Kompression die Kompressionsmethode, die für die weiteren Auswertungen verwendet wird, da sie bezogen auf den Energieverbrauch die bessere Wahl ist. Die schnellste Kompression ist schneller, verbraucht weniger Energie pro Sekunde und erreicht eine nahezu ähnliche Kompressionsrate. Die Dauer, und somit der Gesamtenergieverbrauch für die schnellste Kompression, ist abhängig von der Größe der Ausgangsdateien und deren Entropie. Eine 4MB große Datei mit einer Entropie von 4 Bit pro Byte benötigt knapp eine Sekunde für die Komprimierung. Eine 8 MB große Datei benötigt bei einer Entropie von 4 knapp zwei Sekunden. Für die gleiche Entropie benötigt eine 16MB große Datei knapp 4 Sekunden (Abbildung 6.10).

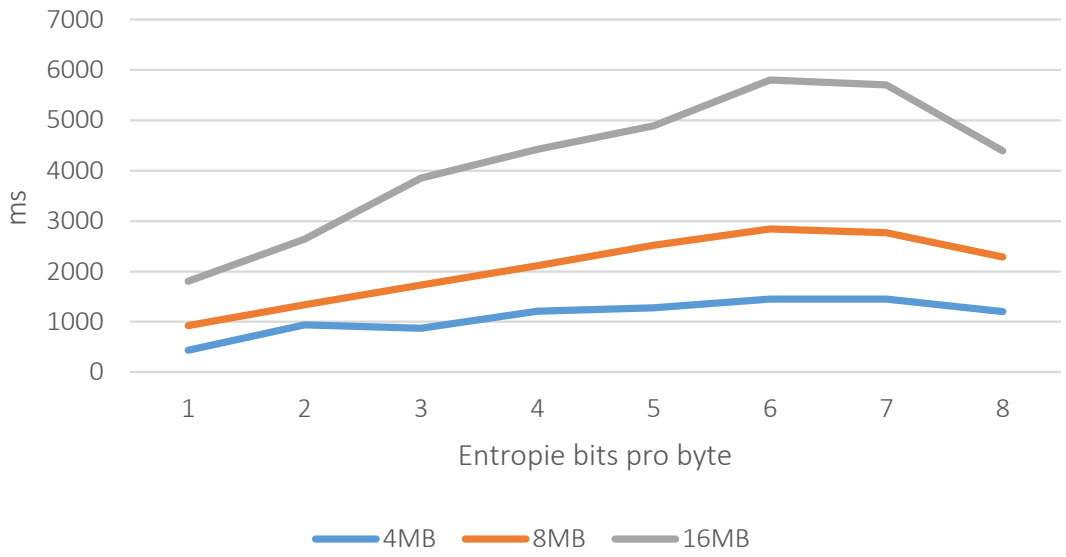

Abbildung 6.10.: Dauer für schnellste Kompressionsmethode

Damit eine Aussage für die Dauer und den damit verbundenen Energieverbauch unabhängig von der Ausgangsdateigröße getroffen werden kann, muss eine Abhängigkeit gefunden werden. Die Dauer hat eine lineare Abhängigkeit von der Dateigröße für die schnellste Kompressionsmethode (Abbildung 6.11). Pro MB benötigt die schnellste Komprimierungsmethode somit je nach Entropie minimal 110ms bei einer Entropie von 1 Bit pro Byte und maximal 360ms bei einer Entropie von 6 Bit pro Byte.

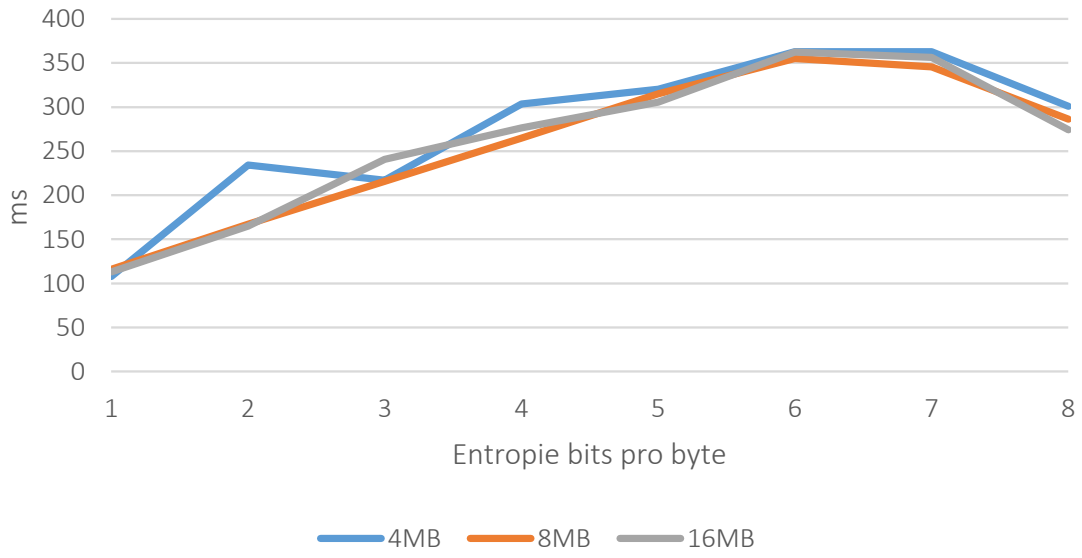

Abbildung 6.11.: Dauer für schnellste Kompressionsmethode (pro MB)

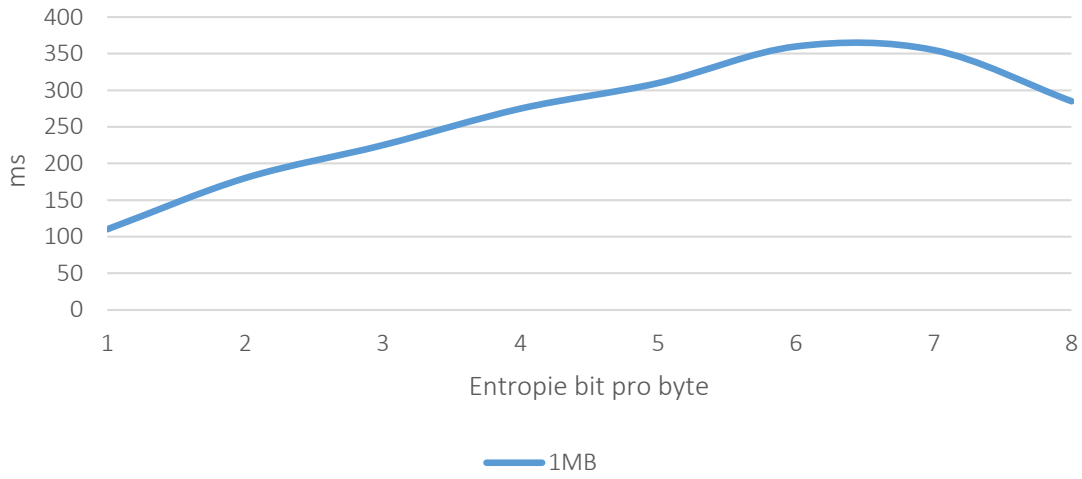

Abbildung 6.12.: Dauer pro MB

Eine geglättete Kurve aus dem Durchschnitt der 3 Kurven von vier, acht und sechzehn

MB (Abbildung 6.12) dient als Grundlage für die aus den Ergebnissen abgeleitete Formel. F steht für die Größe der Ausgangsdatei in MB, t ist die Dauer für die Komprimierung anhand 6.12 und c die Komprimierungskosten für die gewählte Komprimierungsmethode (Abbildung 6.8). Die Variable t ist abhängig von der Entropie der Ausgangsdatei und kann aus der Tabelle 6.2 auf Seite 87 oder aus der Abbildung 6.12 entnommen werden. Die Variable c ist für die schnellste Kompressionsmethode 275.

$$Zip(F, t, c) = F \cdot t \cdot c/1000$$

6.5.2. Send-Test

Beim Send-Test wurde überprüft, ob die Entropie einer Datei Einfluss auf die Dauer und somit den Energieverbrauch beim Versenden über mobiles Internet und Wireless hat. Die Entropie einer Datei hat keinen Einfluss auf die Dauer beim Versenden. Beim Versenden über Wireless benötigen Dateien unterschiedlicher Größe und unterschiedlicher Entropie immer 175mW (Abbildung 6.13). Die kleinen Schwankungen in den Grafiken sind auf Messungenauigkeiten zurückzuführen. Es wurden mehrere Messungen durchgeführt und daraus der Durchschnitt berechnet. Die Werte schwanken alle um 175mW +- 10mW.

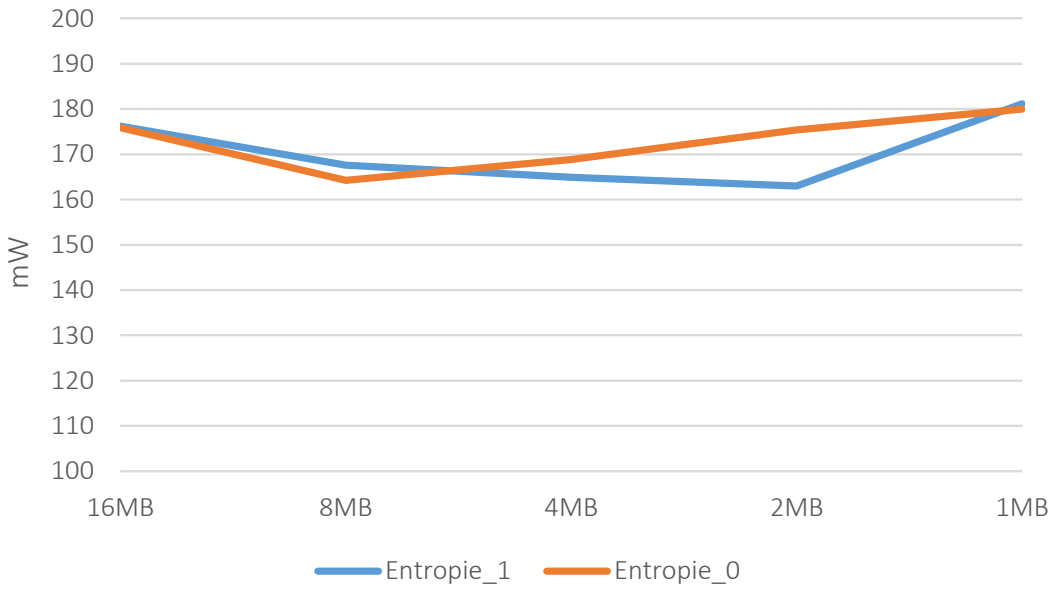

Abbildung 6.13.: Entropieunabhängigkeit

Die Entropie beeinflusst den Energieverbauch beim Versenden nicht, aber die Dateigröße beeinflusst den Energieverbrauch, da der Versendevorgang bei einer größeren Datei länger dauert als bei einer kleineren. Der letzte Faktor der den Energieverbrauch

beim Versenden beeinflussen kann ist, neben der Übertragungsgeschwindigkeit, die Übertragungsart. Der Energieverbrauch der CPU liegt beim Versenden über Wireless-LAN und beim Versenden über Mobiles Internet (3G) konstant bei 50mW-75mW. Die Energieverbrauch der Wireless- und der 3G Komponente hingegen weisen einen deutlichen Unterschied beim Energieverbrauch auf. Die Wireless-Komponente verbraucht 125mW (Abbildung 6.14). Die 3G-Komponente hingegen verbraucht mit 450mW (Abbildung 6.15) deutlich mehr Energie als die Wireless-Komponente.

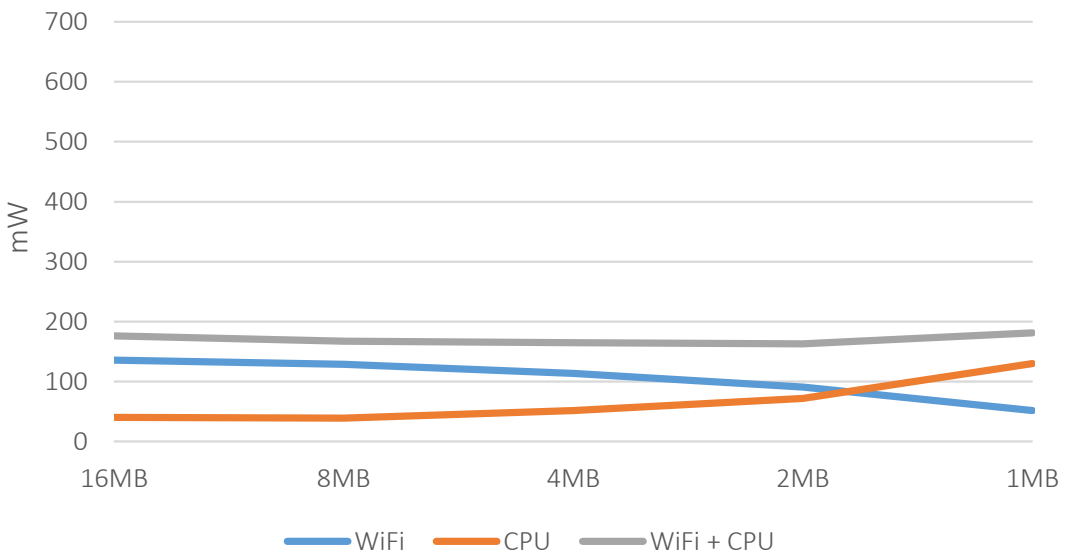

Abbildung 6.14.: CPU + WiFi

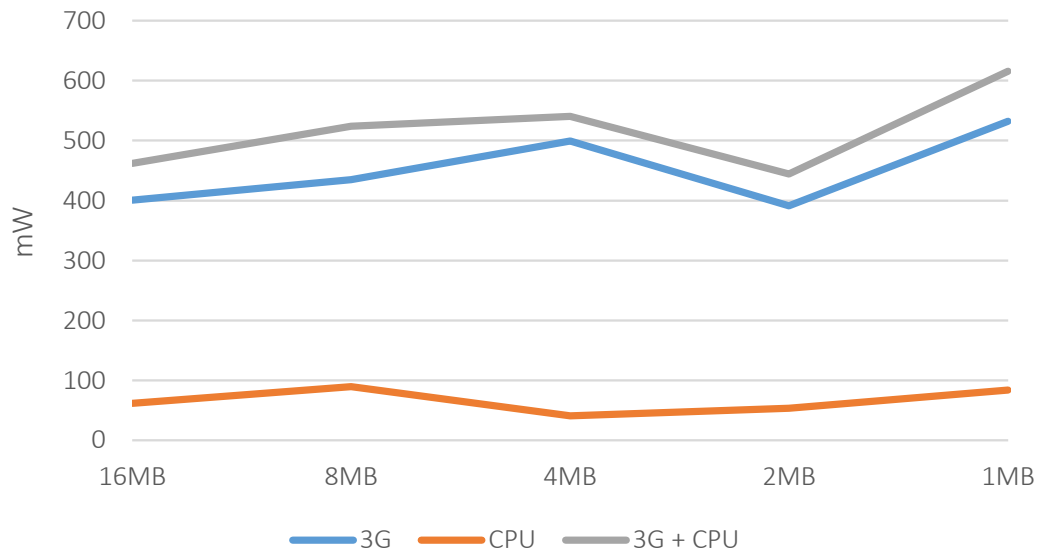

Abbildung 6.15.: CPU + 3G

Der letztendliche Energievebrauch beim Versenden hängt von drei Faktoren ab. Der Dateigröße, der Übertragungsgeschwindigkeit und der Übertragungsart. Die Übertragungsdauer lässt sich durch folgende Formel berechnen:

$$SendDuration(S, F) = F \cdot 1024/S$$

F ist die Dateigröße (in MB) und S die Übertragungsgeschwindigkeit in kb/s. Die Sendedauer ist dann die Zeit in Sekunden, die eine Datei der Größe F bei Übertragungsgeschwindigkeit S benötigt. Nachdem nun die Dauer bekannt ist, die benötigt wird um eine Datei zu verschicken, kann die für das Versenden benötigte Energie mit folgender Formel berechnet werden:

$$Send(F, SendDuration(S, F), T) = F \cdot SendDuration(S, F) \cdot T$$

Die Dateigröße F (in MB) wird mit der Sendedauer und den für das Versenden benötigten Energie je nach Übertragungsart T verrechnet. Für Wireless beträgt der Wert 175mW und für 3G beträgt dieser 500mW (Tabelle 6.1).

Übertragungsart	CPU	Datenmodul	Gesamt
WiFi	50mW	125mW	175mW
3G	50mW	450mW	500mW

Tabelle 6.1.: Energieverbrauch je Übertragungsart

6.5.3. Kombinations-Test

Die Erkenntnisse und die Formeln aus den beiden Tests ergeben in Kombination, dass sich das Komprimieren einer Datei für alle realistischen Fälle immer lohnt. Die Ausnahmefälle sind eine extrem hohe Übertragungsrate oder eine extrem große Ausgangsdatei bei einer niedrigen Übertragungsgeschwindigkeit. Anhand der Ergebnisse aus den Komprimierungstests lässt sich nachfolgende Tabelle aufstellen, die verwendet werden kann, um näherungsweise den Energieverbauch für die Komprimierung einer Datei zu errechnen.

Entropie [Bit/Byte]	Kompressionsrate	Dauer in ms pro MB
1	77%	110
2	62%	180
3	50%	225
4	39%	275
5	29%	310
6	19%	360
7	10%	355
8	0%	285

Tabelle 6.2.: Kompressionsrate und Dauer pro Entropie

Um den Gesamten Energieverbauch für das Komprimieren und das Versenden der komprimierten Datei zu errechnen, wird noch eine weitere Hilfsformel benötigt. Mit dieser Formel lässt sich abhängig von der Entropie der Datei die neue Dateigröße nach dem Komprimierungsvorgang ermitteln.

$$NewFileSize(R, F) = (1 - R) \cdot F$$

Die Dateigröße F wird hierbei mit der Kompressionsrate R der dazugehörigen Entropie verrechnet, welche aus Tabelle 6.2 entnommen werden kann. Die neue Dateigröße wird benötigt, da nun beim Versenden die Komprimierte Datei versendet werden muss und nicht die originale Datei. Die sich ergebende Formel für die Errechnung des Energieverbrauches für den Komprimierungs- und den Sendeprozess lautet:

$$ZipAndSend = Zip(F, t, c) + Send(NewFileSize(R, F), SendDuration(S, NewFileSize(R, F)), T)$$

Eine 16MB große Datei benötigt beim Versand bei einer Übertragungsgeschwindigkeit von 140kb/s knapp 2 Minuten. Bei Energiekosten von 175 mW betragen die Energiekosten somit 20000 mJ für das reine Versenden über Wireless und 60000 mJ für das reine Versenden über 3G (500 mW). Für die schnellste Kompressionsmethode zeigt sich, dass sich die Komprimierung immer lohnt, da die Kosten für die Komprimierung gering sind, und deutlich Zeit beim Versenden der Datei eingespart werden kann. Für eine Entropie von 8 hingegen, lohnt sich das Komprimieren nie, da keine Reduktion der Dateigröße

und somit keine Ersparnisse beim Versenden entstehen. Im Wireless (Abbildung 6.16) sind die Ersparnisse nicht so hoch, wie beim Versand über 3G (Abbildung 6.17).

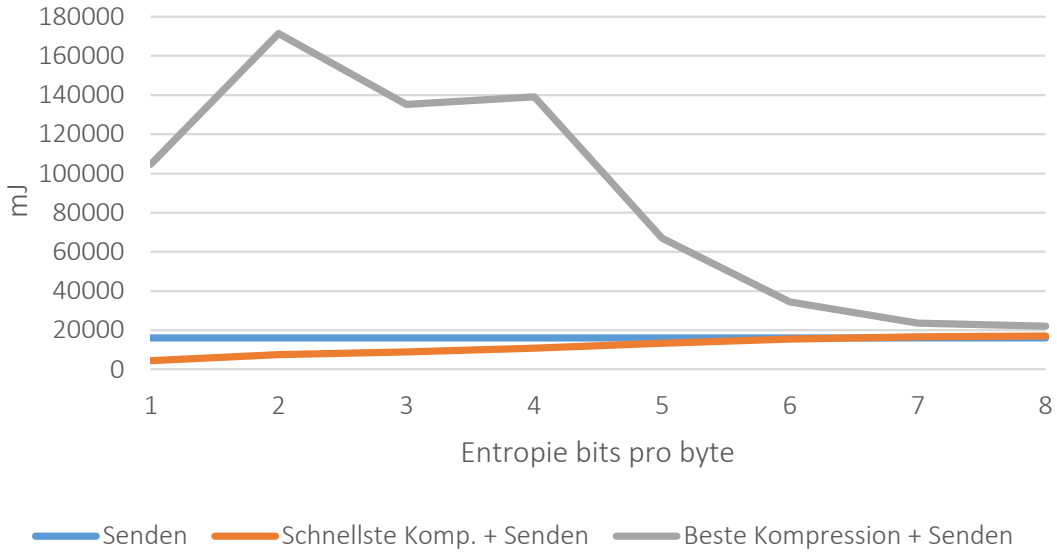

Abbildung 6.16.: Energiekosten über WiFi 16MB

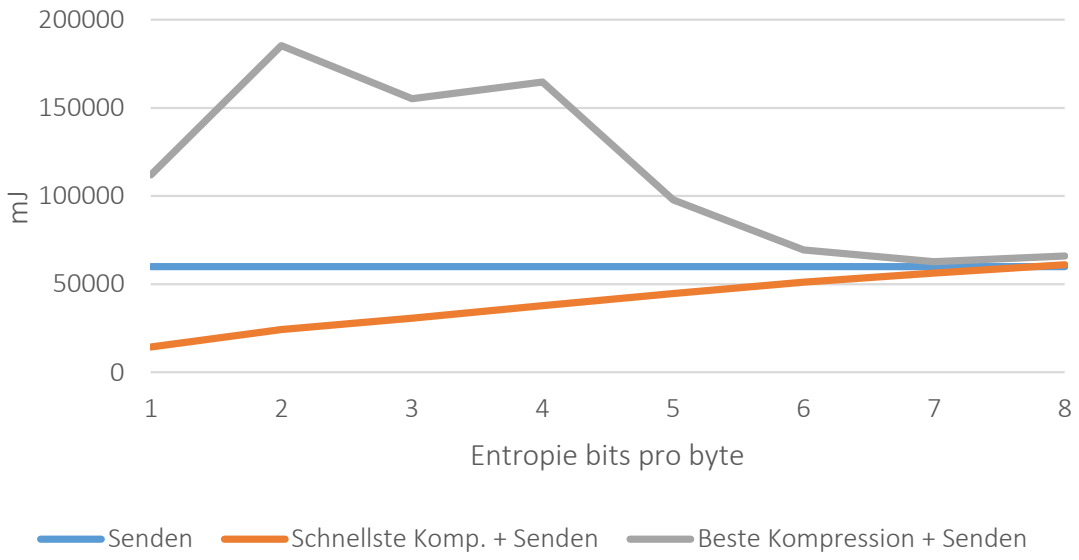

Abbildung 6.17.: Energiekosten über 3G 16MB

6.6. Fazit

Die Auswertung der Tests haben ergeben, dass es sich unter realistischen Bedingungen lohnt, eine Datei vor dem Versenden zu komprimieren. Gerade bei Dateien mit niedrigen Entropien kann durch den Komprimierungsvorgang die Dateigröße deutlich verringert und somit die Kosten für das Versenden reduziert werden. Ein positiver Effekt für den Benutzer ist zusätzlich die kürzere Dauer der Übertragung, sowie die geringeren Kosten für die Nutzung des mobilen Internets. Ist eine Datei extrem klein, lohnt sich eine Komprimierung aufgrund des Mehraufwands nicht. Ebenso wenn eine Datei eine Entropie von 8 Bit pro Byte aufweist. Jedoch sind die zusätzlichen Kosten hier so gering, dass es sich lohnt, sobald eine Datei eine geringere Entropie aufweist. Die beste Kompressionsmethode hingegen lohnt sich nie, da für eine geringfügig bessere Komprimierung viel mehr Energie für den Kompressionsvorgang benötigt wird, die beim Versenden nicht wieder eingespart werden kann.

7. Fazit und Ausblick

In diesem Projektpraktikum wurde ein System, bestehend aus den drei Komponenten Mobile, Server und Klassifizierung mit dem Ziel der Verkehrsflußanalyse, umgesetzt.

Hierzu wurde für die Mobile-Komponente eine Anwendung für Android-Smartphones von Grund auf entwickelt. Diese Mobile-Komponente nimmt Sensordaten auf, komprimiert diese und sendet sie über das Internet an den Server. Für Anwender ist zudem eine Benutzeroberfläche, einfache Steuerungsmöglichkeiten, ein Kartenbildschirm mit dem zurückgelegten Weg und eine Zusammenfassung der Eckdaten der Fahrt vorhanden.

Für die Verarbeitung der Sensordaten wurde ein Server mit Ubuntu Linux und dem Framework Ruby on Rails aufgesetzt. Die Server-Komponente kümmert sich um die Entgegennahme der vom Smartphone gesendeten Sensordaten und hinterlegt sie in der Datenbank. Außerdem wurde eine Web-Oberfläche implementiert, um die gesammelten Sensordaten darstellen zu können.

Aus den gesammelten Sensordaten werden mittels Machine Learning Events klassifiziert. Die Klassifikation mittels Machine Learning liefert gute Ergebnisse, und somit eine präzise Erkennung von Ereignissen. Die Versuche die Klassifizierung zunächst ohne Machine Learning durchzuführen, waren nur für das Klassifizieren von Bremsereignissen aus den GPS-Sensordaten geeignet. Die Accelerometer-Sensordaten ohne Machine Learning zu klassifizieren, führte zu keinem Erfolg.

Abschließend lässt sich zusammenfassen, dass ein robustes System entwickelt wurde und viele der gesetzten Ziele erreicht wurden. Dennoch besteht Verbesserungsbedarf und es bieten sich durchaus viele Erweiterungsmöglichkeiten an.

Die mobile Komponente eignet sich gut, um weitere Daten aufzunehmen, die der Nutzer manuell hinzufügen kann, wie zum Beispiel die Intentionswahl, die derzeit keine Auswirkungen auf die Auswertung der Daten hat. Außerdem sollen die erkannten Ereignisse auf dem Endbildschirm angezeigt werden. Im Allgemeinen können die Daten verwendet werden, um auf die persönlichen Bedürfnisse des Nutzers angepasste Routenvorschläge zu erstellen.

Eine Erweiterung im Bereich des Servers, wurde bereits in Kapitel 5.2.6 erwähnt. Dies wäre die Implementierung einer Pipeline zwischen Server und Klassifikator. Der angesprochene Lösungsansatz basierend auf der Ruby-Java-Bridge, wäre eine Möglichkeit dieses Problem zu lösen.

Im Bereich der Klassifizierung gibt es noch Optimierungspotential. Mittels angepasstem Tiefpass Filter kann das Rauschen noch weiter unterbunden werden, wodurch die Eventerkennung sich verbessern würde. Des Weiteren arbeitet der Klassifikator derzeit auf Daten mit unterschiedlicher Frequenz. Für eine optimale Klassifizierung müssten mehr Trainingsdaten von unterschiedlichen Testgeräten gesammelt werden, um später alle Geräte optimal für eine Klassifikation unterstützen zu können.

A. Anwenderhandbuch

A.1. Die App

Dieses Kapitel beschäftigt sich mit der Installation, Registrierung und Nutzung der mobilen App. Sie wird dazu genutzt, beim Fahrradfahren mit dem Smartphone, Bewegungsdaten zu sammeln und zu speichern. Ist eine Internet Verbidnung vorhanden, können die gesammelten Daten durch Betätigen des *Absenden* Buttons an den Server weiter gegeben. Dort werden die Daten dann ausgewertet.

A.1.1. Systemanforderungen und Installation

Diese Applikation benötigt ein Android Smartphone mit API 10 (v2.3.3), sowie die GPS-Daten des Nutzers. Es muss also vor der Fahrt sichergestellt werden, dass der Dienst aktiviert ist und die Applikation die nötigen Rechte besitzt. OSMDroid benötigt eine Internetverbindung zur Darstellung der Kartendaten.

Download: https://github.com/MobileSensors/MobSens/blob/master/MobileSensingCollector/bin/MobileSensingCollector.apk?raw=true

A.1.2. Anmeldung

Um Daten hochladen zu können, ist eine Anmeldung erforderlich. Ohne eingeloggt zu sein, kann die Aufnahme zwar gestartet werden, die Daten aber nicht zur Auswertung an den Server geschickt werden. Die Login-Daten werden in dem Einstellungsbildschirm eingegeben. Mit diesen Login-Daten kann auch auf das Nutzerprofil im Web zugegriffen werden, siehe A.2

A.1.3. Benutzung und Datenaufnahme

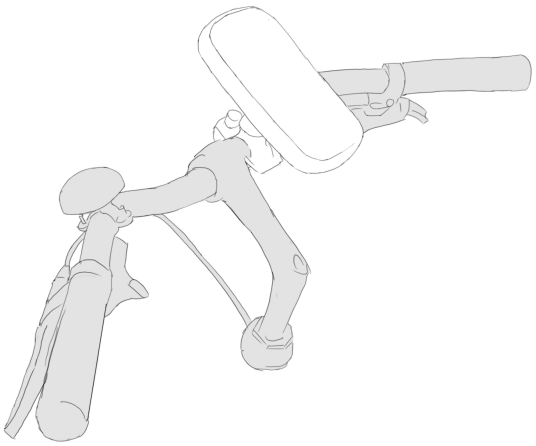

Abbildung A.1.: So wird das Gerät am Lenker befestigt

Das Smartphone muss vor der Fahrt mittig auf dem Lenker des Fahrrades platziert werden (Abbildung A.1). Dabei muss darauf geachtet werden, dass das Smartphone gut fixiert ist und nicht während der Fahrt herunterfallen oder verrutschen kann, da dies sowohl das Smartphone beschädigen, als auch die Daten verfälschen könnte. Eine speziell für den Gerätetyp angefertigte Halterung wird empfohlen, um Komplikationen zu vermeiden.

Vor der Fahrt wird der Nutzer aufgefordert, die Intention zu wählen, also aus welchen Gründen er Fahrrad fährt. Diese Wahl soll in die Bewertung der Strecke mit einfließen, um in der Zukunft differenziertere Bewertungen und Empfehlungen zu geben. Diese Wahl wird zwar bei der Datenübermittlung mitgesendet, fließt in der momentanen Version allerdings nicht in die Auswertung mit ein. Es stehen drei Intentionen zur Auswahl. *Sport* bezeichnet eine körperlich anspruchsvolle Fahrt, *Freizeit* eine Strecke die auf Erholung zielt, und *Alltag* eine Nutzfahrt, z.B. für Einkäufe. Es ist möglich eine Intention zu markieren, um eine nähere Beschreibung zu erhalten. Wenn die Wahl getroffen wurde, wird sie durch Drücken des Start-Knopfes bestätigt.

Abbildung A.2.: Ansicht während der Fahrt

Sobald die Aufnahme gestartet wurde, erscheint der sogenannte Kartenbildschirm auf dem Display. Er zeigt eine OpenStreetMap-Karte der Umgebung. Die Karte richtet sich automatisch in die Fahrtrichtung aus, und muss nicht während der Fahrt neu ausgerichtet oder positioniert werden. Durch Antippen der Karte erscheinen die Kartenzoomtasten von OpenStreetMap am unteren Kartenrand. Mit diesen Tasten kann der Zoomfaktor der Karte angepasst werden. Während der Fahrt wird eine blaue Linie gezeichnet. Diese Linie zeigt die bereits zurückgelegte Strecke an. Eine blaue Raute zeigt die momentane Position an. Am unteren Rand der Karte wird die aktuelle Geschwindigkeit in Kilometer pro Stunde und die insgesamt zurückgelegte Strecke in Metern angezeigt. Am unteren Bildschirmrand befindet sich der Stop-Knopf. Sobald dieser Knopf betätigt wird, ist die Aufnahme beendet.

A.2. Bedienung der Web-Oberfläche

Die Web-Oberfläche dient dazu, die mit der Smartphone-Applikation gesammelten Daten anzuschauen und zu einem gewissen Grad bearbeiten zu können. Bevor das jedoch möglich ist, ist es notwendig ein Benutzerkonto anzulegen, um sich anschließend einloggen zu können. Ein Link zur Anmeldung ist auf der Startseite unter *Sign up* gegeben, der den Benutzer zum Anmeldebildschirm weiterleitet, wo er sich mit einer gültigen E-Mail-Adresse und einem Passwort anmelden kann. War die Anmeldung erfolgreich, wird

man anschließend auf die Seite mit einer Übersicht seiner eigenen Aufnahmen weitergeleitet. Hier werden alle Aufnahmen, die der Nutzer mit der Applikation gesammelt und abgeschickt hat, aufgelistet. Für jede Aufnahme besteht die Möglichkeit sich die gesammelten Daten anzusehen, den Titel der Aufnahme zu ändern oder die Aufnahme gar zu löschen. Als Administrator kann sich unter dem Unterpunkt *Recordings*, welcher unter dem Punkt Admin in der Titelleiste verfügbar ist, eine Liste der Aufnahmen aller Nutzer angezeigt werden lassen, wie in Abbildung A.3 zu sehen ist.

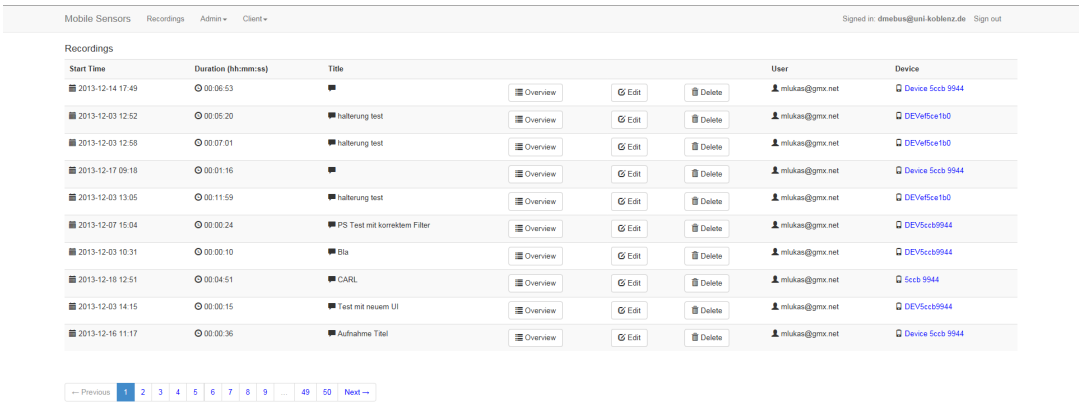

Abbildung A.3.: Weboberfläche Admin Recordings

Hat man sich für eine Aufnahme entschieden, gelangt man durch die Schaltfläche Overview zu einer Übersicht der ausgewählten Aufnahmen. In Abbildung A.4 und A.5 ist die Seite einer Aufnahme zu sehen.

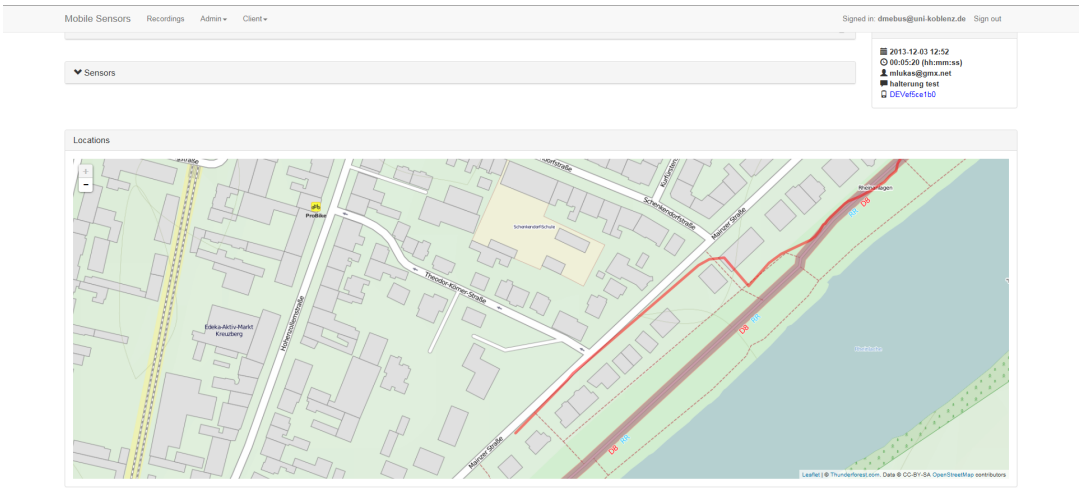

Abbildung A.4.: Übersichtsseite eines Recordings

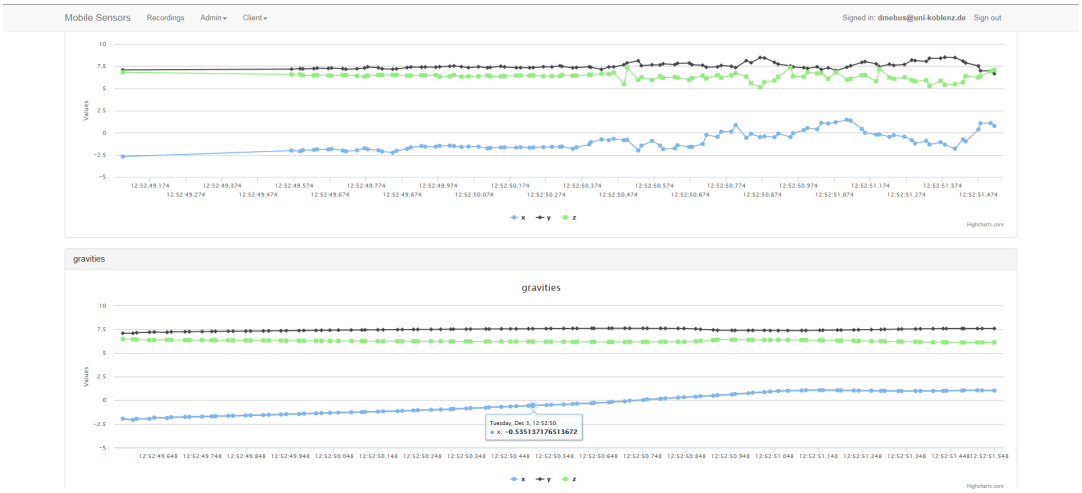

Abbildung A.5.: Übersichtsseite eines Recordings

A.3. Installation Web-Server

Für das Aufsetzen des Web-Servers ist es notwendig, die in Abschnitt 5.2.1 genannten Softwarepakete zu installieren. Dies geschieht in Debian-verwandten Betriebssystemen wie Ubuntu über das Programm aptitude beziehungsweise das Programm apt-get. Die folgende Anleitung dient als grober Leitfaden und erhebt keinen Anspruch auf Vollständigkeit. Jede Konfiguration eines Betriebssystems ist unterschiedlich und es ergeben sich dadurch bedingt oft Unterschiede bei den einzelnen Schritten. Dennoch sind die zu erfüllenden Aufgaben einer Installation im Folgenden konkret aufgelistet, um ein grobes Bild eines Installationsvorgangs zu vermitteln:

Stoppen des Apache Web-Servers:

```
$ service apache2 stop
```

Auschecken der Server-Software:

```
$ cd /var/www
$ git clone https://github.com/MobileSensors/MobSens.git
```

Link anlegen und Rechte vergeben:

```
$ ln -s MobSens/MobileSensingServer ./mobilesensors
$ chown -R www-data .
```

Die default.conf des Apache Web-Servers:

```
<VirtualHost _default_:80>
    # ServerName www.yourhost.com # Commented out for default
    DocumentRoot /var/www/mobilesensors/public
    <Directory /var/www/mobilesensors/public>
        AllowOverride all
```

```
        Options -MultiViews
    </Directory>
</VirtualHost>
```

Datenbank konfigurieren:

```
$ vim /var/www/mobilesensors/config/database.yml
```

Installation der Rubygems:

```
$ cd /var/www/mobilesensors
$ bundle install
```

Durchführung der Datenbank-Migration, Kompilierung der Assets und Anlegen der Routen:

```
$ cd /var/www/mobilesensors
$ RAILS_ENV=production rake db:migrate
$ RAILS_ENV=production rake assets:precompile
$ RAILS_ENV=production rake routes
```

Starten des Delayed Job Daemons und des Apache Web-Servers:

```
$ RAILS_ENV=production rails runner bin/delayed_job start
$ service apache2 start
```

Wenn alle Schritte fehlerfrei durchgeführt sind, sollte die Server Software nun auf dem Host laufen.

Literatur

[ABH12] Shahid Ayub, Alireza Bahraminisaab und Bahram Honary. "A Sensor Fusion Method for Smart phone Orientation Estimation". In: *Proceedings of the 13th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting, Liverpool*. 2012.

[Bra14] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. Internet Engineering Task Force (IETF), 2014. URL: http://tools.ietf.org/html/rfc7159.

[Che14] Eric Cheever. *Representation of Numbers*. Mai 2014. URL: http://www.swarthmore.edu/NatSci/echeeve1/Ref/BinaryMath/NumSys.html.

[Eri+08] Jakob Eriksson u. a. "The pothole patrol: using a mobile sensor network for road surface monitoring." In: *MobiSys*. Hrsg. von Dirk Grunwald u. a. ACM, 19. Juni 2008, S. 29–39. ISBN: 978-1-60558-139-2. URL: http://dblp.uni-trier.de/db/conf/mobisys/mobisys2008.html#ErikssonGHNMB08.

[Fie+99] R. Fielding u. a. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Network Working Group, 1999. URL: http://tools.ietf.org/html/rfc2616.

[KS10] F. Klinker und G. Skoruppa. *Ein Optimiertes Glaettungsverfahren motiviert durch eine technische Fragestellung*. PDF. Fakultaet fuer Mathematik, TU Dortmund, 2010. URL: http://www.mathematik.tu-dortmund.de/~klinker/Paper/Klinker,Skoruppa-Glaettung-%28pre%29.pdf.

[LL13] O.D. Lara und M.A. Labrador. "A Survey on Human Activity Recognition using Wearable Sensors". In: *Communications Surveys Tutorials, IEEE* 15.3 (2013), S. 1192–1209. ISSN: 1553-877X. DOI: 10.1109/SURV.2012.110112.00192.

[LZ11] Zhiyun Qian ZhaoguangWang Robert P. Dick Z. Morley Mao Lei Yang Lide Zhang Birjodh Tiwana. *Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones*. PDF. EECS Department, University of Michigan Ann Arbor, MI, USA, 2011. URL: http://robertdick.org/publications/zhang10oct.pdf.

[Med+11] Artis Mednis u. a. "Real time pothole detection using Android smartphones with accelerometers." In: *DCOSS*. IEEE, 2011, S. 1–6. ISBN: 978-1-4577-0512-0. URL: http://dblp.uni-trier.de/db/conf/dcoss/dcoss2011.html#MednisSZKS11.

[Mit97] Thomas M. Mitchell. *Machine Learning*. 1. Aufl. New York, NY, USA: McGraw-Hill, Inc., 1997. ISBN: 0070428077, 9780070428072.

[Par] In: ().

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 1558602402.

[Sha05] Y. Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180. Network Working Group, 2005. URL: `http://tools.ietf.org/html/rfc4180`.

[Smi02] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. Newnes, 2002. ISBN: 0-7506-7444-X.

[V.+12] Astarita V. u. a. "A mobile application for road surface quality control: UNIquALroad". In: *Procedia-Social and Behavioral Journal* 54 (2012).