FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

Fahreridentifikation mittels Machine Learning

Driver identification using Machine Learning

Masterarbeit

Zur Erlangung des akademischen Grades

Master of Science in Engineering

der Fachhochschule Campus Wien
Masterstudiengang: ITS-20

Vorgelegt von:
David Lechner

Personenkennzeichen:
1810537012

ErstbetreuerIn / ErstbegutachterIn:
Dr. Martin Schmiedecker

ZweitbetreuerIn / ZweitbegutachterIn:
Kevin Koch

Eingereicht am:
13.05.2020

Erklärung:

Ich erkläre, dass die vorliegende Masterarbeit von mir selbst verfasst wurde und ich keine anderen als die angeführten Behelfe verwendet bzw. mich auch sonst keiner unerlaubter Hilfe bedient habe.

Ich versichere, dass ich diese Masterarbeit bisher weder im In- noch im Ausland (einer Beurteilerin/einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Weiters versichere ich, dass die von mir eingereichten Exemplare (ausgedruckt und elektronisch) identisch sind.

Datum: Unterschrift:

Kurzfassung

Nicht nur durch die neuen Technologien Autonomes Fahren und *Vehicle-to-Everything* (V2X) erzeugen Fahrzeuge Unmengen an Daten. Auch schon heute tauschen Steuergeräte über den CAN-Bus Messwerte und Befehle aus, welche für Kontroll- und Sicherheitsfunktionen genutzt werden. Typische Daten sind beispielsweise Lenkradwinkel, Bremsdruck, Motordrehzahl und Fahrpedalrohwert. Aus diesen Daten ist es möglich, viele Informationen zu gewinnen. Eine davon ist, den Lenker eines Fahrzeuges während der Fahrt nur durch das Fahrverhalten zu identifizieren. Sie basiert darauf, dass jede Person ein individuelles Fahrverhalten im Straßenverkehr hat, das sich in den Fahrzeugdaten widerspiegelt. Die vorliegende Masterarbeit stellt ein System vor, das diese Aufgabenstellung adressiert. Ein erstes Ziel ist das Validieren der bereits bestehenden Methoden mit den vorhandenen Testdaten. Dabei wird der *Machine Learning* (ML) Algorithmus *Random Forest* zur Klassifizierung von CAN-Nachrichten eingesetzt. Hier konnte ein Ergebnis von 85% erzielt werden. Das bedeutet, dass von 180 Datenpunkten pro Fahrer 150 korrekt mit dem Algorithmus zugeordnet wurden. Ein weiteres Ziel ist, das ML-Modell durch geschickte Parametrisierung zu optimieren, was jedoch nur ein Prozent eingebracht hat. Im zweiten Teil der Arbeit wird dargelegt, wie das System in ein Fahrzeug integriert werden kann. Warum eine Architektur ohne *Cloud* für diese Anwendung und hinsichtlich Privatsphäre besser ist und was unter *Edge Computing* zu verstehen ist, wird ebenfalls erläutert. Die verschiedenen Softwarekomponenten geben einen Einblick, wie das System auf einem *Embedded-Device* mit limitierten Ressourcen funktioniert. Des Weiteren werden mehrere Entscheidungsansätze besprochen, wann ein Fahrer als ausreichend identifiziert gilt. Im Zuge dessen wird auch evaluiert, wie lange dieser Prozess dauert. Aufbauend darauf werden Anwendungen diskutiert, die für das System infrage kommen. Vor allem eignen sich jene, welche die Information über die aktuell fahrende Person erst gegen Ende der Fahrt benötigen. Als Beispiel wird ein digitales Fahrtenbuch angeführt, das vollkommen ohne Interaktion des Benutzers auskommt.

Abstract

Vehicles generate an enormous amount of data, not only due to the new technologies autonomous driving and Vehicle-to-Everything. It is common practice for the control units to exchange measured values and commands via the CAN-Bus, which are used for control as well as safety functions. Typical values include break pressure, steering wheel angel and engine speed. It is possible to gain a lot of information from these values. For example can a driver be identified only based on their driving behaviour. The idea behind this is that individuals have their own unique characteristics while handling different traffic situations. This master thesis introduces a system that aims at mastering this task. The first goal is to validate already existing methods with the test data at hand. In order to do that the machine learning (ML) algorithm Random Forest is used. It is able to classify CAN-messages and link them to the driver. The results showed a precision of 85% that represents 150 out of 180 correctly assigned data points. Another goal of the thesis is to optimize the machine learning model with clever parameterization. However, this only resulted in a small improvement of 1 percent. The second part of the thesis deals with the integration of the system into a vehicle. Furthermore, it is explained why an architecture without cloud interaction is better with respect to privacy and what the term Edge Computing means in this context. The following section outlines the used software components and shows how the whole system works on an embedded device with limited resources. The subsequent part analyses several approaches that determine a driver with the given ML model outcomes. In the course of that the duration of the decision process is evaluated. Based on the results and insights, eligible applications are discussed. Especially those applications where the driver does not have to be identified until the end of the trip are suitable. One implementation of this is a digital driver logbook, that works without any human interaction.

Abkürzungsverzeichnis

AI	Artificial Intelligence
API	Application Programming Interface
ASAM	Association for Standardization of Automation and Measuring Systems
ASIC	Application-Specific Integrated Circuit
BSP	Board Support Package
CAN	Controller Area Network
CCU	Car Connectivity Unit
CPU	Central Processing Unit
CV	Cross Validation
CRC	Cyclic Redundancy Check
DLC	Data Length Code
ECU	Electronic Control Units
EMV	Elektromagnetische Verträglichkeit
FPGA	Field Progammable Gate Array
GPU	Graphics Processing Unit
HDF5	Hierarchical Data Format Version 5
IaaS	Infrastructure-as-a-Service
MLaaS	Machine Learning-as-a-service
MDF	Measurement Data Format
ML	Machine Learning
OBD	On-Board-Diagnose
OEM	Original Equipment Manufacturer
PoC	Proove of Concept
PKI	Public Key Infrastructure
REST	Representational State Transfer
RFE	Recursive Feature Elimination
SaaS	Software-as-a-Service
TPU	Tensor Processing Unit

Schlüsselbegriffe

CAN-Bus
Fahreridentifikation
Machine Learning

Inhaltsverzeichnis

1. Einleitung

In der heutigen Zeit werden Unmengen an Daten von verschiedensten Geräten generiert und versendet. Den Anfang haben *Smartphones*, dann *Wearables* gemacht. Mit dem Aufkommen des Bereichs Internet of Things (IoT) kann jegliches technische Gerät – von der Lampe bis hin zur Fertigungsmaschine – mit dem Internet verbunden sein und Status- beziehungsweise Messdaten übermitteln. Dieser Trend macht auch vor Fahrzeugen keinen Halt. Schon längst sind moderne Autos mit LTE-, GPS und WLAN-Modulen ausgestattet und senden Daten unter anderem zum Hersteller. Gartner prognostiziert für das Jahr 2020 470 Millionen vernetzte Fahrzeuge [Gar19]. In Zukunft werden wahrscheinlich alle Fahrzeuge mit etlichen Sensoren ausgestattet sein und kommunizieren untereinander, mit der Umwelt, dem Fahrer oder sonstigen Service-Anbietern. Dies gründet vor allem auf den wachsenden Themen *Vehicle-to-Everything* (V2X) und Autonomem Fahren. Insbesondere beim Letztgenannten macht die generierte Datengröße noch einen großen Sprung, da neben Sensoren auch Radar und Videokameras zur Umwelterkennung hinzukommen. Jedoch versenden schon heute *Electronic Control Units* (ECUs) Daten im Auto, wie zum Beispiel Lenkradwinkel, Gangposition und Bremsdruck, welche für Kontroll-, Sicherheits- und Komfortfunktionen genutzt werden. Aus diesen Daten können viele Informationen gewonnen werden. Eine davon ist, den Lenker eines Autos während der Fahrt nur durch das Fahrverhalten zu identifizieren.

Daraus lassen sich weitere unterschiedliche Anwendungen ableiten. Einige von ihnen schaffen Komfort und erleichtern in gewisser Weise das Leben des Fahrers. Andere indes könnten gegen den Fahrzeughalter und die Fahrerin selbst eingesetzt werden, diese gehen mit datenschutzrechtlichen Bedenken einher.

Doch zunächst zu den Anwendungen, welche eine positive Auswirkung haben können. Moderne Autos – vor allem jene mit einem Automatik Getriebe – bieten die Möglichkeit, sich an den Fahrstil anzupassen. Wenn beispielsweise eine Person zum schnelleren Beschleunigen neigt, lernt dies das Auto und schaltet demnach erst bei einer höheren Motordrehzahl in den nächsthöheren Gang. Dasselbe gilt bei einem gemächlichen Fahrstil, wobei hier eher früher geschalten wird. Lernt das Auto nun von einer Person mit dem zweitgenannten Stil und wird aber auch hin und wieder mit anderen Personen, zum Beispiel Familienmitgliedern, geteilt, kann es für diese einen Komfortverlust darstellen. Identifiziert das Auto jedoch durch das Fahrverhalten eine andere Person, könnte es den gelernten Stil temporär vergessen oder gar ein neues Profil anlegen und erneut lernen.

Im Unternehmensbereich, wo Dienstfahrzeuge oder Lieferwägen zum Einsatz kommen, ist es meistens notwendig ein Fahrtenbuch zu führen. Bei einem klassischen Fahrtenbuch werden die gefahrenen Kilometer, Datum und Uhrzeit, Name des Fahrers, Abfahrts- und Zielort sowie die Unterschrift in einem Buch im Fahrzeug festgehalten. Durch das händische Eintragen kann es zu Zeitverlusten, unvollständige Dokumentation und auch manipulierten Daten kommen, was im Endeffekt in hohen Kosten resultiert. Elektronische Fahrtenbücher sind in der Lage diese Daten digital aufzuzeichnen. Mithilfe eines *On-Board-Diagnose II* (OBDII) Dongles werden der Kilometerstand und die Fahrzeugnummer erfasst. Die GPS-Position, Datum und Uhrzeit, wie auch die Fahreridentifikation und gegebenenfalls der Zweck der Fahrt

werden mit einem gekoppelten Smartphone in das System ergänzt [1] [2]. Die Abrechnung und Auswertung lässt sich dadurch erheblich erleichtern, jedoch ist noch immer eine Interaktion des Lenkers erforderlich, was wiederum Raum für Fehler und Manipulation schafft. Kommt eine Fahreridentifikation basierend auf dem Fahrverhalten zum Einsatz, kann das komplette System in das Fahrzeug integriert werden. Die Positionsdaten können dabei von einem integrierten Navigationssystem ausgelesen werden und die restlichen Fahrzeugdaten über den *Controller Area Network* (CAN, siehe Kapitel 2.2) Bus. Die Fehlerwahrscheinlichkeit verringert sich dabei enorm und die Benutzerfreundlichkeit wird erhöht, da nichts mehr händisch eingetragen werden muss.

Überdies ist auch eine Art Diebstahlwarnung zu realisieren. Dem System sind eine Reihe an Fahrerprofilen bekannt, welche zuvor eingelernt wurden. Bei jeder neuen Fahrt wird überprüft, ob das momentane Fahrverhalten des Lenkers erkannt wird. Stellt das System zum Beispiel zu einer Wahrscheinlichkeit von 90% fest, dass sich das Profil nicht unten den bekannten befindet, kann beispielsweise eine Benachrichtigung an den Fahrzeughalter versendet, oder die Fahrerin zum Anhalten gebracht werden.

Der Mechanismus kann weiters dazu verwendet werden, Fahrzeug-Funktionen und Leistung fahrerabhängig zu steuern. Ein Familienvater ist so etwa in der Lage, die zur Verfügung stehende Leistung einzuschränken, wenn seine Kinder mit dem Auto fahren.

Durch das Aufzeichnen und Analysieren von personenbezogenen Daten kommen natürlich auch datenschutzrechtliche Bedenken auf. Werden die Daten in eine *Cloud* – sei es eine vom Hersteller, der Versicherung oder einer anderen Drittpartei – gesendet und ausgewertet, können Personen von diesen Unternehmen oder Organisationen eindeutig identifiziert werden. Liegen zudem Standortdaten des Autos vor, kann auch eine genaue Ortung durchgeführt werden. Dies kann in einigen Fällen problematisch sein. So könnte eventuell der Hersteller bestimmte Services anbieten, welche personalisierte Werbungen während dem Fahren anzeigen. Zum Beispiel ist es dadurch möglich, bevorzugte Restaurants in der Navigationsansicht hervorzuheben. Auch Versicherungen können diese Informationen für sich nutzen, um Fahrer- und Fahrstil abhängige Versicherungspakete anbieten zu können [3]. Hier wird genauso ein OBDII-Dongle verbunden mit einer App für die Fahrstilanalyse eingesetzt. Werden viele Notbremsungen und rasche Beschleunigen verzeichnet, kann die Versicherungsprämie erhöht werden. Der Dongle und das Smartphone könnten durch das System ersetzt werden.

Die angeführten Beispiele zeigen, dass ein System, welches die Person hinter dem Lenkrad eines Fahrzeuges eindeutig identifiziert, Benutzervorteile bringen kann. Zudem erschließt sich ein neues Geschäftsfeld für Autohersteller, um vielleicht Premium-Features anbieten zu können. Jedoch stellen sich auch Fragen zur Privatsphäre und wie mit solch sensiblen Daten umgegangen wird.

1.1. Überblick und Ziele

In der Masterarbeit wird ein System vorgestellt, welches den Lenker eines Fahrzeuges anhand des Fahrstils eindeutig identifizieren kann. Ausgangspunkt ist, dass jedes Individuum ein anderes Verhalten in verschiedenen Verkehrssituationen hat. Einer bremst eher früher an, dafür nicht so kräftig, währendessen eine später und härter bremst. Person A fährt eine Kurve mit 30 km/h im dritten Gang und lenkt dabei etwas weniger ein. Person B andererseits nimmt eine Kurve schneller und muss daher mehr einlenken. Dafür muss vielleicht

[1] Drivebox: `https://www.drivebox.at/drivebox_main.html`
[2] Vimcar: `https://vimcar.de/fahrtenbuch`
[3] Signal Iduna: `https://www.app-drive.de`

die Lenkradposition in der Kurve nicht mehr angepasst werden, wohingegen vielleicht einmal kurz die Bremse gedrückt wird. All diese Informationen - Geschwindigkeit, Drehmoment, Lenkradwinkel, Bremsdruck usw. - werden über den CAN Bus als Nachrichten ausgetauscht. *Electronic-Control-Units* (ECUs) verarbeiten diese und steuern dementsprechend den Motor, das Getriebe oder eine Öldruckpumpe an.

Um aus verschiedenen CAN-Daten einen Zusammenhang zur Lenkerin herstellen zu können, wird eine Untermenge von *Artificial Intelligence* (AI), nämlich *Machine Learning* (ML) verwendet. Bei ML kommen mehrere mathematische Algorithmen zum Einsatz, welche statistische Modelle aufgrund von Trainingsdaten aufbauen. Die Modelle versuchen daraufhin ein Muster in den Daten zu erkennen, um später eine Aussage über den Lenker treffen zu können. Die Lerndaten bestehen dabei aus den verschiedenen CAN-Nachrichten und einer Fahreridentifikation - dem Zielwert. Nach der Trainingsphase des Modells werden nur noch die CAN-Daten eingespielt. Als Resultat wird die Fahrer-ID mit größter zutreffender Wahrscheinlichkeit ausgegeben.[Con12]

Das Ziel der Masterarbeit ist, bereits existierende Methoden zur Fahreridentifizierung mit *Machine Learning* (siehe 1.3) und den vorliegenden Daten (siehe 3.1) zu validieren, beziehungsweise zu optimieren. Im Zuge dessen soll herausgefunden werden, wie lange die Trainingsphase eines ML-Modells sein muss, um eine Trefferquote von über 85% erzielen zu können. Des Weiteren gilt es die Dauer zu bestimmen, welche für die Identifikation (zu 85%) einer Fahrzeuglenkerin mit einem bereits trainierten Modell benötigt wird. Ein weiteres Ziel ist das System in ein Fahrzeug zu integrieren und mit (fast) Echtzeit-Daten zu erproben. Hierfür wird ein *Embedded-Device* mit beschränkten Ressourcen eingesetzt. Folgende Liste fasst die Ziele noch einmal zusammen:

- Sind die existierenden Methoden mit vorhandenen Daten valide?
- Können die existierenden Methoden optimiert werden?
- Wie lange dauert die Trainings- und Identifikationphase für eine Identifikationsrate von min. 85%?
- Kann das System in ein Fahrzeug mit einem *Embedded-Device* integriert werden?

1.2. Struktur der Arbeit

Diese Arbeit geht eingangs auf die Motivation und Problemstellung ein. Danach wird ein Überblick mit den Zielen geschaffen, sowie vergleichbare Arbeiten vorgestellt. Im zweiten Kapitel werden die Grundlagen behandelt. Das beinhaltet *Machine Learning*, den CAN-Bus und *Edge Computing*. Kapitel 3 beschreibt den Versuchsaufbau und die Umsetzung des Systems. Dabei wird auf die vorliegenden Daten eingegangen und wie diese bestmöglich vorbereitet werden. Des Weiteren wird auch die Implementierung eines ML-Algorithmus beschrieben und wie damit der Lenker eins Fahrzeuges identifiziert werden kann. Der nächste Abschnitt zeigt die Optimierung des ML-Modells und der Daten. Kapitel 5 widmet sich der Integration in ein Fahrzeug und beschreibt unter anderem die Architektur für eine Realisierung. Darauf werden die erzielten Ergebnisse präsentiert sowie diskutiert. Schlussendlich lässt die Zusammenfassung die Arbeit noch einmal Revue passieren.

1.3. Verwandte Arbeiten

Zu diesem Thema sind bereits einige Papers zu finden. Zum Beispiel konnten 2005 Forscher aus Japan eine Fahreridentifizierung mit einer Genauigkeit von 73% durchführen.[4] Sie verwendeten jedoch CAN-Bus Nachrichten von einem Simulator. Später wurde die Trefferquote im Labor zwar auf 89.6% erhöht, aber die Anwendung unter realen Bedingungen brachte nur 71% ein.[5] Im Folgenden werden noch drei Papers vorgestellt, welche die Identifikation ausschließlich mit echten Fahrzeugdaten untersucht haben.

In einem Paper von 2016 [ETKK16] wurde untersucht, ob Einzelpersonen basierend auf ihren natürlichen Fahrverhalten identifiziert werden können. Für die Datenbasis wurden CAN-Nachrichten eines Serienfahrzeuges verwendet. 15 Teilnehmerinnen mussten zuerst bestimmte Manöver auf einem abgesperrten Parkplatz durchführen und danach eine ca. 80 km lange vordefinierte Strecke abfahren. Für die Analyse wurde *Machine Learning* mit verschiedenen Algorithmen verwendet. Dabei konnte festgestellt werden, dass bei einem 1 zu 1 Vergleich die Teilnehmer zu 100% unterscheidbar sind. Des Weiteren konnte eine hohe Identifikationswahrscheinlichkeit bei nur acht Minuten Trainingszeit erzielt werden.

Die Arbeit von B. Gahr et al. von 2018 [GRDW18] setzt auf die soeben beschriebene auf. Da gezeigt wurde, dass eine Identifikation zu 100% möglich ist, hat diese es versucht, die Methoden mit anderen Daten zu validieren. Dafür wurden aber nicht direkt die Nachrichten vom CAN-Bus abgegriffen, sondern über ein Smartphone, welches über Bluetooth mit einem OBDII-Dongle verbunden wurde. Hinzu sind noch andere Sensordaten gekommen. Dadurch konnte mit den Methoden jedoch nur eine Identifikationsrate von maximal 70% erzielt werden. Daher wurde ein anderer Ansatz gewählt, bei dem nur Daten während eines Bremsvorgangs in Betracht gezogen werden. Das hat zu Ergebnissen zwischen 80 und 99,5% geführt.

Ein anderes Paper [HSS+16] verfolgte einen ähnlichen Ansatz, bei dem nur die Daten während einer Kurve berücksichtigt werden. Die CAN-Nachrichten wurden hierbei mit einem proprietären *Data-Logger* aufgezeichnet. Es folgte eine Analyse der 12 häufigsten Kurven im Datensatz. Dabei konnte ein Fahrer verglichen mit einem zweiten Fahrer mit einer Wahrscheinlichkeit von fast 77% unterschieden werden. Besteht das Set aus fünf Fahrern, liegt die Identifikationsrate bei 50,1%.

1.4. Neuigkeitswert

Die Masterarbeit wird teilweise auf die bereits bestehenden Papers aufsetzten und bewehrte Methoden übernehmen. Ein wesentlicher Unterschied ist aber, dass die hier vorliegenden Daten (siehe 3.1) weder in einem bestimmten Setting noch unter anderen Kontrolleinflüssen und direkt vom CAN-Bus mitgemessen worden sind. Wie auch aus den Zielen hervorgeht, wird versucht, die Methoden dahingehend zu verbessern, sodass eine möglichst schnelle Identifikation durchgeführt werden kann.

[4] [WOM+05]
[5] [MNO+07]

2. Grundlagen

In diesem Kapitel werden die Grundlagen für das zu realisierende System beschrieben. Es beinhaltet eine Einführung in *Machine Learning*[1] und gibt dabei einen Überblick auf die verschiedenen Methoden und Anwendungsfälle. Speziell werden Algorithmen beschrieben, welche laut Literatur[2] besonders interessant für dieses Thema sind. Zudem wird der *Controller Area Network* Bus vorgestellt, welcher für die Datensammlung im Kontext der Masterarbeit essenziell ist.

2.1. Machine Learning

Machine Learning ist ein Bereich der Computer Wissenschaften und beschäftigt sich mit Algorithmen und Techniken zur Lösung von komplexen Problemen, wo konventionelle Programmiermethoden nicht vielversprechend sind. Wir werden heutzutage im alltäglichen Leben mit vielen Anwendungen konfrontiert, die ohne ML nicht möglich wären. Beispiele dafür sind Sprachsteuerung, Bild-, Gesichts- und Handschrifterkennung, Stauvorhersage aber auch Autonomes Fahren und die Erkennung von Brustkrebs [KEE+15]. Schon vor 1980 wurde versucht, einige solcher komplexen Probleme zu lösen, was jedoch nicht immer von Erfolg gekrönt war. Erst Mitte der 2000er Jahre ist der Fortschritt in dem Bereich drastisch gestiegen. Gründe dafür gibt es mehrere. Zum einen ist mit dem Aufkommen des Internets eine viel größere Anzahl an Daten vorhanden und zum anderen sind Rechenleistung beziehungsweise Speicherplatz billiger, leichter zugänglich und vor allem performanter. Des Weiteren sind die Algorithmen verbessert und angepasst worden.

Oft werden *Machine Learning* und Künstliche Intelligenz (engl. *Artificial Intelligence*, kurz AI) mit einander gleichgesetzt, was jedoch nicht stimmt. AI ist ein viel breitgefächerter Forschungsbereich, der mittels mehrerer Ansätzen versucht, Maschinen das "Denken"beizubringen. ML hingegen ist ein möglicher Ansatz dafür.

Um das Vorgehen zum Lösen eines komplexen Problems mittels *Machine Learning* besser verstehen zu können, wird es im Folgenden anhand einer Anwendung, welche handgeschriebene Buchstaben erkennt, erklärt. Zu aller erst wird eine Vielzahl an verschiedenen Datenpunkten (engl. *data points*) - Bilder mit handgeschriebenen Buchstaben - benötigt. Zusätzlich müssen diese mit den enthaltenen Buchstaben markiert (engl. *labelled*) sein. Das Ziel ist, dass die Anwendung nicht nur die Buchstaben im Datenset erkennt, sondern auch jene, die nicht enthalten sind. Mit der konventionellen Methode wird anfangs versucht zu verstehen, wie die Bilder mit den Buchstaben zusammenhängen. Danach wird eine Reihe an Regeln festgelegt, um auch neue Bilder erkennen zu können. Da es jedoch eine große Variation von Handschriften gibt, kann das Regelset sehr schnell sehr groß werden. *Machine Learning* Algorithmen gehen hierbei einen generelleren Lösungsweg, indem sie direkt von den markierten Daten lernen und sich das Regelset (engl. *model*) selbst aneignen. Je mehr Bilder von Buchstaben vorhanden sind, desto genauer wird das ML-*Model*. Werden neue Bilder ohne Markierung

[1] Informationen aus [RRC19]
[2] [GRDW18], [HSS+16]

hinzugefügt, kann das *Model* die Buchstaben erkennen. Diese Art um das Problem zu lösen wird im Kontext von ML als Klassifizierung bezeichnet. Es gibt auch noch zwei weitere, welche adressiert werden:

- Prognose (engl. *Prediction* oder *Regression*): Trainieren eines *Models* mit historischen Daten zur Vorhersage zukünftiger Werte. Z.B. der Bedarf eines bestimmten Produktes in den Sommerferien.
- Klassifizierung (engl. *Classification*): Datenpunkte in eine oder mehrere Kategorien bzw. Klassen einteilen. Z.B. Identifizierung einer Email als Spam oder nicht, Erkennen welches Tier auf einem Bild zu sehen ist (Katze, Hund, Löwe, ...).
- Gruppierung (engl. *Clustering*): Unterteilung vieler Datenpunkte in wenige Gruppen, in denen Punkte mit gleichen Eigenschaften enthalten sind. Im Gegensatz zur Klassifizierung ist die Anzahl der Gruppen im Vorhinein nicht bekannt.

Zur besseren Vorstellung veranschaulicht Abbildung 2.1 die Arten. Abschnitt 2.1.2 und folgende gehen darauf näher ein.

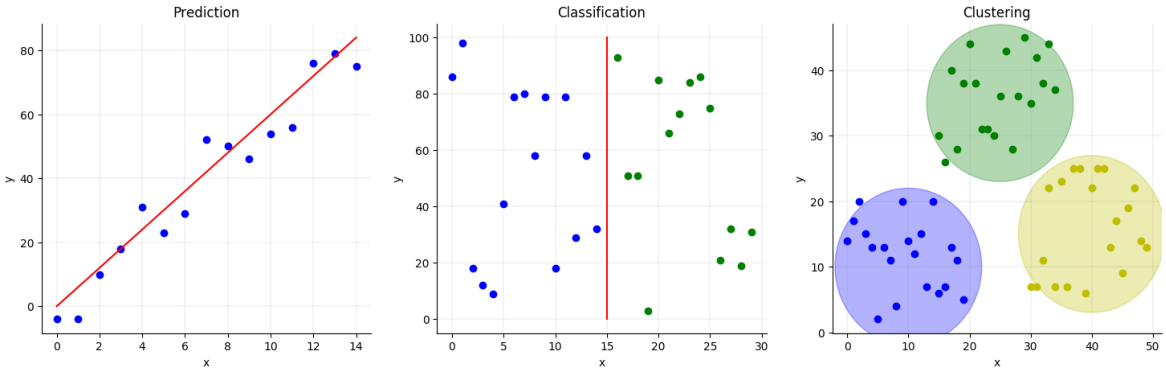


Abbildung 2.1.: Arten von *Machine Learning* Problemen

2.1.1. Ansätze

Um ein ML-*Model* anzulernen, gibt es verschiedene Möglichkeiten. An eine davon - das Überwachte Lernen - wurde bereits in der Einführung anhand eines Beispiels herangeführt. Dieses Unterkapitel geht auch auf die anderen Möglichkeiten ein.

Überwachtes Lernen

Beim Überwachten Lernen (engl. *supervised learning*) wird dem *Model* ein Datenset bestehend aus Datenpunkten und den dazugehörigen korrekten Antworten zu einer Frage übergeben. Der ML-Algorithmus versucht aufgrund der Eigenschaften eines Datenpunktes einen Zusammenhang zu der Antwort zu finden. Wenn daraufhin neue Datenpunkte hinzugefügt werden, kann das *Model* basierend auf den Eigenschaften eine Antwort prognostizieren. Das folgende abstrakte Beispiel demonstriert wie *supervised learning* funktioniert.

Seit der Geburt lernt ein Kind, wie bestimmte Objekte aussehen und wie sie heißen. Es sieht jeden Tag einen Hund in verschiedenen Positionen, mal sitzend, laufend, stehend usw. Dadurch kann es auch den Nachbarshund als solchen identifizieren und zu einer Katze unterscheiden. Während der Lernphase kann natürlich auch ein Fehler vorkommen und das Kind kann zum Beispiel einen Wolf als einen Hund missinterpretieren. Doch die Eltern und

Geschwister erklären dem Kind, dass es sich dabei nicht um einen Hund handelt. Es passt daher das Verständnis an und wird immer besser, das Tier zu identifizieren. Im Kontext von ML ist die oben beschrieben Anwendung zur Erkennung von handgeschriebenen Buchstaben genauso überwachtes Lernen.

Nicht-Überwachtes Lernen

Bei dieser Art des Lernens sind im Datenset nicht die korrekten Antworten zu den einzelnen Datenpunkten enthalten. Der Algorithmus ist darauf ausgelegt Trends zu erkennen und Datenpunkte mit Gemeinsamkeiten zu gruppieren, ohne zu wissen, worum es sich dabei handelt. Um auf das oben beschrieben Beispiel mit dem Hund zurückzukommen, werden in einem *Model* viele Bilder von verschiedenen Tieren eingespielt. Die Bilder sind jedoch nicht mit dem darauf enthaltenen Tier markiert. Nachdem der Algorithmus alle Charakteristiken der Bilder analysiert hat, ist das *Model* in der Lage, gleichartige Bilder zusammenzufassen. Es kann somit eine Gruppe erstellen, die alle Hunde enthält und eine andere, der alle Katzen zugeordnet sind. Am Ende hat das ML-*Model* noch immer keine Vorstellung, um welches Tier es sich tatsächlich handelt.

Ein anderes Beispiel ist in der Analyse von Kaufverhalten zu finden. Das Datenset besteht hierbei aus den verschiedenen gekauften Produkten der Kunden und das Ziel ist Korrelationen darin zu finden. Das *Model* soll in der Lage sein zum Beispiel folgendes bestimmen zu können: Kunden die Schultaschen kaufen, kaufen auch Stifte. Oder Kunden die Bier kaufen, kaufen auch Chips.

Teil-Überwachtes Lernen

Teil-Überwachtes Lernen (engl. *semi-supervised learning*) liegt zwischen den beiden vorher beschriebenen Arten des Lernens. Es wird ein Datenset verwendet, das hauptsächlich aus unmarkierten Datenpunkten besteht. Ein kleiner Teil davon hat jedoch eine Markierung. Im ersten Schritt kommen Techniken zur Gruppierung zum Einsatz, um gleichartige Datenpunkte zu bündeln. Der zweite Schritt besteht darin, die bereits bekannten Datenpunkte dazu zu verwenden, andere Daten in der gleichen Gruppe zu markieren. Einer der größten Vorteile davon ist, dass nicht viel Zeit für das manuelle Markieren von Daten aufgewendet werden muss. Der Nachteil ist aber, dass es im Vergleich zum Überwachten Lernen komplexer ist.

Auch hier lässt sich das Beispiel mit den Tieren anwenden. Aus Zeit- oder anderen technischen Gründen können nicht alle Bilder von Hunden und Katzen durchgesehen und markiert werden. Bei ein paar ist es jedoch möglich. Beim Teil-Überwachten Lernen gruppiert zuerst das ML-*Model* alle Bilder mit Hunden und Katzen (gleiche Eigenschaften). Danach können aus den paar Markierten alle anderen abgeleitet werden.

Bestärkendes Lernen

Die letzte hier vorgestellte Art ist bestärkendes Lernen (engl. *reinforcement learning*). Es kommt einerseits zum Einsatz, wenn sich die Situationen fortlaufend ändern, zum Beispiel beim Autofahren oder bei einem Spiel. Das *Model* muss sich hierbei immer an neue Bedingungen anpassen. Andererseits wird es auch verwendet, wenn ein großer Zustandsraum existiert. Bei beispielsweise Schach ist es kaum möglich mittels *brute-force* den besten nächsten Zug herauszufinden, da es viel zu viele Möglichkeiten im Verlauf eines Spieles gibt. Der Algorithmus ist also darauf ausgelegt eine Entscheidung basierend auf dem momentanen internen Zustand und der Umgebung zu treffen, um ein vordefiniertes Ziel zu erreichen. Ein Ziel kann

gegebenenfalls sein, ein Auto innerhalb der Spur zu halten. Je länger der Algorithmus lernen kann, desto besser und genauer werden die Entscheidungen in Hinblick auf eine langfristige Auswirkung.

2.1.2. Regression

Die Regressionsanalyse (engl. *regression analysis*) ist eine auf Statistik basierende *Machine Learning*-Technik, welche aufgrund von oftmals historischen Daten zur Vorhersage verwendet wird. Dabei werden Relationen in markierten Daten analysiert, um eine Prognose für bestimmte Eigenschaften treffen zu können. Zum Beispiel kann damit ein *Model* erstellt werden, welches den aktuellen Preis einer Immobilie bestimmt. Die historischen markierten Daten können hier die Eigenschaften Quadratmeteranzahl, Nachfrage, Lage als Kategorie, Kaufpreis, Verkaufspreis etc. haben. Im Kontext von ML werden die Eigenschaften als *Features* bezeichnet. Da die Problemstellung in der Masterarbeit eine Klassifizierung erfordert, wird nur ein Algorithmus kurz vorgestellt.

Linear Regression

Bei dieser Art der Regression wird versucht, den Wert einer kontinuierlichen Variable vorherzusagen, beispielsweise den Tagesendkurs eines bestimmen Unternehmens an der Börse. Einfachheitshalber wird angenommen, dass der aktuelle Aktienkurs nur von den Kursen und dem Volumen der letzten zehn Tage abhängt. Das ML-*Model* würde daher auf der folgenden Formel basieren:

$$Kurs = Konstante + a * Kurs_{\text{Tag-1}} + b * Volumen_{\text{Tag-1}} + c * Kurs_{\text{Tag-2}} + d * Volumen_{\text{Tag-2}} + \cdots + s * Kurs_{\text{Tag-10}} + t * Volumen_{\text{Tag-10}}$$

Wenn die Koeffizienten a, b, c, etc. bekannt sind, kann der Kurswert an Tag X berechnet werden. Hierfür werden historische Daten verwendet. Sie bestehen jeweils aus einem *Input*/*Output* Paar. Die *Input* Daten sind die letzten zehn Kurswerte und Volumina und *Output* ist der Kurswert am dazugehörigen Tag. Wenn der Wert der Koeffizienten ermittelt wird, der die meisten historischen Daten mit akzeptablen Fehlern erfüllt, kann das *Model* den Aktienkurs des nächsten Tages vorhersagen. Da ausschließlich markierte Daten zum Lernen verwendet werden, fällt lineare Regression unter überwachtes Lernen.

2.1.3. Classification

Wie bereits in der Einleitung des Kapitels beschrieben, geht es bei der Klassifizierung um das Einteilen von Datenpunkten in vordefinierte Kategorien. Es fällt ebenfalls unter *supervised learning*, da die Modelle mit bereits kategorisierten Daten trainiert werden. Ein *Classifier* kann entweder binäre Entscheidungen treffen oder aus einem Set von Kategorien wählen.

Logistic Regression

Logistic Regression kommt zum Einsatz, wenn der Prognoseraum binär ist - also 0 oder 1, ja oder nein. Um auf das vorherige Beispiel mit den Aktienkursen zurückzukommen: Eine Anwendung soll herausfinden, ob der morgige Aktienwert höher ist, als der heutige. Es könnte durchaus mit der linearen Regression ebenso erreicht werden, indem man den aktuellen Wert mit dem prognostizierten vergleicht. Jedoch ist logische Regression für solche Problemstellungen die bessere Methode.

Random Forest

Der Algorithmus *Random Forest* gilt als einer der effektivsten und intuitivsten, wenn es um das Lösen von Klassifizierungsproblemen geht. Er basiert auf sogenannten *Decision Trees*, welche anhand der *Features* einen Entscheidungsbaum aufbauen. Mehrere von diesen bilden einen *Forest*. Ein *Decision Tree* ist eine Baumstruktur bestehend aus Knoten (engl. *Node*), welche eine deterministische Entscheidung basierend auf Variablen treffen, und Kanten, welche zum nächsten Knoten oder zum sogenannten *leaf node* führen. Dieser letzte Knoten stellt das Ergebnis dar, beziehungsweise die resultierende Klasse.

Zum Beispiel besteht ein Datenset aus Bildern von Fortbewegungsmitteln. Mithilfe einiger Fragen, welche sich auf die *Features* beziehen, über ein Objekt soll herausgefunden werden, ob es sich dabei um ein Auto oder ein Fahrrad handelt. Diese Fragen könnten sein: Hat es einen Motor(?), können mehr als eine Person es benutzen(?), hat es ein Lenkrad(?), wie viele Reifen hat es(?). Jeder Knoten repräsentiert eine dieser Fragen und dessen Kanten führen zum nächsten basierend auf der Antwort. Wenn alle Fragen richtig beantwortet sind, führt der letzte Knoten zum Ergebnis - also ob auf dem Bild ein Auto oder ein Fahrrad zu sehen ist. Abbildung 2.2 zeigt einen *Decision Tree* in einer vereinfachten Form.

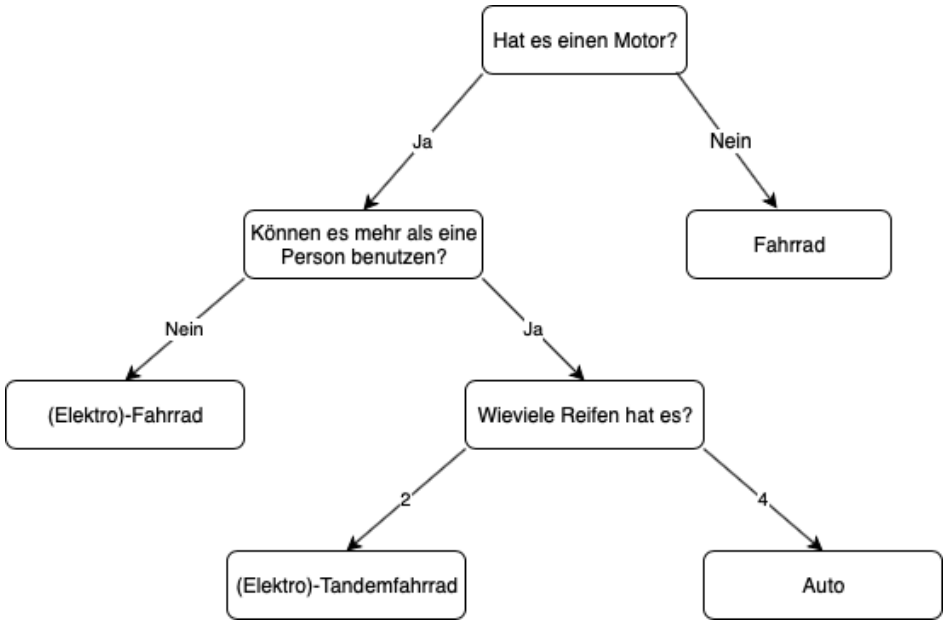


Abbildung 2.2.: Beispiel *Decision Tree*

Das Ziel eines *Decision Trees* ist also ein ML-*Model* mit Entscheidungsregeln basierend auf Trainingsdaten zu erstellen. Anhand dieser Regeln soll eine Kategorie vorhergesagt werden können. Die Erstellung erfolgt dabei in drei Schritten:

Schritt 1 Verwende das Attribut oder *Feature* mit der größten Bedeutung als *root*.

Schritt 2 Erstelle eine Entscheidung mit dem höchsten Informationsgehalt.

Schritt 3 Wiederhole rekursiv Schritt 1 und 2 bis der Baum fertig aufgebaut ist und es keine Informationen mehr zum Aufteilen gibt.

Wenn viele *Features* in einem Datenset vorhanden sind, ist die größte Herausforderung, die beste Variable oder eine Kombination aus mehreren für einen Knoten zu finden (für Schritt 2). Dieser Vorgang wird *attribute selection* genannt und wird entweder durch *information gain* oder *Gini Impurity* durchgeführt. Ersteres kommt eher zum Einsatz, wenn die *Output-*

Variable eine Kategorie ist (z.B.: Fortbewegungsmittel) und zweiteres wenn sie numerisch und kontinuierlich ist (z.B.: Alter einer Person).

In der Informationstheorie kann der Informationsgehalt einer Variable mithilfe der Entropie bestimmt werden. Wenn die Auftrittswahrscheinlichkeit eines Wertes klein ist, ist der Informationsgehalt hoch. Bei einer großen Wahrscheinlichkeit sinkt der Informationsgehalt. Entropie ist nun die Summe aller Auftrittswahrscheinlichkeiten der möglichen Werte und wird mit der Formel 2.1 berechnet. So hat zum Beispiel ein Würfelwurf eine höhere Entropie (2,58) als ein Münzwurf (1) und würde den Vorzug bei dem Aufteilen eines Datensets erhalten.

$$H = -\sum_{i=1}^{n}(p_i log_2 p_i) \tag{2.1}$$

wobei

p_i ist die Auftrittswahrscheinlichkeit des $i$ten-Elements.

n ist die Anzahl an möglichen Werten.

H ist die Entropie.

Das *Gini Impurity Criterion* auf der anderen Seite bestimmt die Wahrscheinlichkeit wie oft ein zufällig gewählter Datenpunkt aufgrund eines bestimmen *Features* falsch klassifiziert wird. Je kleiner dieser Wert ausfällt, desto besser eignet sich das *Feature* zum Aufteilen bei einem Knoten und wird bevorzugt. Die vereinfachte Formel hierfür ist:

$$Ic(p) = 1 - \sum_{i=1}^{J} p_i^2 \tag{2.2}$$

wobei

$Ic(p)$ ist die *Gini Impurity*.

p_i ist die Wahrscheinlichkeit des $i$ten-Elements.

J ist Anzahl der Kategorien.

Ein *Random Forest* bestimmt nun durch ein Mehrheitsvotum der Ergebnisse aus den verschiedenen *Decision Trees* die höchstwahrscheinlich zutreffende Kategorie. Abbildung 2.3 veranschaulicht dies. Würde jeder *Tree* mit den gleichen Daten und Parametern trainiert werden, ist das Ergebnis überall das gleiche und es ergibt sich dadurch kein Vorteil. Es kommt daher *Data Bagging* zum Einsatz. Dabei wird für jeden Baum ein eigener Beutel erstellt, indem zufällige 62,3% der Daten des originalen Datensets hineingegeben werden. Die 62,3% sind laut [RRC19] vom Algorithmus festgelegt. Das bedeutet, dass sich einerseits die gleichen Datensätze in jedem der Beutel befinden und andererseits auch, dass Daten mehrfach in einem Beutel vorhanden sind. Die Anzahl muss nämlich überall mit dem originalen Set übereinstimmen. Danach wird *Feature Bagging* durchgeführt. Für die nun vorhandenen Beutel werden nicht alle *Features* verwendet, sondern nur zufällige $\sqrt{n}$ (Standardparameter). Hier werden aber auch mehrere Beutel mit unterschiedlichen *Features* erstellt. Mithilfe von *cross validation* (CV) wird herausgefunden, welcher *Feature Bag* am besten für die Daten in Frage kommt und nur dieser wird schlussendlich für das Trainieren des *Decision Trees* verwendet. Bei der *cross validation* werden jeweils die 37,7% der Daten verwendet, welche nicht in einem Beutel vorhanden sind. Dieser Prozess geschieht während der Auswahl für den besten *Feature Bag*. Das heißt, dass das Model ohne jegliche Testdaten bereits in der Erstellung eine Aussage über die Genauigkeit treffen kann. Ein weiterer Vorteil der sich daraus ergibt, ist, dass anhand CV und des Informationsgehaltes ein Rückschluss auf aussagekräftige *Features* gezogen werden

kann. Als Optimierung können unwichtige eliminiert werden und damit Rechenleistung sowie -Zeit gespart werden.

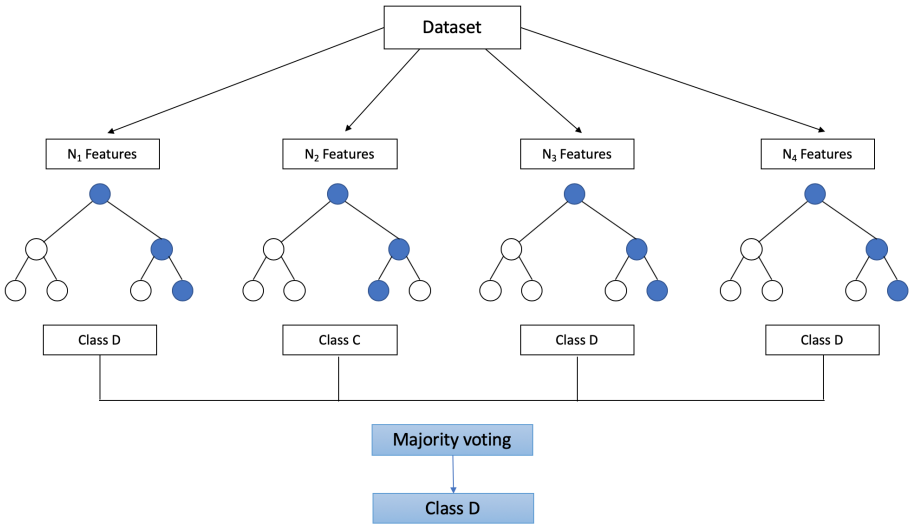


Abbildung 2.3.: Darstellung *Random Forest*

2.1.4. Hardware

Die meisten Algorithmen für Künstliche Intelligenz, beziehungsweise speziell für *Machine Learning*, sind sehr rechenaufwendig [DD18]. Grund dafür ist, dass sehr viele Daten verarbeitet werden müssen. Bei der Sub-Domäne *Deep Learning* sind die Datenmengen noch mehr, da es sich hierbei meist um Bild-, Video- oder Audioaufnahmen handelt. Die darunterliegende Hardware ist daher von sehr großer Bedeutung, um Systeme effizient entwickeln, trainieren und einsetzten zu können.

Traditionelle Computersysteme wie *Workstations* und Laptops sind anfangs die offensichtlichste Wahl für die Entwicklung. Es erlaubt eine schnelle Implementierung der ersten Ml-Modelle, welche kurzerhand gleich auf der lokalen *Central Processing Unit* (CPU) validiert werden können. Dadurch ist auch eine hohe Flexibilität gegeben, was das Optimieren und Debuggen erleichtert. Dies funktioniert jedoch nur, wenn die Menge an Trainingsdaten überschaubar bleibt. Soll ein Modell für den Betrieb trainiert werden, kann es mitunter Tage oder sogar Wochen dauern, bis es fertig durchgerechnet ist. Die Architektur der CPU erlaubt es nämlich nicht, viele Berechnungen parallel auszuführen. Die *Graphics Processing Unit* (GPU) hingegen bietet einen sehr hohen Parallelisierungsgrad, da Kalkulationen für Pixel und Matrizen meist einfach sind, aber oft durchzuführen sind. Bei *Machine Learning* Algorithmen handelt es sich um ähnliche Aufgaben. Aufgrund dessen haben nach und nach immer mehr Entwickler GPUs für ML verwendet. Diesen Trend haben auch Hersteller erkannt und bieten GPUs mit spezieller Unterstützung für dieses Feld an.

Grafikkarten unterscheiden sich zu CPUs im Kontext von ML in drei großen Punkten. Erstens besitzen GPUs eine große Anzahl an kleinen Prozessoren, was die oben angesprochene Parallelisierung ermöglicht. Zweitens können viel höhere Datenmengen verarbeitet werden, da der lokale Speicher größer ist. Dadurch muss keine Zeit aufgewendet werden, um Daten von externen Speicherträgern zu laden, beziehungsweise zu schreiben. Das dritte Unterscheidungsmerkmal ist die Möglichkeit, *floating point* Operationen durchführen zu können. Sie

erlauben einen größeren Zahlenbereich bei der gleichen Anzahl an Bit. Vor allem hinsichtlich der Genauigkeit bringt dies einen Vorteil.

Field Progammable Gate Arrays (FPGAs) ist eine andere Chipklasse, die immer populärer für ML wird [RRC19]. Sowie GPUs stellen FPGAs eine hohe Rechenleistung und Parallelisierung bereit. Zu dem kommt noch ein viel höherer On-Chip Speicher mit einem generell geringeren Stromverbrauch. Weitere Vorteile sind, dass auch verschiedenartige Operationen gleichzeitig ausgeführt werden können und es eine Unterstützung von mehreren Datentypen gibt. Im Bereich funktionale Sicherheit sind diese Chips ausgereifter und werden daher häufiger für sicherheitskritische Applikationen wie Autonomes Fahren eingesetzt. Im Vergleich zu GPUs ist es komplexer, darauf aufbauende Software zu implementieren.

Das Unternehmen *Google* hat für seine *(Deep) Machine Learning* Anwendungen (Spracherkennung, Übersetzer, ...) einen eigenen *Application-Specific Integrated Circuit* (ASIC) namens *Tensor Processing Unit* (TPU) entworfen und entwickelt. Die Architektur unterscheidet sich in der von GPUs in dem Sinne, dass sie keine komplett parallelen Prozessoren besitzen, sondern lediglich ein Set von *floating point* Instruktionen. Durch eine höchst effiziente Umsetzung sind TPUs 15 bis 30-mal schneller als GPUs bei einem geringeren Stromverbrauch. *Tensor Processing Units* sind aber nicht käuflich zu erwerben und kommen daher nicht für einen lokalen Einsatzzweck infrage.

2.1.5. Machine Learning-as-a-Service

Machine Learning-as-a-service (MLaaS) ist ein Überbegriff von *Cloud*-basierten Technologien, welche für ML-Anwendungen benötigt werden. Es wird Datenvorbearbeitung, Trainieren und Validierung der Modelle, sowie der Betrieb zur Verfügung gestellt. Von *Google* wird dies unter dem Namen *Google Cloud AI* für Kunden angeboten [Goo]. Hierbei kommen die oben erwähnten TPUs mit dem hauseigenen AI Framework *TensorFlow* zum Einsatz. Es werden drei Produkte unterschieden. *AI Hub* ist ein Repository von Plug-and-Play Komponenten, die einen schnellen Einstieg ermöglichen und für eine bessere Zusammenarbeit innerhalb einer Organisation dienen. *KI-Bausteine* sind unter anderem *Vision AI* zur visuellen Erkennung, *Natural Language* für z.B. Übersetzungen und *AutoML Tables* für das strukturieren von Trainingsdaten. *AI Platform* ist ein codebasierte Entwicklungsumgebung für ML-Entwickler.

Die *Amazon Machine Learning* Dienste sind eine der automatisiertesten Lösungen in diesem Bereich [Ama]. Es werden binäre und multiclass Klassifikationen sowie Regression bereitgestellt. Der Kunde muss dabei lediglich die Zielvariable festlegen und im Hintergrund werden die Daten vorbearbeitet, der richtige Algorithmus ausgewählt und optimiert. Durch den hohen Automationsgrad geht jedoch die Flexibilität verloren. Deshalb gibt es auch *SageMaker*, was eine eigene Umgebung für *Data-Scientists* ist und zur Erstellung eigener Modelle dient.

Microsoft hat mit *Azure Machine Learning Studio* ebenfalls eine Lösung am Markt [MS2]. ML Studio ist dem Produkt von *Amazon* sehr ähnlich. Jede Tätigkeit kann mittels *drag and drop* durchgeführt werden. Dadurch ist eine geringe Lernkurve für Einsteiger gegeben. Mit über 100 Methoden für Klassifizierung und Regression, aber auch Anomalieerkennung wird dem Kunden sehr viel geboten.

Neben den genannten Unternehmen existieren noch *IBM*, *SAP*, *BigML* sowie weitere. Jeder von ihnen stellt neben browserbasierten Oberflächen auch *Representational State Transfer-Application Programming Interfaces* (REST-APIs) zur Verfügung, mit denen Aufgaben automatisiert abgearbeitet werden können.

2.2. CAN Bus

Der *Controller Area Network* Bus ist einer der bekanntesten Übertragungsmöglichkeiten in einem Fahrzeug und findet auch Verwendung in anderen Industrieanlagen wie beispielsweise Fahrstühlen [ZS14]. Er verbindet mehrere Steuerungseinheiten (ECUs) zum Austausch von fahrzeugrelevanten Daten. Wie bereits in der Einführung erwähnt, sind typische Nachrichten die Motordrehzahl, Bremsdruck, Blinker etc. Es gibt jedoch auch eine Reihe von weiteren Bus-Systemen, welche im Einsatz sind, andere Anforderungen haben und für verschiedene Zwecke benötigt werden. Prominente Beispiele sind *FlexRay*, *LIN* und *MOST*. Sie werden in sechs Klassen und nach Übertragungsrate eingeteilt (siehe Tabelle 2.1).

Tabelle 2.1.: Klassifikation der Busssysteme nach Übertragungsrate [ZS14]

Klasse	Bitrate	Typischer Vertreter	Anwendung
Diagnose	< 10 Kbit/s	ISO 9141 K-Line	Werkstatt- und Abgastester
A	< 25 Kbit/s	LIN	Karosserieelektronik
B	25 ... 125 Kbit/s	CAN (Low Speed)	
C	125 ... 1000 Kbit/s	CAN (High Speed)	Antriebsstrang, Fahrwerk, ...
D	> 1 Mbit/s	FlexRay	Bachbone-Netz
Infotainment	> 10 Mbit/s	Ethernet	Multimedia

Der CAN Bus wurde in den 1980er Jahren vom Unternehmen *Bosch* mit dem Ziel entwickelt, die Kabelbäume zu verkürzen, um so Gewicht und Material zu sparen. Damals wurden bis zu 2 km Kabel in Fahrzeugen eingezogen, was mit der steigenden Anzahl an Steuergeräten mehr und mehr geworden wäre, da die Verbindungen ausschließlich Punkt zu Punkt waren. Seit 1991 findet der Bus als erstes den Einzug in Klasse C-Netze und stellt seither die Spezifikation für die Basis jeglicher CAN-Implementierungen dar. Die Konzipierung des Buses ist so ausgelegt, dass eine hohe Fehlersicherheit bei kurzen Botschaftslängen für einen Echtzeitbetrieb gegeben ist.

In der Tabelle ist ersichtlich, dass der Einsatzbereich auch in der Klasse B liegt, welche die Komfort-Systeme umfasst. Generell wird zwischen High- und Low-Speed unterschieden. Die Übertragungsrate des High-Speed-CAN ist bei 125 Kbit bis zu 1 Mbit pro Sekunde. Er verbindet maximal 20 ECUs des Antriebsstrangs (Motorelektronik, Getriebe- und Lenkradsteuerung, ABS, etc.). Aufgrund der hohen Sicherheitsrelevanz ist eine möglichst robuste Datenübertragung gefordert. Ein Fehler gerade im Automotive-Bereich könnte verheerende Folgen haben, wenn es dadurch zu einem Unfall kommt. Der Low-Speed-Can ist hingegen nicht so sicherheitskritisch, da vorwiegend Steuergeräte für zum Beispiel die Scheibenwischer, Zentralverriegelung oder Klimaanlage vernetzt werden. Die Datenrate beschränkt sich bei diesem Bus auf maximal 125 Kbit/s bei maximal 30 Knoten.

Die höchstmögliche Geschwindigkeit von einem Mbit pro Sekunde kann jedoch nur bei einer Strecke mit bis zu 40 Metern erreicht werden. Je länger die Kabel werden, desto geringer wird die Übertragungsrate. Der Grund für diese Einschränkung liegt bei der gleichzeitigen Datenverarbeitung von Busteilnehmern. Tabelle 2.2 zeigt weitere Längen zu Übertragungsraten.

Tabelle 2.2.: CAN Übertragungsrate zur Kabellänge

Übertragungsrate [kbit/s]	10	20	50	125	250	500	1000
Kabellänge [m]	6700	3300	1300	530	270	130	40

In der CAN-Spezifikation ist im Wesentlichen nur der *Data-Link Layer* beschrieben, was zu unterschiedlichen Umsetzungen des *Physical Layer* geführt hat. Im Allgemeinen ist CAN ein bitstrom-orientierter Linien-Bus mit einer Zwei-Draht Leitung (CAN-High und CAN-Low). An beiden Enden muss der Bus mit einem 120 Ohm Widerstand terminiert werden, um Reflexionen zu verhindern. Über beide Leitungen werden zwei komplementäre Signale übertragen, sodass bei einer Unterbrechung noch immer die Daten zugestellt werden können. Für die Übertragung selbst unterscheiden sich die Bit im Signalpegel. Eine logische *1* wird hochohmig (*rezessiv*) erzeugt und ein niederohmiges Signal (*dominiant*) definiert eine logische *0*.

2.2.1. Data-Link Layer

Der CAN-Bus ist ein *Broadcast*-Medium, bei dem die Nachrichten keine Ziel-und Quelladresse inkludieren und jedes Steuergerät jede Nachricht empfängt. Die Botschaft beinhaltet stattdessen eine eindeutige Kennung, den *Message Identifier*, durch welchen die ECUs entscheiden, ob sie eine Nachricht weiter verarbeiten oder verwerfen. Ursprünglich war die Kennung mit 11 Bit spezifiziert, aber aus Erweiterungsgründen wurde sie später auf 29 Bit festgelegt. Zusätzlich kommen noch 1 beziehungsweise 3 Steuerbits hinzu. Ein Steuergerät kann auf den Bus zugreifen, wenn mindestens drei Bitzeiten nichts gesendet/empfangen wurde. Dabei wird das CSMA/CR Verfahren mit Arbitrierung auf Basis von Prioritäten eingesetzt. Der *Message Identifier* dient neben der Kennung hierfür auch als Angabe der Priorität einer Nachricht. Je niedriger die Zahl, desto höher die Priorität. Kommt es also auf dem Bus zu einer Kollision, "gewinnt"die Botschaft mit der höheren Kennung.

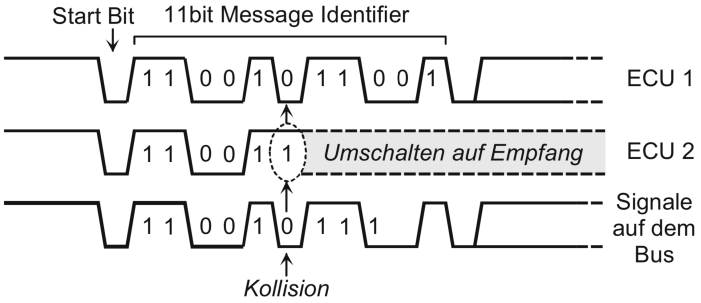


Abbildung 2.4.: Kollision von CAN-Nachrichten (Quelle: [ZS14])

Abbildung 2.4 zeigt Verhalten, wenn zwei Steuergeräte gleichzeitig auf den Bus senden. ECU1 startet mit dem Senden des *Message Identifiers 110 0101 1001*$_{\mathrm{B}}$, währendessen auch ECU2 mit der Kennung *110 0111 0000*$_{\mathrm{B}}$ beginnt. Beim sechsten Bit kommt es zu einer Kollision, da sich dort die Kennungen unterscheiden. Auf der Leitung dominiert jedoch die *0*, da das Signal wesentlich niederohmiger im Vergleich zur *1* ist. Die Steuergeräte überprüfen beim Senden, ob der Signalpegel auch dem jeweiligen gesendeten Bit entspricht und so erkennt ECU2, dass es zu einer Kollision gekommen ist. Es stellt sofort das Senden ein und auf Empfangen um. ECU1 führt das Übertragen unverzögert fort. Frühestens nach dem Ende der Nachricht und drei freien Bitzeiten versucht das zweite Steuergerät das erneute Versenden.

Die *Payload* einer CAN-Nachricht kann zwischen null und acht Byte lang sein, wobei die Anzahl im *Data Length Code* (DLC) Feld innerhalb der *Control Bits* steht. Eine 15 Bit lange Prüfsumme (*Cyclic Redundancy Check*, CRC) dient zur Fehlererkennung. Die Bitgeneratoren der Empfänger synchronisieren sich mithilfe des Startbits des Senders und werden durch zusätzlich eingefügte *Stuff-Bits* nachsynchronisiert. Nachdem ein CAN-Controller eine

Nachricht empfangen hat, wird das Paketformat und der CRC-Code überprüft. Stimmt alles überein, sendet der Controller innerhalb des *Acknowledge und End of Frame* Felds eine positive Empfangsbestätigung. Andernfalls ein *Error Frame*, was zu einem Ignorieren der Nachricht bei allen anderen Empfängern führt. Dadurch ist gewährleistet, dass die Daten am Bus und der Teilnehmer konsistent bleiben. Der Sender, bei dem es zu einem Fehler gekommen ist, startet automatisch einen neuen Versuch. Abbildung 2.5 fasst das CAN-Nachrichtenformat noch einmal zusammen.

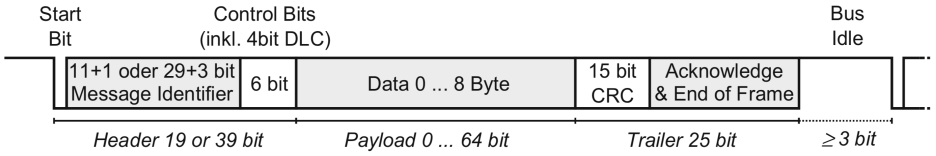


Abbildung 2.5.: CAN-Paketformat (Quelle: [ZS14])

2.2.2. Fehlerbehandlung

Durch diese Fehlererkennungsverfahren ergibt sich eine hohe Übertragungszuverlässigkeit. Viele verschiedene Untersuchungen haben eine Restfehlerwahrscheinlichkeit von 10^{-11} ausgemacht. Auch die Übertragungswiederholung tritt sehr schnell ein, da spätestens am Ende einer Botschaft die Erkennung eines möglichen Fehlers vorliegt. Zusätzlich hat jeder CAN-Controller einen eigenen Fehlerspeicher, in dem Sende- und Empfangsfehler protokolliert werden. Dabei wird unterschieden, ob nur der Controller selbst einen Fehler erkennt, oder ob ihn auch andere feststellen. Erkennt ein Steuergerät, dass wiederholt Sendefehler erzeugt werden, stellt es als ersten Schritt das Senden ein. Kann das Problem mithilfe einer Fehler-Ursachen-Analyse und -Behebung nicht gelöst werden, schaltet sich das Gerät vollständig ab. Dies wird unter dem Begriff *Bus off* verstanden. Tritt jedoch vielleicht nur eine vorübergehende Störung durch elektromagnetische Effekte (EMV-Störung) auf, geht das Steuergerät wieder in Normal-Betrieb über, nachdem diese vorbei sind. Die Anzahl der Fehler, welche toleriert werden und Zeit die verstreichen darf ist entweder von Standards beziehungsweise von der Gesetzgebung vorgeschrieben oder durch den Hersteller festgelegt.

2.2.3. CAN Matrix

Die CAN-Spezifikation gibt keinerlei Vorgaben für Nachrichtenkennungen, Formatierung oder Bedeutung der zu sendenden Nachrichten. Dadurch ist eine sehr hohe Flexibilität und geringer Overhead gegeben, welcher auch für eine solche Umgebung notwendig ist. So legt jeder *Original Equipment Manufacturer* (OEM) oder Zulieferer die Punkte für sich individuell selbst fest. Sie müssen nur innerhalb eines Fahrzeuges einheitlich auf allen Steuergeräten umgesetzt sein. Dies beinhaltet folgendes:

- welche Steuergeräte welche CAN-Nachrichten unten welchen Bedingungen senden/verarbeiten
- den *Message Identifier* bzw. die Priorität der Nachrichten
- mit welchem Zyklus gesendet wird
- welche Daten oder Signale enthalten sind
- die Normierung der Daten, bzw. die Umrechnungsbeziehung zwischen den Rohdaten und den physikalischen/logischen Größen

All diese Informationen sind in der sogenannten CAN-Matrix (CAN-Datenbank, CANdb, DBC-Datei) enthalten und sind auf jedem Busteilnehmer zu implementieren. Erst mit einer solchen Datei ist es möglich, den CAN-Datenstrom auszuwerten und im Netzwerk teilzunehmen. Das Listing 2.1 zeigt einen Ausschnitt eines DBC-Files mit dem Signal *Bremsdruck*.

```
BO_ 262 ESP_05: 8 Bremse_ABS_MK100
    ...
    SG_ ESP_Schwelle_Unterdruck : 14|2@1+ (1.0,0.0) [0.0|3] "" Gateway_MQB
    SG_ ESP_Bremsdruck : 16|10@1+ (0.3,-30) [-30.0|276.6] "Unit_Bar"
        Allrad_Haldex_TT3,Daempfer,Gateway_MQB,Lenkhilfe_EPS,Magnetic_Ride,
        PDC_PLA,PDC_PLA_ARA
    SG_ ESP_Fahrer_bremst : 26|1@1+ (1.0,0.0) [0.0|1] "" Allrad_Haldex,
        Allrad_Haldex_TT3,Gateway_MQB,Lenkhilfe_EPS,VAQ
    ...
```

Listing 2.1: Ausschnitt CAN DBC-File

In der ersten Zeile ist die CAN-Nachricht definiert. *262* ist der *Message Identifier*, *ESP_05* heißt die Nachricht, die Länge (DLC) ist *8* und wird von dem Steuergerät *Bremse ABS MK100* versendet. Die Zeilen darunter spezifizieren die Signale, welche in der Nachricht enthalten sind. Anfangs steht die Bezeichnung, nach dem ":"die Stelle des ersten Bits und die Anzahl der Bit. Die restlichen Felder legen die Skalierung, das Offset, das Minimum und das Maximum, sowie, falls vorhanden, die Einheit fest. Am Ende werden alle Steuergeräte gelistet, welche das Signal weiterverarbeiten dürfen.

2.2.4. MDF

Measurement Data Format (MDF) [fSoAS14] ist ein binäres Dateiformat, welches 1991 von der Firma *Vector Informatik GmbH* in Zusammenarbeit mit der *Robert Bosch GmbH* entwickelt wurde. Es ist speziell für Messdaten im Automotive-Bereich konzipiert und ist seit 2009 in der Version 4 als offizieller Standard von *Association for Standardization of Automation and Measuring Systems* (ASAM) öffentlich zugänglich. Ein wesentlicher Vorteil des Formates ist, dass die Messdaten sehr schnell und speicherplatzsparend abgespeichert werden können. Des Weiteren wird auch eine Optimierung des Lesevorgangs durch Vorsortierung und Indizierung unterstützt. Neben den eigentlichen Nutzdaten werden auch Metadaten aus der DBC-Datei mitgespeichert. Dies ist vor allem für weitere Analysen und die Interpretation der Rohdaten notwendig. Beispielsweise gehören dazu die Information zur Umwandlung in physikalische Werte und Signalnamen.

2.3. Edge Computing

Cloud Computing hat einen enormen Einfluss auf das tägliche Leben und die Arbeitswelt in den verschiedensten Bereichen genommen. *Software-as-a-Service* (SaaS) Produkte wie die *Google-Apps*, *Social Media*, *Github*, *Office 365* sowie das Verarbeiten von großen Datenmengen für *Machine Learning* Anwendungen sind kaum mehr wegzudenken, beziehungsweise zu realisieren ohne dieser Möglichkeit. Zudem sind viele Unternehmen auf die Infrastrukturen der *Cloud* (*Infrastructure-as-a-Service*, IaaS) angewiesen, um den Betrieb aufrecht erhalten zu können [SCZ+16]. Mit dem Aufmarsch der IoT ist die *Post-Cloud*-Ära angebrochen, in der mit dem Internet verbundene Geräte Daten erzeugen, verteilen aber auch speichern, analysieren und verwenden. Mit dem Jahr 2020 prognostiziert Cisco 50 Milliarden IoT-Geräte, deren Anwendungen kurze Antwortzeiten brauchen, persönliche Daten verarbeiten oder so eine große Datenmenge erzeugen, dass es eine Herausforderung für die Netzwerke darstellt

[Eva11]. Da es dann meist der Fall ist, dass dieselben Geräte wiederum Daten von der *Cloud* empfangen und darauf zum Beispiel eine Aktion setzen, macht es Sinn, die Daten gleich an der selben Stelle zu verarbeiten. Dies wird unter *Edge Computing*, also dem Erzeugen, Auswerten und Verarbeiten von Daten an dem äußersten Knoten des Netzwerks, verstanden [SCZ+16].

Den ganzen Rechenaufwand in die *Cloud* auslagern hat sich für die letzten Jahre als eine effiziente und praktikable Lösung erwiesen. Je mehr Daten jedoch von IoT-Geräten erzeugt und an Rechenzentren gesendet werden, desto eher kommt es zu einem Flaschenhals bei den Transportwegen. Das resultiert unmittelbar in einer höheren Latenz und wirkt sich in weiterer Folge auf eine schlechte Benutzererfahrung der Anwendungen aus. Selbstfahrende Autos zeigen ein zusätzliches Problem, das daraus entsteht, auf. Sie generieren pro Sekunde ein Gigabyte an Messdaten, auf Basis dessen in Echt-Zeit eine Entscheidung getroffen werden soll [CW1]. Bei einer Übertragung all dieser Daten würde die Antwort zu lange auf sich warten lassen. Einerseits stößt das lokale Funknetz in Sachen Bandbreite, Zuverlässigkeit und Verfügbarkeit an seine Grenzen, wenn viele Autos es gleichzeitig tun. Andererseits können selbst im größten Rechenzentrum nicht alle Anfragen zeitgleich und schnellstmöglich abgearbeitet werden, sodass eine weitere Wartezeit hinzukommt. Die Autos wären somit kaum imstande eine Entscheidung rechtzeitig umzusetzen, welche womöglich einen Unfall verhindert. Eine Gruppe von Forschern [YHQL15] hat einen *Proofe-of-Concept* (PoC) mit Bildanalysen vorgestellt, bei dem die Antwortzeit von 900 auf 169 Millisekunden verkürzt werden konnte, indem die Auswertungen von der *Cloud* zum Rande des Netzwerkes verlagert wurden.

Der nächste Punkt, der in *Edge Computing* eine Rolle spielt, ist der Datenschutz (engl. *Privacy*). In einem *Smart Home* kann beispielsweise über den Wasser- und Energieverbrauch festgestellt werden, wie viele Personen sich in dem Haushalt momentan aufhalten. Normalerweise erfolgt die Datenspeicherung und eventuell die Auswertung in der *Cloud*, was natürlich mehr Angriffsvektoren bietet. Hierbei verbleiben aber alle Daten sozusagen im Haus und werden nur lokal verarbeitet, jedoch müssen selbstverständlich auch hier Vorkehrungen bezüglich Datensicherheit getroffen werden. *Wearables* haben zudem noch Zugriff auf Gesundheitsdaten, welche sehr sensible personenbezogene Daten sind. Die Geräte schicken diese zum *Service Provider* für die Speicherung und Aufbereitung. Geschieht der Schritt nicht und die Daten bleiben am Gerät, wo die Benutzerin stets volle Kontrolle darüber hat, ist es leichter, Datenschutz gewährleisten zu können. Der Anwender kann sich weiterhin dazu entschließen, die Daten in die *Cloud* zu senden, um etwaige Dienste zu konsumieren. Essentiell bleibt aber, dass eben die Eigentümerin darüber entscheidet, was mit ihren Daten passiert.

IoT-Geräte beziehungsweise Geräte, welche sich am Rande des Netzwerks befinden, sind oft nicht an das lokale Stromnetz angebunden und so stellt die Batterie eine ausschlaggebende Ressource dar. Alle Berechnungen in die *Cloud* auszulagern hilft dabei Energie zu sparen. Das Übertragen der Daten benötigt jedoch vor allem mit WLAN auch einen hohen Energieaufwand. Hier fließt folgendes mit ein: Um wie viele Daten handelt es sich? Wie oft werden die Daten gesendet? Wie gut ist die Signalstärke? Welche Bandbreite steht zur Verfügung? Soll der Energieverbrauch des gesamten Kommunikationsweges einfließen? Erst wenn all diese Fragen beantwortet sind, kann der Energieverbrauch zur lokalen Verarbeitung gezogen werden. Laut der Literatur [SCZ+16] ist es bei vielen Anwendungen günstiger, zweiteres umzusetzen.

3. Versuchsaufbau

Dieses Kapitel geht auf den Versuchsaufbau des Systems ein. Anfangs werden die zur Verfügung stehenden Daten beschrieben und wie diese für *Machine Learning* vorverarbeitet werden. Danach wird der erste Versuch gezeigt, wie mit ML und den Daten eine Fahreridentifizierung durchgeführt werden kann. Für jegliche Programmieranwendung wurde *Python* aufgrund der zahlreichen Module und des vorhandenen *Know-Hows* verwendet.

3.1. Trainingsdaten

Die Daten, welche für die Masterarbeit vorliegen, wurden dankenswerterweise von einem Kollegen bereitgestellt, der diese im Zuge einer Studie über einen längeren Zeitraum aufgezeichnet hat. Dabei handelt es sich um CAN-Nachrichten, aufgenommen in neun Fahrzeugen (VW Golf 7, keine näheren Informationen) während des Fahrens. Es wurde sichergestellt, dass jeweils nur eine Person hinter dem Lenkrad gesessen ist. Ein Boardcomputer mit Internetkonnektivität (ALEN, siehe 5.2) im Fahrerraum ist für die Aufzeichnung verwendet worden. Das *Embedded-Device* ist über zwei CAN-Bus Schnittstellen mit der Fahrwerks-Linie (F-CAN) und der Komfort-Linie (K-CAN) verbunden und kann somit alle Nachrichten in fast-Echtzeit vom Bus mitlesen. Dabei ist aber sichergestellt, dass es nicht möglich ist, auch Nachrichten senden zu können, da dies bei einer Fehlfunktion zu einem Unfall führen und Insassen gefährden könnte. Die Nachrichten werden nach dem Empfangen mit einem synchronisierten Zeitstempel, aufgelöst in Nanosekunden, versehen und in eine MDF Datei mit den Metadaten von der entsprechenden DBC-Datenbank gespeichert. Eine Datei enthält jeweils Messdaten über eine Dauer von einer Minute. Danach wurde die Datei temporär auf einer Festplatte gespeichert und auf einen File-Server über eine LTE-Funkverbindung hochgeladen. Um sie über alle Fahrzeuge hinweg eindeutig zuordnen zu können, wurde eine Fahrzeugidentifikationsnummer dem Dateinamen angehängt.

In einer Datei sind 46 verschiedene Signale der beiden CAN-Busse enthalten. Tablle 3.1 listet die meisten davon. Jene, die sich vorrausichtlich nicht für die Analyse eignen, wie z.B. die Innentemperatur oder der Status des Standlichts, sind ausgenommen. Für die komplette Liste siehe Anhang. Pro Datei sind durchschnittlich 100000 Messwerte vorhanden, daraus ergeben sich bei einer Anzahl von 10359 Dateien über eine Milliarde Messpunkte. Die folgende Liste gibt einen Einblick über die vorhandenen Daten.

- Anzahl Fahrer: 9
- Datengröße (MDF): 4.9 GB
- ⌀ Fahrtenanzahl: 46
- Fahrtenanzahl insgesamt: 421
- ⌀ Fahrzeit pro Fahrer pro Fahrt: 23 Minuten
- ⌀ Fahrzeit pro Fahrer: 18 Stunden
- Fahrzeit insgesamt: 165 Stunden
- ⌀ Gefahrene Kilometer: 1046

- Gefahrene Kilometer insgesamt: 9411
- Zeitraum: 13.07.2019 - 21.08.2019
- Geographischer Bereich: Stuttgart, Deutschland

Die Fahrrouten sind nicht speziell beschränkt, sondern sind von Fahrzeug zu Fahrzeug unterschiedlich, was die Fahrtdauer, Tageszeit und Route betrifft. Dadurch sind die Messdaten in sehr realen Bedingungen aufgenommen worden.

Signalname	CAN-Bus	Beschreibung	Einheit	Wertbeschr.
LWI_Lenkradwinkel	F-CAN	Lenkradwinkel	Grad	-
LWI_Lenkradw_Geschw	F-CAN	Lenkradwinkel-Geschwindigkeit	Winkelsekunde	-
ESP_Fahrer_bremst	F-CAN	Bremspedal betätigt	Boolean	0: nicht betätigt 1: betätigt
ESP_Bremsdruck	F-CAN	Bremsdruck	Bar	-
MO_Fahrpedalrohwert_01	F-CAN	Fahrpedalposition	-	-
MO_Kuppl_schalter	F-CAN	Kupplung betätigt	Boolean	0: nicht betätigt 1: betätigt
MO_Drehzahl_01	F-CAN	Motordrehzahl	rpm	-
KBI_Tankfuellstand_Prozent	F-CAN	Tankfüllstand	%	-
KBI_Kilometerstand	F-CAN	Kilometerstand	km	-
MO_Gangposition	F-CAN	Gangposition	-	1: 1. Gang oder Rückwärtsgang 2-6: 2. - 6. Gang
ESP_VL_Radgeschw_02	F-CAN	Geschwindigkeit Rad links vorne	km/h	-
ESP_VR_Radgeschw_02	F-CAN	Geschwindigkeit Rad rechts vornen	km/h	-
ESP_HL_Radgeschw_02	F-CAN	Geschwindigkeit Rad links hinten	km/h	-
ESP_HR_Radgeschw_02	F-CAN	Geschwindigkeit Rad rechts hinten	km/h	-
ESP_v_Signal	F-CAN	Durchschnittliche Radgeschwindigkeit	km/h	-
ESP_HL_Fahrtrichtung	F-CAN	Fahrtrichtung Rad links hinten	-	0: Vorwärts 1: Rückwärts
ESP_HL_Fahrtrichtung	F-CAN	Fahrtrichtung Rad links hinten	-	
ESP_Laengsbeschl	F-CAN	Längsbeschleunigung	m/s^2	-
ESP_Querbeschleunigung	F-CAN	Querbeschleunigung	m/s^2	-
ESP_Gierrate	F-CAN	Gierrate	Grad pro Sekunde	-
Wischer_vorne_aktiv	K-CAN	Wischer vorne aktiv	Boolean	0: nicht aktiv 1: aktiv
BH_Blinker_li	K-CAN	Blinker links aktiv	Boolean	
BH_Blinker_re	K-CAN	Blinker rechts aktiv	Boolean	

Tabelle 3.1.: Signalbeschreibung

Je nach Steuergerät und Nachrichtentyp ist die Signalhäufigkeit unterschiedlich. Zum Beispiel sendet das Lenkrad- und Motorsteuergerät mit der gleichen Frequenz – nämlich 100-mal in der Sekunde. Andere jedoch, wie das Getriebesteuergerät, nur mit einer Frequenz von 10Hz. Abbildung 3.1 zeigt die Häufigkeit einiger Signale. Alle nicht-gelisteten haben entweder eine Frequenz von 10, 50 oder 100Hz.

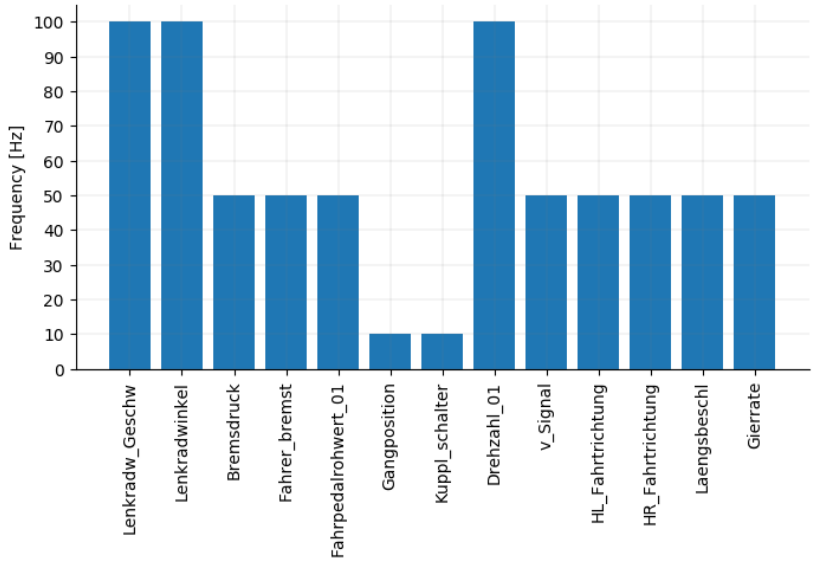


Abbildung 3.1.: Signalhäufigkeiten

3.2. Datenvorbearbeitung

Werden alle Signale eliminiert, welche nicht unmittelbar vom Fahrer beeinflusst werden und so auch keine Individualität aufweisen, ergibt sich eine Anzahl von 14 verschiedenen Signalen. Das Einschalten des Blinkers oder des Fernlichtes lässt nicht direkt auf den Lenker schließen, da es vielmehr von der Umgebung - Kreuzung oder Einbruch der Dunkelheit - abhängig ist. Die Wahl des Ganges in einer Kurve jedoch schon, siehe Kapitel 1.1. Diese 13 Signale (später auch als *Features* bezeichnet) sind:

Bremsdruck, *Gangposition*, *Geschwindigkeit*, *Motordrehzahl*, *Fahrpedalrohwert*, *Längsbeschleunigung*, *Querbeschleunigung*, *Lenkradwinkel*, *Gierrate*, *Bremspedal betätigt*, *Lenkradgeschwindigkeit*, *Kupplung betätigt*, *Fahrtrichtung*

Für jede *Machine Learning* Anwendung ist es ausschlaggebend, dass die Daten durchgängig strukturiert und einheitlich sind. Das Ziel ist also, Datensätze zu bekommen, wo jedes Signal einen Wert hat und einer Fahrerin zugeordnet ist. Da sich die Häufigkeit jedoch unterscheidet, müssen die Signale angeglichen werden, um so eine gemeinsame Frequenz zu bekommen. Laut dem Paper von M. Enev et al. liegt der Wert mit dem besten Endresultat bei 0.5Hz.[1] Aus allen Signalpunkten wurde daher der Durchschnitt über zwei Sekunden berechnet. Dadurch gehen aber auch Informationen verloren, wie zum Beispiel eine kurze Beschleunigung nach einer Kurve. Um dies zu erhalten, wird auch der Maximal- sowie der Minimalwert mitgenommen. Weiters kommt noch der Median und die Standardabweichung hinzu. Daraus ergibt sich ein Datenset aus 65 Signalen und die Fahreridentifikation. Abbildung 3.2 zeigt einen Auszug der Daten.

Für eine einfachere Weiterverarbeitung und etwaige Manipulationen müssen die Daten zwischengespeichert werden. Hierfür bietet sich das Format *Hierarchical Data Format version 5* (HDF5) an, welches der de facto Standard für die Speicherung von großen Datensets in *Python* ist. Es bietet sowohl ein schnelles Einlesen und Schreiben von Daten, als auch die Anwendung von Operationen auf ganze Datenreihen, wie beispielsweise die Berechnung des Durchschnitts oder die Bestimmung der Standardabweichung.[Col13]

[1] [ETKK16]

LWI_Lenkradw_Geschw_mean	LWI_Lenkradwinkel_min	ESP_Bremsdruck_median	ESP_Fahrer_bremst_mean	MO_Drehzahl_01_std	...	ESP_Laengsbeschl_max	MO_Fahrpedalrohwert_01_std	class
105.849401	12.818577	3.300599	0.906077	1298.212524	...	0.302663	0.000000	0
2.605031	15.639145	4.291138	1.000000	947.661204	...	0.251508	0.000000	0
0.000000	15.210978	0.315680	0.809218	329.509328	...	0.301648	0.000000	0
4.658004	16.779374	0.173465	0.552288	1003.716790	...	0.242073	0.000000	0
122.792461	143.876343	0.896582	0.167568	1109.765179	...	0.883917	8.461988	0
55.320087	26.565342	1.086463	0.104461	988.248651	...	0.241860	0.000000	0
228.393121	328.281093	1.169506	0.097720	1090.197831	...	0.359767	6.653113	0
142.033783	190.538954	1.614181	0.508143	1033.419805	...	0.496814	3.076769	0
35.391337	7.147907	2.626311	1.000000	984.804091	...	0.553433	0.000000	0
0.000000	7.000000	18.207242	1.000000	982.115105	...	0.267885	0.000000	0

Abbildung 3.2.: Auszug aus Datenset

3.3. Scikit-Learn

Die Programmiersprache *Python* erfreut sich in den letzten Jahren an einer immer mehr werdenden Popularität. Dank der einfachen Syntax und einer Vielzahl an wissenschaftlichen Bibliotheken ist sie auch besonders für Datenanalyse interessant. Das Modul *Scikit-Learn* [PVG+11] nützt diese Umgebung und bietet Implementierungen der aktuellsten *Machine Learning* Algorithmen an. Es ist daher eine Antwort auf den wachsenden Bedarf an statistischer Datenanalyse vor allem in nicht wissenschaftlichen Bereichen wie der Software- und Webentwicklung. *Scikit-Learn* gilt als performant und intuitiv, da sie grundlegende Strukturen und Module von *Python* verwendet. Darunter zählen *Numpy*, *Scipy* und *Cython*. Des Weiteren werden auch sehr wenig andere externe Abhängigkeiten benötigt, was zusätzliche Performance bringt.

3.4. Erster Versuch

Als erster Schritt wurde der *Random Forest* Algorithmus mit Standardwerten (siehe weiter unten) auf Messdaten angewendet, um zu validieren, dass eine Identifikation überhaupt mit den vorhanden Daten möglich ist. Hierfür wurden aus den gesamten Daten zufällige 20 Minuten Fahrtzeit pro Fahrer extrahiert, wobei der jeweils erste Datensatz der Beginn einer Fahrt ist. Laut Miro Enev et al. [ETKK16] werden zwischen zehn und 15 Minuten Trainingszeit benötigt, um eine Identifikationsgenauigkeit von über 90% zu erreichen. Da die Forscher mehr Daten zur Verfügung hatten, wurde die Zeit um fünf Minuten verlängert. Die neun (9 Fahrer) Datensets wurden dann zu einem zusammengefügt, vermischt und in die *Features* beziehungsweise die Fahrerkennung (*class*) aufgeteilt. Daraus sind zufällige sechs Minuten (30%) entfernt worden, welche später zum Testen des *Models* verwendet werden. Das Listing 3.1 zeigt den Programmcode in *Python*.

```
1  import os
2  import sys
3  import pandas as pd
4  from sklearn.utils import shuffle
5  from sklearn.model_selection import train_test_split
6  from sklearn.ensemble import RandomForestClassifier
7  from sklearn import metrics
8
9  hdf5_input = sys.argv[1]
10 signals = ['can0_LWI_Lenkradw_Geschw_mean', 'can0_LWI_Lenkradwinkel_mean', ...]
11 frames = []
12 for file in os.listdir(hdf5_input):
13     data = pd.read_hdf(os.path.join(hdf5_input, file))
14     frames.append(data)
15
16 result = pd.concat(frames, sort=False)
17 result = shuffle(result)
18 X = result[signals]
```

```
19 Y = result['class']
20 X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3)
21
22 clf = RandomForestClassifier()
23 clf.fit(X_train, Y_train)
24 Y_pred = clf.predict(X_test)
25 accuracy = metrics.accuracy_score(Y_test, Y_pred)
26
27 print('Accuracy:', accuracy)
```

Listing 3.1: Erster Versuch mit *Random Forest*

Dabei ist eine Identifikationsrate von 85% Prozent erzielt worden. Das bedeutet, dass das ML-*Model* über 150 Datenreihen aus 180 pro Fahrer ($6[min]*60*0,5Hz = 180$) den tatsächlichen Fahrern zuordnen konnte und bestätigt eine Identifizierung mit den vorhanden Daten. Bei einem erneuten Versuch war die Wahrscheinlichkeit fast 91% und beim dritten unter 83%, mit jeweils anderen zufälligen Daten. Die Ergebnisse von weiteren Durchläufen sind in der Abbildung 3.3 ersichtlich. Die erste Schlussfolgerung daraus ist, dass eine Identifikation mit den Daten und der Methode möglich ist. Weiters hat die Auswahl der Daten einen erheblichen Einfluss auf das Ergebnis. So war die geringste Wahrscheinlichkeit 78,66% und die höchste 92,53%. Im Mittel sind 86,4% erzielt worden.

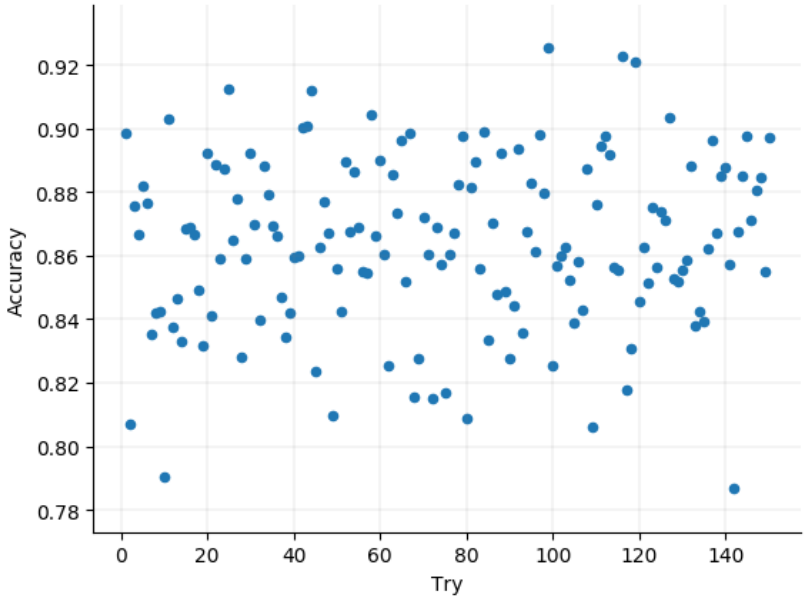


Abbildung 3.3.: *Random Forest* Versuche

4. Optimierung

Das Kapitel *Optimierung* zeigt wie die Identifikation mit dem verwendeten Algorithmus verbessert werden kann. Es wird einerseits versucht, die Daten zu optimieren, wie zum Beispiel die Anzahl der *Features* oder die Dauer der Trainingszeit. Andererseits wird auch die Methode durch geschickte Parametrisierung verbessert.

4.1. Random Forest Parameter-Optimierung

Die *Random Forest* Implementierung von *Scikit-Learn* bietet mehrere Parameter, welche die Trefferquote beeinflussen. Tabelle 4.1 listet diese mit ihren Standardwerten und einer kurzen Beschreibung.

Tabelle 4.1.: *Random Forest* Parameter [BLB+13]

Parameter	Standardwert	Beschreibung
criterion	gini	Funktion zur Qualitätsmessung bei der Teilung. Mögliche Werte: gini: Gini-impurity entropy: Informationsgehalt
min_samples_split	2	Minimale Anzahl an Daten in einem Blatt, bevor es aufgeteilt wird
min_samples_leaf	1	Minimale Anzahl an Daten für ein Blatt
max_depth	None	Baumtiefe
n_estimators	100	Anzahl an *Trees* im *Forest*

Wie bereits im vorigen Abschnitt gezeigt, steuern die verwendeten Trainings- und Testdaten maßgeblich das *Model*. Die Wahl der Parameter spielt dennoch eine Rolle. Um die Genauigkeit zu verbessern, können die Werte dahingehend angepasst werden. Damit es zu keinen Verfälschungen durch unterschiedliche Daten kommt, wurde jeder Optimierungsversuch sowohl mit denselben Trainings- als auch mit denselben Testdaten durchgeführt. Die Basis bildet daher ein 20-minütiges Datenset pro Fahrer, dessen Genauigkeit mit Standardparameter bei 91,28% liegt. Zuerst ist jeder Parameter für sich verbessert und die Standardwerte der jeweils anderen herangezogen worden. Es wird nämlich angenommen, dass die Standardwerte bereits ein sehr gutes Ergebnis liefern. Jedoch auch, dass sich die Parameter gegenseitig beeinflussen, sodass die optimalsten Werte nur in Abhängigkeit der anderen bestimmt werden können.

4.1.1. criterion

Der erste Wert, den es zu verbessern gilt, ist *criterion*. Er legt die Funktion fest, wie die Aufteilung bei einem Knoten durchgeführt wird und kann somit entweder *gini* oder *entropy* sein. Abbildung 4.1 zeigt die Auswirkungen. Obwohl laut [RRC19] *entropy* für Kategorien besser geeignet ist, kann erkannt werden, dass die durchschnittliche Genauigkeit bei beiden etwa gleich ist. Jedoch fällt die Durchlaufzeit bei *gini* (0.77 Sekunden zu 2.4) wesentlich

kürzer aus. Für alle weiteren Optimierungen wurde daher auf Grund des besseren Zeitfaktors nur noch *gini* verwendet.

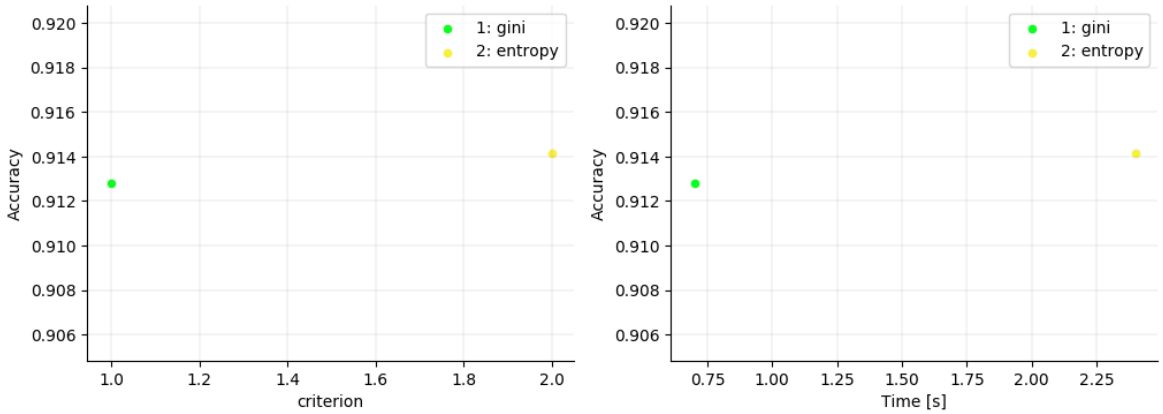


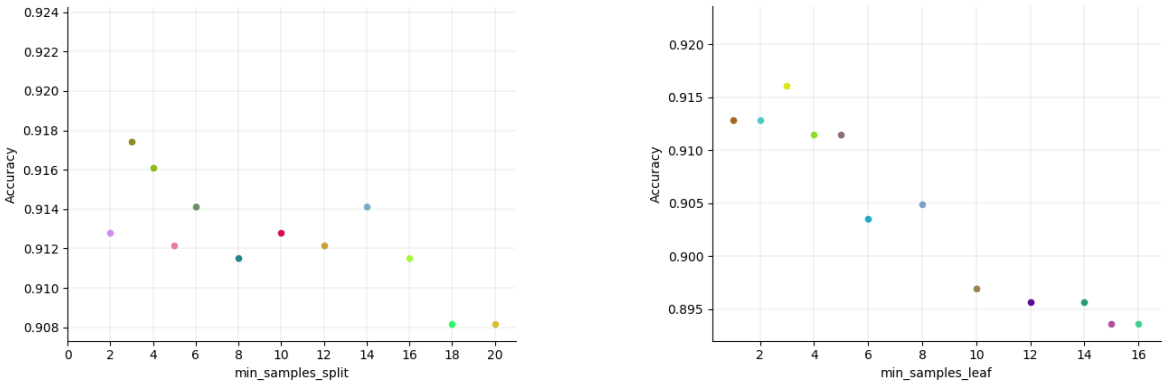
Abbildung 4.1.: *Random Forest* Parameteranalyse *criterion*

4.1.2. min_samples_split

Als zweites wird der Parameter *min_samples_split* analysiert. Dieser bestimmt die mindest Anzahl an Datenpunkten in einem Knoten, welche für das Aufteilen notwendig sind. Auch hier wurden die Standardwerte für die anderen verwendet. Aus der Abbildung 4.2a geht hervor, dass der Parameter das Ergebnis um fast 1 Prozent beeinflusst (90.8 - 91.8%), wobei die höchste Genauigkeit mit dem Wert *3* erzielt wurde. Weiters ist zu erkennen, dass je höher der Parameter gewählt wird, desto geringer die Trefferquote ausfällt.

4.1.3. min_samples_leaf

Der nächste Parameter ist *min_samples_leaf* und definiert die minimale Anzahl an Datenpunkten in den *leaf nodes*. Wie die Abbildung 4.2b zeigt, nimmt auch hier die Genauigkeit mit ansteigenden Werten ab. Das Maximum liegt bei *3* mit 91.16%. Der Einfluss auf das ML-*Model* ist etwas höher, da das schlechteste erzielte Ergebnis fast bei 89% ist.



(a) *min_samples_split* (b) *min_samples_leaf*

Abbildung 4.2.: *Random Forest* Parameteranalyse

4.1.4. max_depth

max_depth beschreibt die Baumtiefe. Ist der Wert auf *undefined* (repräsentiert durch *0* in der Grafik) gesetzt, wird ein Baum soweit expandiert, bis in einem Blatt *min_samples_leaf* Datenpunkte erreicht werden. Das beste Resultat wird laut Grafik 4.3 bei einem Wert von *15* erzielt. Danach fällt es leicht ab und pendelt sich ein.

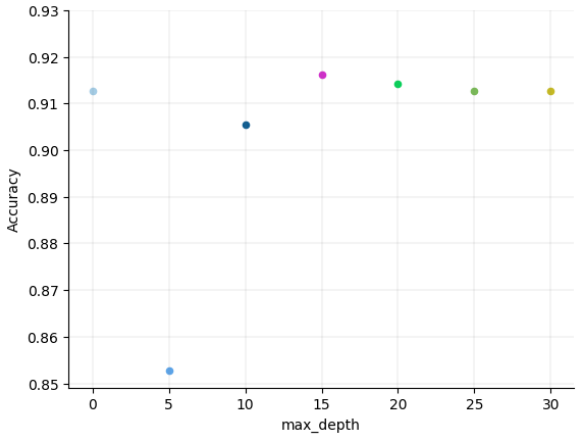


Abbildung 4.3.: *Random Forest* Parameteranalyse *max_depth*

4.1.5. n_estimators

Die Anzahl der Bäume in einem *Forest* werden durch den Parameter *n_estimators* spezifiziert. Laut Literatur [RRC19], steigt mit der Erhöhung der Anzahl auch die Genauigkeit des *Models*. Die Abbildung 4.4 bestätigt dies. Jedoch steigt auch die Trainings- beziehungsweise die Testzeit, was ebenfalls in der Grafik ersichtlich ist.

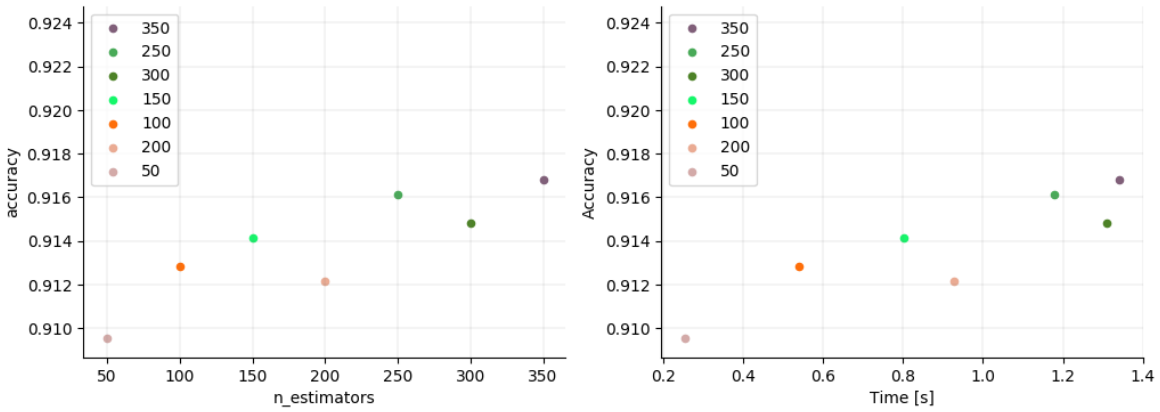


Abbildung 4.4.: *Random Forest* Parameteranalyse *n_estimators*

4.1.6. Verbesserte Parameter

Die Tabelle 4.2 zeigt unter anderem die Zusammenfassung der ersten Analyse. Wird nun das *Model* mit diesen Werten auf die Daten angewendet, ergibt sich eine Genauigkeit von 91.41%. Das heißt, es konnte lediglich eine Verbesserung um 0.13% gegenüber den Standardwerten

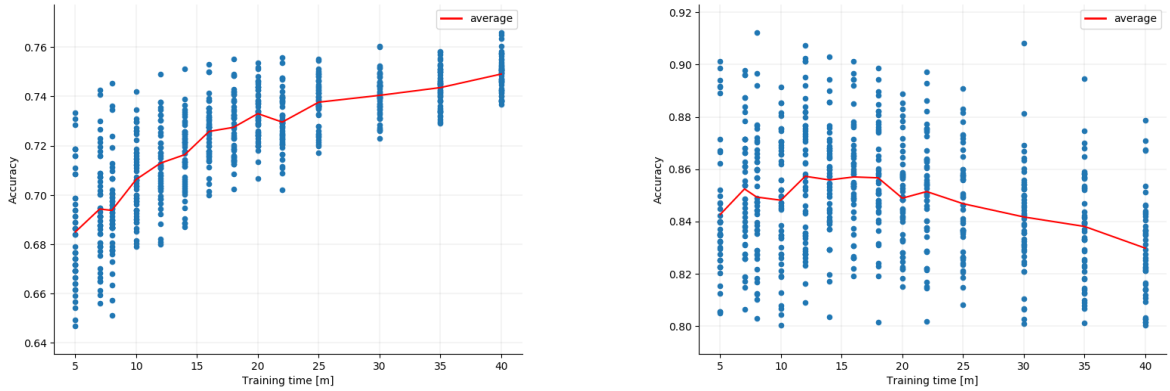
erreicht werden. Dies bestätigt die erste Annahme, dass die Defaultparameter bereits sehr gute Ergebnisse liefern. Weiters kann die Genauigkeit noch verbessert werden, in dem die beste Kombination aller Werte der Parameter gefunden wird. Hierfür müssen 16128 Tests mit dem *Model* durchgeführt werden, was einen hohen Zeit- und Rechenaufwand bedeutet. Dabei ist eine größtmögliche Genauigkeit von 92.21% herausgekommen, was einer Verbesserung von fast einem Prozentpunkt mit den Daten entspricht. Die Parameterwerte sind ebenfalls in der unten angeführten Tabelle ersichtlich. Gegenüber der ersten Analyse haben sich nicht mehr als zwei Parameter (*min_samples_split* und *n_estimators*) verändert und das auch nur marginal. Für jede fortlaufenden Berechnungen wird diese Kombination verwendet.

Tabelle 4.2.: Verbesserte *Random Forest* Parameter

Parameter	Standardwert	Verbesserte Wert (mit Default)	Beste Kombination
criterion	gini	gini	gini
min_samples_split	2	3	3
min_samples_leaf	1	3	1
max_depth	None	15	None
n_estimators	100	350	300

4.2. Trainingszeit

Die Trainingszeit eines ML-*Models* wirkt sich unmittelbar auf die Genauigkeit der Ergebnisse aus. Je mehr Daten anfangs eingespeist werden, desto mehr kann das *Model* lernen und ein genaueres Ergebnis bei neuen Daten liefern. Jedoch steigt auch die Wahrscheinlichkeit, dass die Anzahl an Datenpunkten höher wird, welche ungünstig für diese Anwendung sind. Die Genauigkeit sinkt wiederum dadurch. Laut [ETKK16] und [EBG18] wird bei etwa zehn Minuten Trainingszeit eine Trefferquote von 90% erzielt. Dies entspricht 300 Datenpunkte pro Fahrer. Die Autoren haben auch gezeigt, dass der allgemeine Fall - mehr Lernen für bessere Ergebnisse - hier nicht zwangsweise zutrifft. Ihre Modelle zeigen nach über 30 Minuten keine Verbesserungen mehr. Da aber auch nicht genau hervorgeht, mit welchen Daten sie zu diesen Erkenntnissen gekommen sind, wird dies versucht nachzustellen und zu validieren. Einerseits mit Daten, welche komplett zufällig aus dem ganzen Datenset mit allen Fahrern gewählt werden und andererseits mit Daten bei den ersten x Minuten nach Fahrtbeginn. Dabei ist aber immer die Anzahl an Datenpunkten pro Fahrerin ident. Um eine genauere Aussagekraft der Analysen zu bekommen, wurden die Tests jeweils 50 mal mit unterschiedlichen Daten durchgeführt. Die Ergebnisse sind in den Abbildungen 4.5a und 4.5b ersichtlich. Sie zeigen, dass zwischen zwölf und 18 Minuten an Trainingszeit mit Daten ab einem Fahrtbeginn die beste Trefferquote erzielt werden kann. Ab da an sinkt sie kontinuierlich. Dies deckt sich mit den Aussagen von [ETKK16] und [EBG18], wenngleich die Genauigkeit nicht konsequent erzielt werden konnte. Werden jedoch rein zufällige Daten für das Modell verwendet, ist erstens die durchschnittliche Genauigkeit (rote Linie) viel geringer, zweitens steigt sie mit der Anzahl der Daten und drittens gibt es weniger Ausreißer nach oben und nach unten je mehr Daten verwendet werden. Das Maximum liegt jedoch auch nur bei knapp über 76% mit 40 Minuten an Trainingsdaten pro Fahrer. Es ist daher anzunehmen, dass die erwähnten Autoren nicht mit zufälligen Daten aus allen CAN-Signalen gearbeitet haben, sondern mit einem Subset.



(a) Analyse mit zufälligen Daten

(b) Analyse mit Fahrtbeginn

Abbildung 4.5.: Analysen der Trainingszeit

4.3. Feature Optimierung

Eine weitere Möglichkeit das Modell zu optimieren ist, *Features*, welche keine Auswirkung auf die Klassifizierung haben, zu entfernen. Dies spart Rechenleistung, -zeit und könnte unter Umständen auch das Endresultat verbessern. Hierfür gibt es mehrere Möglichkeiten.

4.3.1. Feature Korrelationen

Eine davon beschäftigt sich mit der Korrelation von *Features* [Cho10]. Dabei wird analysiert, ob es zwischen Merkmalen einen Zusammenhang gibt. Es kann entweder der *Pearson-* oder der *Spearman*-Koeffizient herangezogen werden. Beide können Werte zwischen *-1* und *+1* annehmen. Ersterer kommt bei Daten zum Einsatz, welche sehr stark linear und kontinuierlich sind. Der zweite ist besser, wenn sich die *Features* zusammen verändern, aber nicht unbedingt um den gleichen Faktor. Gibt es streng zusammenhängende Merkmale, können diese (bis auf eines) eliminiert werden. Dadurch reduzieren sich die zu verarbeitenden Daten und das spart Rechenaufwand. Bei den vorhandenen Daten ist anzunehmen, dass die verschiedenen *Features* der einzelnen Signale - Minimal-, Maximal-, Durchschnittswert, Standardabweichung und Median - einen hohen Korrelationswert zueinander haben. Die Abbildung 4.6 belegt dies mithilfe des *Spearman*-Koeffizienten (*+1*, dunkelblau). Daraus ist weiters zu erkennen, dass es einen offensichtlichen Zusammenhang zwischen dem Lenkradwinkel und den verschiedenen Kräften gibt, welche auf das Auto einwirken. Dies gilt ebenso für den Fahrpedalrohwert. Ansonsten können keine zusätzlichen Korrelationen ausgemacht werden.

4.3.2. Feature Importance

Bevor jedoch einfach die stark korrelierenden *Features* entfernt werden, kann noch eine weitere Analyse durchgeführt werden. Hierbei zeigt sich ein zusätzlicher Vorteil des Algorithmus *Random Forest*. Wie bereits in den Grundlagen erklärt, wird nämlich mithilfe von *Gini Impurity* das wahrscheinlich bestmögliche *Feature* gefunden, um die Daten bei einem Knoten aufzuteilen. Dadurch kann bereits während der Erstellung der verschiedenen *Decision Trees* festegellt werden, welchen Einfluss ein bestimmtes Datenattribut hat. Mit der Implementierung von *Scikit-Learn* können genau diese Werte dargestellt werden. Aus der Abbildung 4.7a ist zu entnehmen, dass der Bremsdruck das stärkste Merkmal ist, um eine Fahrerin zu identifizieren. Dies deckt sich auch mit den Ergebnissen von Gahr et al., welche ausschließlich dieses Signal

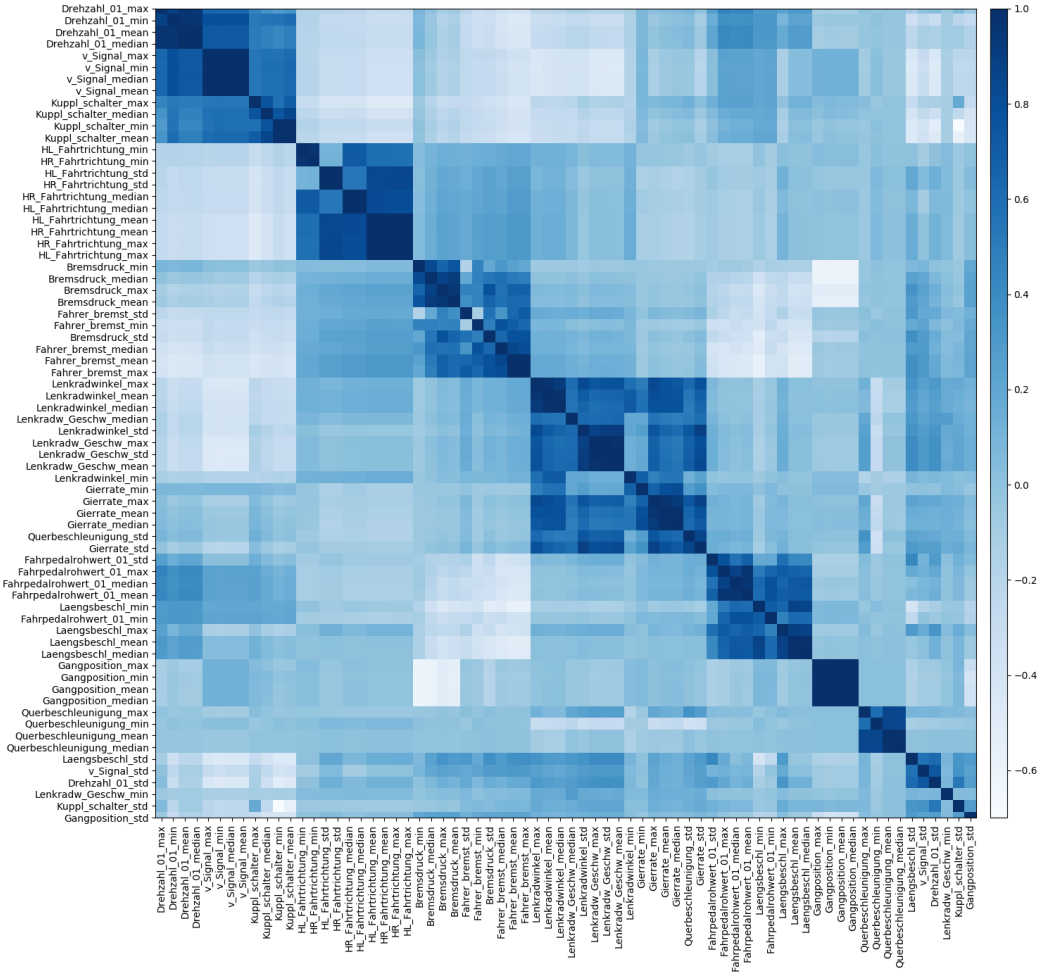


Abbildung 4.6.: *Feature* Korrelation

zur Identifizierung verwendet haben. Verwunderlich ist jedoch, dass die reine Gangposition die zweitgrößte Rolle einnimmt. Stellt man die Werte aller Fahrer über die gesamte Fahrzeit dar (siehe Abbildung 4.8), kann erkannt werden wieso. Für fünf Fahrer sind nämlich keine korrekten Werte vorhanden. Sie zeigen durchgehend den Gang 14. Dies kann mehrere Gründe haben. Entweder wurde das Aufzeichnungsgerät falsch konfiguriert, das falsche Signal in diesen Autos abgegriffen oder es war schlichtweg nicht vorhanden und 14 ist ein Standardwert. Auch bei den anderen Fahrzeugdaten dürfte ein Fehler unterlaufen sein, da zwischendurch der Gang 13 angezeigt wird. Aus diesem Grund ist für das ML-*Model* die Gangposition so entscheidend. Soll ein neuer Datenpunkt klassifiziert werden, welcher den 14ten Gang enthält, kommen nur noch fünf Fahrer in Frage und dies nur Aufgrund des einen Signals. Es hat jedoch nichts mit dem Individuum hinter dem Lenkrad und dem Fahrverhalten zu tun. Infolge dessen werden dadurch die Ergebnisse des Modells verfälscht und das *Feature* muss daher aus allen vorliegenden Daten entfernt werden. Da es wie erklärt die Trefferquote erhöht hat, geht damit auch ein Performance-Verlust einher. Im vorigen Abschnitt wurde gezeigt, dass mit den Daten eine maximale Genauigkeit von 92% (durchschnittlich 86%) erreicht werden konnte. Nach mehreren erneuten Durchläufen ohne der Gangposition konnte nun lediglich 80% im Mittel und maximal 85% erzielt werden. Das Modell hat sich demnach um mehr als 5% verschlechtert, klassifiziert jedoch die Fahrer basierend auf korrekten Fahrzeugdaten.

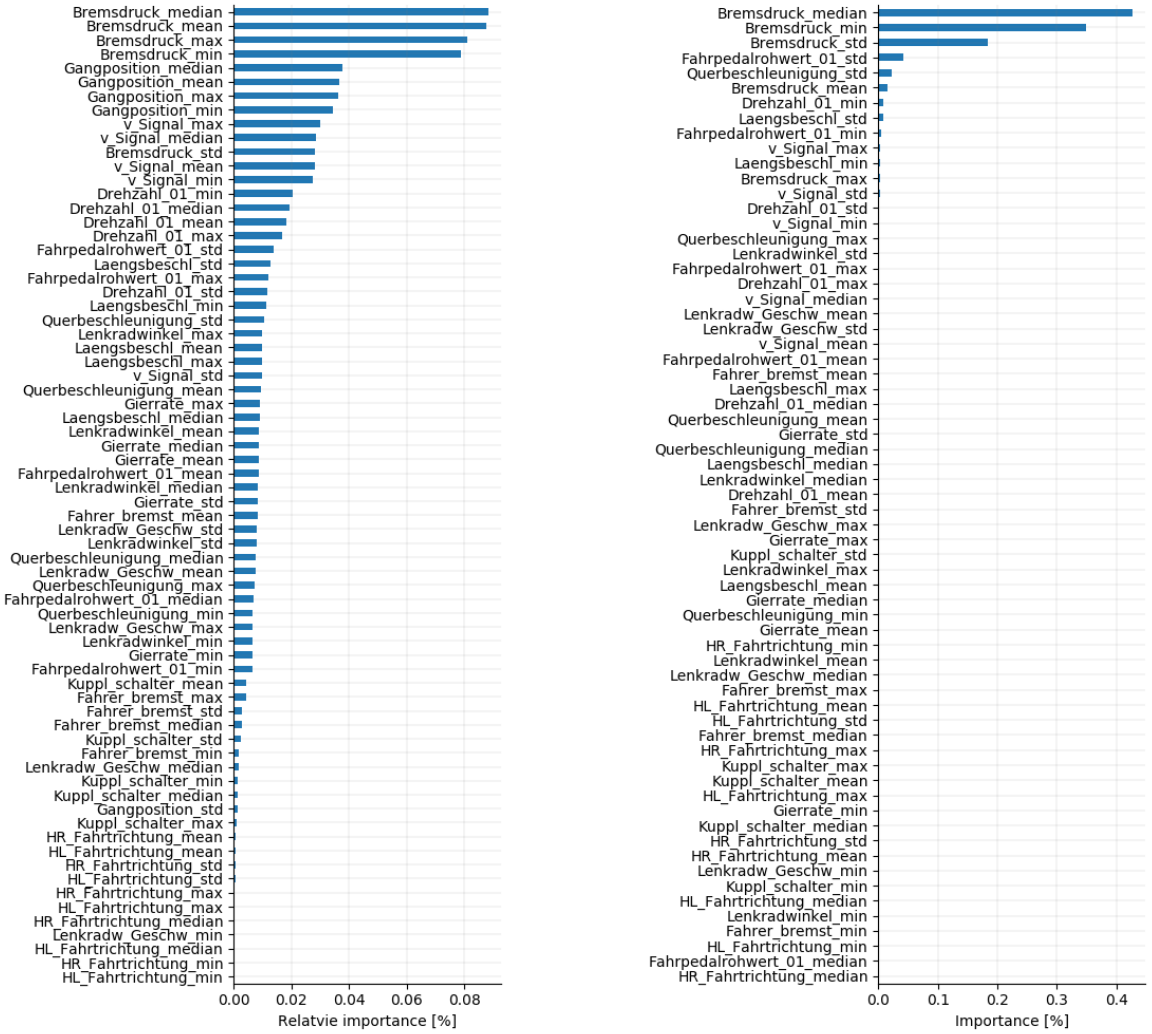
Des Weiteren zeigt die Grafik 4.7a, dass die Geschwindigkeit (*v_Signal*) und die Motordrehzahl ebenso einen großen Einfluss haben, obwohl eher das Gegenteil anzunehmen ist. Warum die Signale so weit oben angeführt werden hat vermutlich damit zu tun, dass die *Feature*-Auswahl durch das *Gini criterion* verzerrt ist. 2007 haben die Forscher Strobl et al. [SBZH07] folgendes herausgefunden: Ihre Versuche zeigen, dass *Features* mit mehr potenziellen Aufteilungspunkten ein gutes Kriterium ergeben. Da diese Anzahl exponentiell mit der Anzahl der verschiedenen möglichen Werte steigt, werden solche Merkmale gegenüber welchen mit einer geringeren Varianz von *Decision Trees* eher bevorzugt. Dafür haben sie ein zusätzliches Attribut mit reinen Zufallswerten in die Daten eingefügt. Bei der anschließenden *Feature Importance*-Analyse haben sich diese Daten als besser gegenüber manch anderer echter Daten herausgestellt. Die Zufallswerte hatten dabei eine sehr große Varianz. Für die zwei genannten Signale trifft dies ebenso zu.

4.3.3. Permutation Importance

Um diese Verzerrung durch das *Gini criterion* zu umgehen, haben die oben genannten Forscher eine alternative Methode zur Messung der *Feature Importance* vorgestellt [SBZH07]. Die Idee dahinter ist, herauszufinden, wie sich die Genauigkeit verändert, wenn ein bestimmtes *Feature* nicht mehr existiert. So könnte zum Beispiel bei jedem erneuten Training des Modells eine Datenreihe weggelassen werden. Dies hat aber den Nachteil, dass es sehr Rechenaufwändig ist und nicht unbedingt wiederspiegelt, was in dem gesamten Datenset relevant ist. Einfach das bestimmte *Feature* nur beim Testdurchlauf wegzulassen ist zwar billiger, funktioniert aber nicht, weil die trainierten Entscheidungsbäume alle Datenpunkte einer Reihe erwarten. Anstatt dessen können die echten Werte durch zufälliges Rauschen mit der gleichen Datenverteilung ersetzt werden. Das geht am leichtesten, wenn die Werte des gleichen *Features* permutiert werden. Das Verfahren heißt daher *Permutation Importance* und besteht aus folgenden drei Schritten, wobei Schritt zwei und drei sukzessive wiederholt werden:

Schritt 1 Berechne die Genauigkeit des Modells mit allen *Features*.

Schritt 2 Wähle ein *Feature* aus, permutiere die Werte und berechne die Genauigkeit des Modells erneut.



(a) *Scikit-Learns Feature Importance*

(b) *Permutation Feature Importance*

Abbildung 4.7.: *Feature Importance*

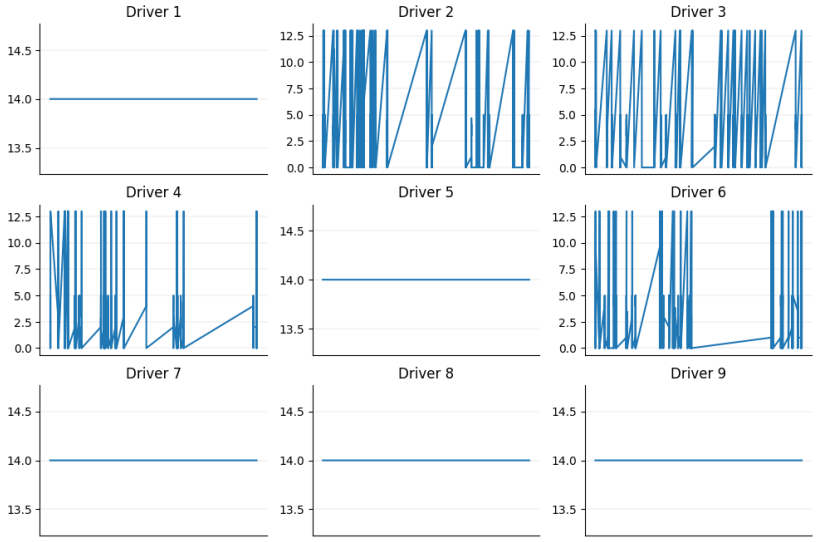


Abbildung 4.8.: Gangposition

Schritt 3 Die Differenz zwischen dem Wert aus Schritt 1 und Schritt 2 ergibt die Wichtigkeit des *Features*.

Mittlerweile ist es als Funktion auch schon in dem *Scikit-Learn* Modul zu finden und kann auf die Daten angewendet werden. Abbildung 4.7b zeigt die Ergebnisse daraus. Hier ist zu erkennen, dass der Bremsdruck neuerlich das stärkste Kriterium ist. Das zweit bestbewertete Signal ist der Fahrpedalrohwert. Weiters bestätigt diese Messung die Annahme, dass es bei der ersten Methode eine Verfälschung gegeben hat, da die Geschwindigkeit und die Motordrehzahl weiter unten angeführt werden. Generell haben die meisten Signale eine geringere Auswirkung auf die Genauigkeit des Modells als anfangs gedacht. So sind beispielsweise der Lenkradwinkel, Lenkradbeschleunigung und die Kupplung sehr weit unten in der Tabelle. Das bedeutet aber nicht, dass diese Informationen gänzlich unwichtig für die Klassifizierung sind. Es heißt nur, dass sie für sich alleinstehend keine großen Auswirkungen haben, in Kombination hingegen vielleicht schon.

4.3.4. Feature Selection

In den letzten zwei Abschnitten wurde die Korrelation zwischen den *Features* gezeigt. Wenn die Eliminierung nach dieser Analyse geht, müssten nahezu $\frac{4}{5}$ der Merkmale wegfallen. Für die korrekte Auswahl kann die *Feature Importance* von *Scikit-Learn* herangezogen werden. Dabei wurden jedoch zwei Widersprüche festgestellt, zum einen die falschen Werte der Gangposition und zum anderen die Verfälschung durch das *Gini criterion*. Mithilfe der Methode *Permutation Importance* konnte dies bereinigt werden und einen möglichst unbeeinflussten Eindruck über die Wichtigkeit der *Features* bekommen. Mit diesen Ergebnissen kann die sogenannte *Feature Selection* durchgeführt werden, bei der die optimale Anzahl an Merkmalen in dem ML-*Model* bestimmt wird. Zu diesem Zweck kann der Algorithmus *Recursive Feature Elimination* (RFE) [GWBV02] verwendet werden. Als erster Schritt wird das Modell mit allen *Features* trainiert und die Genauigkeit bestimmt. Danach kommt die Methode *Permutation Importance* zum Einsatz, um die Wichtigkeit der Merkmale zu bestimmen. Anhand der resultierenden Reihung wird nach und nach das *Feature* mit der geringsten Auswirkung eliminiert, bis nur noch eines (in diesem Fall *Bremsdruck_median*) über bleibt. Bei jedem

Durchlauf erfolgt das erneute Antrainieren mit dem Subset der Merkmale und die Bestimmung der Genauigkeit. Der gesamte Vorgang wurde mit 100 verschiedenen Trainings- bzw. Testdaten durchgeführt. Werden die Resultate in einer Grafik visualisiert (siehe Abbildung 4.9), kann erkannt werden, wie viele *Features* für eine gewisse Trefferquote benötigt werden. So gibt es zwischen den ersten zehn Merkmale eine deutliche Steigerung und ab dem elften flacht sie ab. Ein sehr guter Wert kann mit 21 der verfügbaren 65 *Features* erzielt werden.

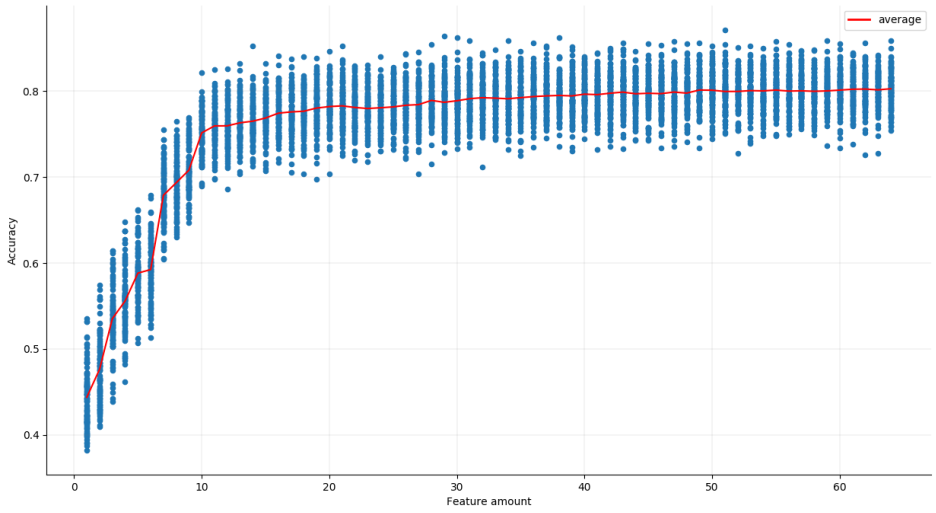


Abbildung 4.9.: *Feature* Auswahl

5. Integration ins Fahrzeug

In diesem Kapitel wird gezeigt, wie das System zur Erkennung des Lenkers prototypisch in ein Fahrzeug integriert werden kann. Die Anwendung soll insbesondere folgende Aufgabe erfüllen:

Aus einem Set von vier bekannten Fahrerprofilen soll die aktuell fahrende Person zugeordnet werden können.

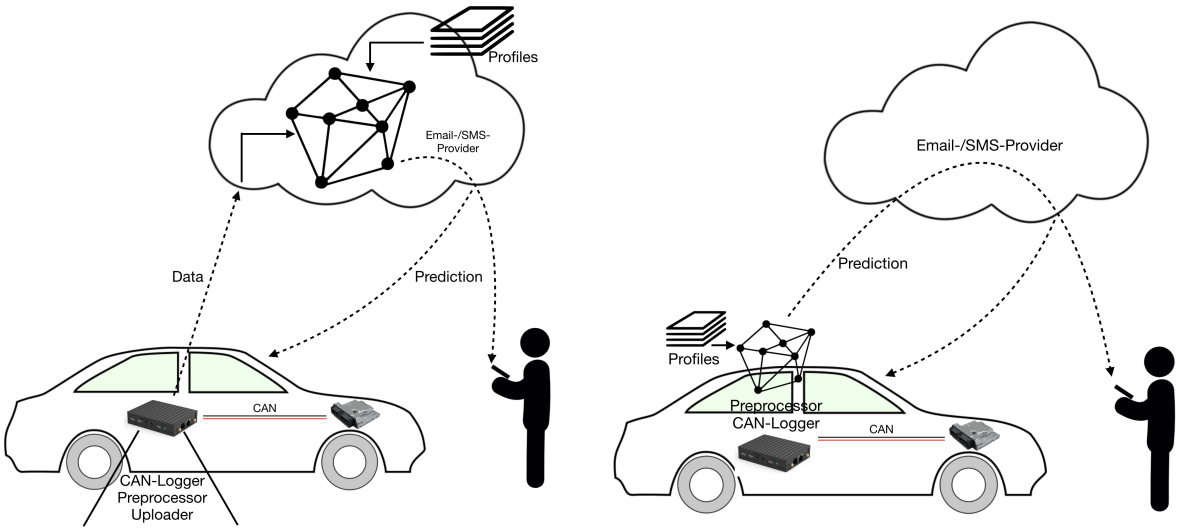
Laut Statistik Austria hat die durchschnittliche österreichische Familie 1,68 Kinder (Stand 2018) [Sta]. Aufgerundet besteht sie daher aus vier Personen und deshalb die Anzahl der Fahrerprofile.

Die Anforderung zielt speziell auf konkrete Anwendungsfälle, welche zum Teil bereits in der Einleitung erläutert wurden, ab. Mit den Erkenntnissen lassen sich Maßnahmen umsetzten, um beispielsweise mehr Komfort bieten zu können. Darunter fällt das Einstellen des bevorzugten Radiosenders oder das Anzeigen der letzten Ziele im Navigationssystem für eine schnellere Auswahl. Andere Möglichkeiten, wie diese Information genutzt werden kann, werden weiters noch im nächsten Kapitel diskutiert.

5.1. Systemarchitektur

Das Gesamtsystem kann mit zwei verschiedenen Architekturen umgesetzt werden. Bei beiden Varianten gibt es vier Komponenten. Ausgangspunkt sind die Steuergeräte, die in einem Fahrzeug über den CAN-Bus vernetzt sind. Eine zentrale Logikeinheit, die *Car Connectivity Unit* (CCU), zeichnet die CAN-Nachrichten auf und verarbeitet diese. Sie ist nicht Teil eines Serienfahrzeuges und muss nachgerüstet werden. Beim ersten Ansatz überträgt die CCU alle CAN-Daten zu einem *Cloud*-Service, wo sie von einem bereits trainierten *Machine Learning Model* klassifiziert werden. Aufgrund dessen liegen die Profile der autorisierten Lenker eines Fahrzeugs auch in der *Cloud*. Das Ergebnis um welche Fahrerin es sich handelt, wird entweder über einen Email- oder SMS-Provider zurück ins Auto oder zum Fahrzeughalter gesendet. Beim zweiten Ansatz wird alles auf der CCU durchgeführt. Sie trainiert ein Model mit den Fahrerprofilen, liest CAN-Nachrichten mit, führt die Klassifizierung durch und versendet das Ergebnis. Die Abbildungen 5.1a und 5.1b bilden die Ansätze ab.

Der Hauptunterschied bei beiden liegt darin, wo jegliche *Machine Learning* Anwendungen laufen. Dies zieht jedoch eine Reihe von Konsequenzen mit sich. In erster Linie wird dem System Komplexität genommen, wenn Komponenten wegfallen. In diesem Fall ist es das ML-*Cloud*-Service. Die CCU muss im zweiten Ansatz keine Verbindung in die *Cloud* aufbauen und Daten übertragen. Da es sich dabei mitunter um sensible Daten handelt, fallen zudem alle Anforderungen bezüglich CIA - *Confidentiality*, *Integrity*, *Authenticity* - weg. Der Kommunikationskanal müsste sonst gegen ein unerlaubtes Abgreifen der Daten verschlüsselt, gegen eine Verfälschung integritätsgeschützt und authentifiziert sein, damit potentielle Angreifer keine Daten einspeisen können. Die ersten beiden Kriterien können einfach mit dem Einsatz von *HTTPS* sichergestellt werden. Hierzu muss lediglich ein entsprechendes Zertifikat am Server installiert sein, welches für den Daten-Upload verwendet wird. Für die Gewährleistung der



(a) Architektur 1

(b) Architektur 2

Abbildung 5.1.: Mögliche Architekturen

Authentizität können zum Beispiel Client-Zertifikate eingesetzt werden. Dafür ist eine eigene *Public Key Infrastructure* (PKI) nötig, um die Zertifikate auf den CCUs zu verwalten.

Ein weiterer Nachteil des zweiten Ansatzes ist, dass Übertragungs- und Verbindungsabbrüche korrekt behandelt werden müssen. Dazu zählt die Verifizierung eines erfolgreichen Uploads, die Wiederaufnahme eines Abbruchs und die Sicherstellung, dass nichts mehrfach hochgeladen wird. Nicht nur die Kommunikation zwischen der *Connectivity Unit* und der *Cloud* erfordert Maßnahmen gegen *Security Threats*, sondern auch die *Cloud* an sich. Sie muss mit einer *Firewall* abgesichert, ein unerlaubter Austausch und Verfälschung der Profile verhindert werden. Wie bereits in den Grundlagen 2.3 *Edge Computing* erläutert, hat die Wahl der Architektur auch eine Auswirkung auf den Datenschutz. Bleiben die Daten nur offline am Gerät, muss deutlich weniger beachtet werden, als wenn sie in die *Cloud* hochgeladen werden.

Der zweite Ansatz hat aber auch Nachteile. So ist beispielsweise die Verteilung von Updates schwieriger, da diese auf jeder CCU eingespielt werden müssen und nicht nur zentral in der *Cloud*. Es könnte mitunter auch zu Performance-Problemen kommen, falls das *Embedded-Device* zu wenig Ressourcen für *Machine Learning* Anwendungen hat. Im Ansatz 1 kann durch eine einfache und schnelle Skalierung der CPUs/GPUs beziehungsweise des Arbeitsspeichers der IaaS-Provider die Leistung an die Anforderungen angepasst werden. Auch die Administration der Fahrerprofile wie zum Beispiel das Hinzufügen oder Löschen lässt sich einfacher gestalten, wenn sie auf der *Cloud* abliegen. Es gibt noch mehr Punkte, welche in die Entscheidung der zu implementierenden Architektur miteinfließen. Darunter fällt das Logging, Monitoring, erweitertes *Security*-Konzept oder wie das gesamte System auf viele Fahrzeuge skaliert werden kann. Da es jedoch nur prototypisch für ein einziges Auto realisiert wird, werden diese Punkte nicht näher erläutert.

Der zweite Ansatz ist aufgrund der überwiegenden Vorteile, vor allem weil die Komplexität durch die Kommunikation mit der *Cloud* wegfällt, zu bevorzugen und wird daher in weiterer Folge umgesetzt.

5.2. Car Connectivity Unit

Der zentrale Baustein des Systems zur Identifizierung eines Fahrers ist die *Car Connectivity Unit*, welche im Auto verbaut wird. In der Architekturbeschreibung wurde schon auf ihre Funktionen hingewiesen. Daraus lassen sich folgende Hardware-Anforderungen für das Device extrahieren:

- kompakte Bauweise für das automotiv Umfeld
- CAN-Interface
- ausreichend Ressourcen für *Machine Learning* Anwendungen
- LTE Modem
- Debug/Flash Schnittstelle

Prinzipiell könnte ein *Raspberry-Pi* mit entsprechenden Modulen und einem robusten Gehäuse in Frage kommen. Da jedoch das Unternehmen *Bosch* eine Reihe an dezidierten CCUs im Portfolio hat und bei einer schon viel Know-How vorhanden ist, wird die sogenannte *Automotive Linux Edge Node* (ALEN) eingesetzt. Ursprünglich ist das israelische Unternehmen *CompuLab* [1] der Hersteller des Geräts und *Bosch* passt es an Automotive-Bedingungen an. Darunter fällt die Verstärkung des Gehäuses, das Hinzufügen einer CAN-Schnittstelle (RJ11-Buchse) sowie der Austausch des LTE-Moduls. Des Weiteren verfügt die ALEN über vier USB-Ports, eine serielle Schnittstelle über Micro-USB, ein WiFi Modul, zwei Gigabit Ethernet-Buchsen, jeweils einen Audio Ein- und Ausgang sowie einen HDMI-Ausgang. Hinzu kommen noch einige GPIOS, I2C, UART und SPI Interfaces. Als Prozessor ist ein 1GHz NXP i.MX 7Dual ARM Cortex-A7 verbaut, 1GB RAM sowie ein 12GB Flash-Speicher. Die Versorgungsspannung beträgt 12V. Die Abbildungen 5.2 zeigt die CCU.

Abbildung 5.2.: *Automotive Linux Edge Node* (ALEN, Quelle: `https://www.compulab.com/products/iot-gateways/`)

Auf der ALEN läuft ein individuelles Linux-System, welches für die Hardware angepasst und mithilfe einer SD-Karte auf das Gerät geladen wird. Zum Erstellen dieser speziellen Linux Distribution wird *Yocto* verwendet.

5.2.1. Yocto

Yocto [Yocb] ist ein *Open-Source* Projekt bestehend aus verschiedenen *Templates*, *Tools* und Methoden, um ein Linux-basiertes Betriebssystem für eingebettete Systeme zu bauen. Es wurde 2010 von der *Linux Foundation* ins Leben gerufen und wird seither auch von ihr verwaltet.

[1] `https://www.compulab.com/products/iot-gateways/`

Die drei Hauptkomponenten sind *BitBake*, *OpenEmbedded-Core* und *Poky*. Ersteres ist die *Build Engine*, ähnlich wie *make* für *C*. Sie interpretiert Konfigurationsdateien und *Recipes* (siehe weiter unten) und führt davon ausgehend Aktionen aus, beispielsweise herunterladen von *Source-Code* oder Binärdateien, konfigurieren und bauen von Applikationen sowie das Dateisystem. Die zweite Komponente ist eine Sammlung aus Basis-*Layer* für einige Hardwarearchitekturen wie zum Beispiel ARM oder MIPS. *Poky* ist sozusagen der Grundstein für jegliches Betriebssystem und vereint alles was dafür benötigt wird.

Jedes *Yocto* Projekt besteht aus mehreren Schichten (*Layer*), wobei die erste eben *Poky* ist. Diese bringt auch den Linux-Kernel mit ein. Die zweite Schicht beinhaltet alle Informationen für die zu verwendete Hardware und wird auch das *Board Support Package* (BSP) genannt. Darunter zählen die Kernel-Konfigurationen, der *Device-Tree* und Treiber. Für die ALEN stellt der Hersteller *Compulab* diese Schicht bereit. Die darüber liegenden *Layer* können weitere Spezifikationen und Modifikationen für das BSP enthalten, wenn z.B. sich die Peripherie geändert hat. Eine Regel der *Yocto-Community* besagt nämlich, dass niemals Basisschichten direkt verändert werden dürfen, sondern nur durch höhere. Dazwischen können noch weitere Schichten liegen, eventuell eine *Security*-Schicht, welche *Recipes* und Konfigurationen für einen sicheren Update-Mechanismus oder *SSH* umfasst. Die letzte Schicht wird dazu verwendet, um Anwender-Software hinzuzufügen, Benutzer anzulegen oder Konfigurationen für Dienste zu ändern.

All diese *Layer* enthalten die angesprochenen *Recipes*, welche Tasks spezifizieren, die wiederum von *BitBake* ausgeführt werden. Im Allgemeinen handelt es sich dabei um Informationen über eine einzelne Applikation. Diese inkludieren meistens den Ort von wo der *Source-Code* bezogen wird (z.B. *GitHub*), spezielle Einstellungen, wie kompiliert wird und wohin die resultierenden Dateien im Zielsystem gespeichert werden. Des Weiteren können auch *Patches* vorhanden sein, die andere *Recipes* anpassen. Sie sind notwendig, wenn ein Pfad geändert werden muss, Kernel-Configs oder neue Dateien hinzukommen. Wird nun über die *Build Engine* ein *Image* gebaut, werden alle *Recipes* nach der Reihe bzw. parallel ausgeführt. Das Ergebnis ist das gesamte *Root-Filesystem* und alle Informationen die für das Booten des Geräts benötigt werden.

Das ist zugleich ein großer Vorteil von *Yocto* [Yoca]. Beim traditionellen Aufsetzen eines Linux-basierten Systems wird ein standard Betriebssystem installiert und im Nachhinein konfiguriert, neue Softwarepakete installiert usw. Hierbei ist das Zielsystem bereits voll eingerichtet und erfordert keine weiteren Aktionen für den gewünschten Betrieb. Zudem gibt es eine große *Community* rund um *Yocto*, die einerseits das Projekt stets voranbringt und andererseits viel Support bei Problemen liefert. Ein Vorteil ist auch, dass die erforderliche *Toolchain* zum Entwickeln voll enthalten ist und leicht an eigene Bedürfnisse angepasst werden kann. Zusätzlich bietet das *Layer*-Modell Flexibilität und Wiederverwendung der einzelnen Schichten. Soll ein System auf eine andere Hardware portiert werden, braucht nur das *Board Support Package* ausgetauscht werden. Durch die Komplexität ergibt sich unweigerlich eine große Einstiegshürde. Die Lernkurve ist anfangs sehr steil und initial muss viel Zeit aufgewendet werden, um alles Notwendige einsatzfähig zu haben. Nichtsdestotrotz ist *Yocto* bei der Erstellung von hardwareabhängigen Betriebssystemen für *Embeded Systems* und vor allem auch für den IoT Bereich sehr beliebt.

Die unten angeführte Abbildung zeigt die *Layer* für dieses Projekt. Hier ist anzumerken, dass die Namenskonvention den Präfix *meta-* für die *Layer* vorschreibt. Die ersten zwei Schichten bilden die Basis mit dem Linux-Kernel. *Freescale* bringt die für NXP wesentlichen Konfigurationen für die CPU ein. Die vierte Schicht installiert *Python* in das *Image* und *meta-alen* enthält alle Informationen speziell für die Hardware, welche von *Compulab* und *Bosch*

kommen. Die letzte Schicht inkludiert die zusätzlichen Module für *Python*, etwa *Scikit-Learn* und *Pandas*, sowie die Applikationen zum Aufzeichnen des CAN-Busses und für *Machine Learning*. In den nächsten Abschnitten wird auf die Software näher eingegangen.

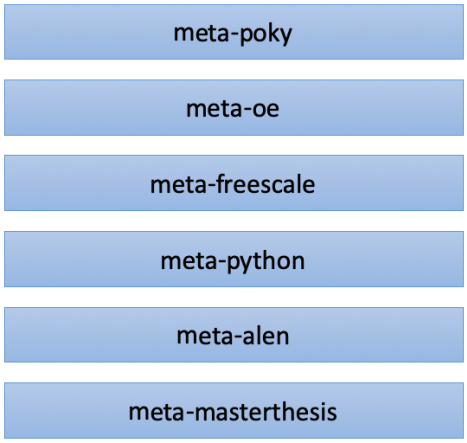


Abbildung 5.3.: *Yocto Layer*

5.2.2. Verkabelung im Fahrzeug

Damit die ALEN Zugriff auf den CAN-Bus bekommt, muss sie an ihn angeschlossen werden. Hierfür kommt prinzipiell jede beliebige Stelle der Linie im Auto infrage. Zum Beispiel kann dazu das CAN-Gateway angezapft werden oder ein T-Stück an den zwei Kabeln irgendwo in der Nähe des Fahrerraums. Hinter dem Lenkrad ist deshalb eine sehr gute Position, da es mit dem Antriebsstrang verbunden ist und viel Platz für die Montage bietet. Der Kabelbaum wird daher aufgetrennt und jeweils eine abzweigende Leitung an den CAN-High und CAN-Low gelötet. Bevor diese in eine RJ11 Buchse gesteckt werden, müssen sie gegen Reflexionen mit einem 120Ω Widerstand terminiert werden. Für die Spannungsversorgung wird der 12V Zigarettenanzünder mit einem Adapter verwendet. Abbildung 5.4 zeigt den schematischen Aufbau.

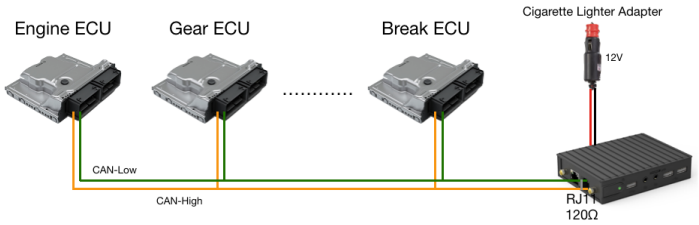


Abbildung 5.4.: ALEN Verkabelung

5.3. Softwarekomponenten

Die Architektur des Systems zeigt bereits auf, welche Softwarekomponenten auf der CCU zu realisieren sind. Diese sind in der folgenden Liste angeführt und in weiterer Folge im Detail erklärt.

1. **CAN-Logger** zeichnet CAN-Nachrichten der Fahrwerks-CAN-Linie in fast-Echtzeit auf und speichert sie in MDF-Dateien ab.
2. **Data-Preprocessor** bereitet die Daten für das *Machine Learning* Model auf und speichert sie in HDF5-Dateien ab.
3. **ML-Model Trainer** erstellt ein *Machine Learning* Model mit einem *Random Forest Classifier* und trainiert es mit den abgespeicherten Fahrerprofilen.
4. **ML-Model Predictor** nimmt neue CAN-Dateien entgegen und klassifiziert die Datenpunkte mit dem ML-Model.
5. **Result Presenter** sendet eine Benachrichtigung an den Fahrzeughalter, wenn eine gewisse Genauigkeit bei ausreichend Datenpunkten erzielt wird.

5.3.1. CAN-Logger

Der Eintrittspunkt vom Fahrzeug zur ALEN ist über die CAN Schnittstelle. Bevor die CAN-Daten überhaupt verarbeitet werden können, muss es dem Betriebssystem möglich sein am Bus teilnehmen zu können. Es braucht auf der Netzwerkebene ein Interface und darüberliegend einen Treiber. Dies wird vom Kernel über das *SocketCAN* Modul bereitgestellt [KB+12]. Es ist eine *Open-Source* Sammlung aus einer Implementierung des CAN-Protokolls, Linux Treiber und einer CAN-Socket API. Das ermöglicht eine ähnliche Handhabung wie mit dem TCP/IP Protokoll-Stack, sodass sich Anwendungen gewöhnlich auf ein CAN-Interface mit einem Socket verbinden und Nachrichten empfangen, sowie senden können.

So verfährt auch der *CAN-Logger*, dessen Aufgabe es ist, relevante Nachrichten vom Bus mitzulesen und abzuspeichern. Die Software ist in der Programmiersprache *Go* entwickelt, da von *Bosch* einige Bibliotheken für das Aufzeichnen und Verarbeiten von CAN-Daten für Linux-Systeme existieren. Durch eine Socketverbindung auf das CAN-Interface wird jede einzelne Botschaft als *Raw Dataframe* eingelesen. Erst durch den Abgleich des *Message Identifiers* mit der abgelegten CAN-Matrix kann die Nachricht interpretiert werden. Die Umrechnung der Werte in den dazugehörigen physikalischen Größen wird von einer *Library* mithilfe der Spezifikationen von der DBC-Datei erledigt. Aus Performancegründen werden nicht alle Nachrichten von der CAN-Linie verarbeitet. In der Datenbank sind nur jene inkludiert, die für diese Anwendung relevant sind. Die originale Datei beinhaltet nämlich für den Antriebsstrang etwa 270 Nachrichten mit 1500 Signalen. Dies gilt konkret für einen VW Golf 7, jedoch unterscheidet es sich kaum zu anderen Fahrzeugtypen. Die tatsächlichen relevanten Signale können aus den Erkenntnissen des vorherigen Kapitels entnommen werden. Das Resultat der *Feature Selection* aus 4.3.4 hat gezeigt, dass nur 21 *Features* für ein aussagekräftiges ML-Ergebnis ausreichend sind. Welche das sind, sind in der Abbildung 4.7b ersichtlich. Diese sind aus den folgenden acht Signalen errechnet und in sechs CAN-Botschaften enthalten.

Bremsdruck, *Geschwindigkeit*, *Motordrehzahl*, *Fahrpedalrohwert*, *Längsbeschleunigung*, *Querbeschleunigung*, *Lenkradwinkel*, *Lenkradgeschwindigkeit*

Die DBC-Datei wird demnach auf ein minimales Subset reduziert. Neben dem angesprochenen geringeren Rechenaufwand für den *CAN-Logger* selbst, sinkt der notwendige Speicherbedarf und auch die darauffolgenden SW-Komponenten werden entlastet. Sie müssen nur die Signale verarbeiten, welche für die *Machine Learning* Anwendung sinnvoll sind. Beim Erhalt einer CAN-Nachricht legt die Software die umgerechneten Werte mit einem in Nanosekunden aufgelösten Zeitstempel in den Speicher ab und nach jeweils einer Minute werden alle gesammelten Daten in eine MDF-Datei gespeichert.

5.3.2. Data-Preprocessor

Das Resultat dieser Komponente sollen Daten sein, die direkt für das *Machine Learning Model* verwertbar sind. Das bedeutet, dass sie das gleiche Format und die Struktur haben müssen, wie jene mit denen das *Model* trainiert wurde. Da für den RF-Algorithmus durch das 4. Kapitel bereits optimale Parameter mit den vorhanden Datensätzen bekannt sind, werden die neuen Daten auf die gleiche Struktur gebracht. Des weiteren wird ebenfalls *Python* mit den verwendeten Modulen eingesetzt.

Hierfür überwacht der *Data-Preprocessor* das Verzeichnis in dem der *CAN-Logger* die Dateien abspeichert und wenn eine neue hinzukommt, wird sie eingelesen. Der erste Schritt ist, die Frequenzen aller Signale anzugleichen. Wie in Abschnitt 3.2 beschrieben beträgt sie 0.5Hz. Danach gilt es die gewünschten 21 *Features* aus den Signalen zu bekommen. Je nach Signal werden Minimal-, Maximal-, Durchschnittswert, Standardabweichung und Median berechnet. Der nächste Schritt löscht alle Datenreihen, wo sich das Auto im Stillstand, zum Beispiel bei einer roten Ampel, befindet. Diese Daten können nichts zur Klassifizierung beitragen, da sie keine Aussagekraft haben. Als letztes erfolgt das Speichern der neuen Datensätze in HDF5 Dateien.

5.3.3. ML-Model Trainer

Der *ML-Model Trainer* ist die erste Softwareinstanz, welche nach dem Hochfahren der ALEN gestartet wird. Die Funktion dessen ist, einen *Random Forest Classifier* zu erstellen und mit abgelegten Fahrerprofilen zu trainieren. Auch hier kommt *Scikit-Learn* zum Einsatz und die Parameter werden wie erwähnt von den Optimierungsresultaten übernommen. Die Profile sind durch den *CAN-Logger* aufgezeichnete und vom *Data-Preprocessor* gefilterte Daten von jeweils einer Person. Dadurch ist sichergestellt, dass sie die gleichen Eigenschaften wie die zu klassifizierenden Daten haben und zu 100% zuordenbar sind.

5.3.4. ML-Model Predictor

Nach dem *Data-Preprocessor* ist der *ML-Model Predictor* geschaltet. Er überwacht wiederum das Verzeichnis, indem die HDF5 Dateien nach der Vorverarbeitung gespeichert werden. Kommt eine neue Messdatei hinzu, wird sie eingelesen und auf ihre Korrektheit validiert. Darunter zählt die Überprüfung, ob alle Datenreihen inkludiert sind und ob alle Signale zu jedem gegebenen Zeitpunkt einen Wert haben. Ergeben sich Inkonsistenzen, wird die Datei übersprungen. So wird vermieden, dass es zu Verfälschungen während der Analyse kommt und dadurch auch zu keiner Missklassifizierung. Hat die Datei die erwartete Struktur, wird sie zu den in dieser gestarteten Fahrt bereits aufgenommen Daten hinzugefügt und in das erstellte Ml-Modell eingespeist. Das Ergebnis ist eine Reihe an Werten, welche jeweils die prognostizierte Klasse sind. Die Einträge repräsentieren somit den klassifizierten Fahrer für die entsprechende Eingabe-Datenreihe. Der nächste Schritt ist die Auswertung, um welche Fahrerin es sich dabei handelt. Als erstes wird die Häufigkeit der vorhandenen Klassen (1 bis 4) ermittelt. Danach können mehrere Methoden zur Evaluierung eingesetzt werden.

Grenzwert

Eine Möglichkeit ist folgende. Übersteigt der prozentuale Anteil eines Fahrers eine gewisse Grenze, ist mit dieser Wahrscheinlichkeit das der Fahrer. Zum Beispiel sind nach fünf Minuten Fahrzeit 150 Datenreihen (30 pro Minute, 0,5Hz) verfügbar. Das Resultat des *Random*

Forest Models hat ergeben, dass etwa sieben Messpunkte (5%) zu Fahrer *1* gehören könnten, 15 (10%) zu Fahrer *2* und über 22 (15%) zum Fahrprofil *3*. 105 (70%) Eingabedaten deuten jedoch auf Fahrer *4* hin. Muss die Grenze von 70% erreicht werden, um eine Fahrer identifizieren zu können, würde das Endergebnis der Klassifizierung diesen Fahrer bevorzugen.

Differenz

Angenommen der *Output* des Modells ist nicht so eindeutig und ergibt jeweils für Fahrerin *1* und *2* 5%, für Fahrerin *2* 30% und 60% für Fahrerin *4*. Nach dem ersten Ansatz wäre hier keine eindeutige Identifikation möglich. Um trotzdem auf ein zuverlässiges Ergebnis zu kommen, kann eine andere Herangehensweise angewendet werden. Sie betrachtet nur die zwei am häufigsten vorkommenden Klassen - in diesem Fall Fahrerin *3* und *4* - und bildet davon die Differenz. Ist diese groß genug, kann es für eine ausreichende Identifikation genügen. Hier bietet sich der Wert 30 an (Kriterium 1). Dadurch ist auch sichergestellt, dass die Klasse mit dem höchsten Anteil mindestens 50% hat (Kriterium 2). Für ein besseres Verständnis sind unten Beispiele angeführt, welche alle für die Fahrerin *4* sprechen.

Tabelle 5.1.: Beispiel Differenzansatz für Fahrerin 1 - 4

	P(1)	*P(2)*	*P(3)*	*P(4)*	Kriterium 1	Kriterium 2	Ergebnis
1. min	0%	0%	30%	70%	70 - 30 $\geq$ 30	70 $\geq$ 50	*P(4)*
2. min	10%	20%	10%	60%	60 - 20 $\geq$ 30	60 $\geq$ 50	*P(4)*
3. min	20%	20%	20%	50%	50 - 20 $\geq$ 30	50 $\geq$ 50	*P(4)*
4. min	10%	10%	30%	60%	60 - 30 $\geq$ 30	60 $\geq$ 50	*P(4)*
5. min	0%	10%	20%	75%	75 - 20 $\geq$ 30	75 $\geq$ 50	*P(4)*

Zeitliche Veränderung

Weiters kann auch die zeitliche Veränderung der Häufigkeiten analysiert werden. Das ist hilfreich, wenn sich die Individualität der Personen erst nach den ersten Fahrminuten für das ML-Modell sichtbar macht. So müssen nicht mehr Daten als nötig gesammelt werden, um die nicht-eindeutigen zu kompensieren. Die Methode analysiert jede Datei für sich und speichert die Zwischenergebnisse ab. Danach wird jeweils das Delta berechnet und der Maximalwert bestimmt. Ist das Delta positiv und übersteigt sowie der Maximalwert eine gewisse Grenze, kann der Lenker identifiziert werden. Die Tabelle 5.2 zeigt ein Beispiel, bei dem die ersten beiden Ansätze noch keinen Fahrer eindeutig identifizieren könnten. Im Durchschnitt sind 45% der Daten auf Fahrer *4* klassifiziert worden, was zu wenig ist. Die zeitbasierte Veränderung spricht jedoch klar dafür.

Um eine Methode auswählen zu können, welche letztendlich für das System zum Einsatz kommt, wurde jede implementiert und mit den Testdaten mehrmals evaluiert. Das hat ergeben, dass sich der zweite Ansatz am besten hierfür eignet. Im Gegensatz zum ersten ist er flexibler und findet schneller den korrekten Fahrerin. Zusätzlich hat sich herausgestellt, dass bereits nach den ersten Minuten das Fahrverhalten der Testfahrer unterscheidbar ist. Daher ist die letzte Implementierung nicht anwendbar, da sich das Delta nicht merklich verändert. Als Absicherung kann des Weiteren die Anwendung so konfiguriert werden, dass die identifizierte Fahrerin erst dann bestätigt ist, wenn sich das Ergebnis bei den darauffolgenden Durchläufen nicht mehr ändert. Umso länger abgewartet wird, desto sicherer gilt es, aber umso länger muss mit der darauf basierenden Aktion gewartet werden. Es gilt daher das

Tabelle 5.2.: Beispiel Zeitliche Veränderung der Wahrscheinlichkeiten für Fahrer 1 - 4

	P(1)	*P(2)*	*P(3)*	*P(4)*
1. min	20%	20%	30%	30%
2. min	20%	25%	25%	30%
3. min	10%	20%	30%	40%
4. min	0%	20%	25%	55%
5. min	0%	10%	20%	75%
⌀	10%	18%	26%	45%
Δ	-20%	-10%	-15%	+45%
max	20%	25%	30%	75%

richtige Verhältnis zwischen der Identifikationszeit und wie lang auf die Umsetzung der Aktion gewartet werden darf, zu finden. Beispielsweise macht es für individuelle Anpassungen keinen Sinn länger als zehn Minuten zu warten, aber der Eintrag ins digitale Fahrtenbuch kann auch erst am Ende der Fahrt erfolgen.

5.3.5. Result Presenter

Wie die Aktion nun ausgelöst wird, übernimmt die Komponente *Result Presenter*. Je nach Service kann dies auf unterschiedlichen Kommunikationswegen geschehen. Soll das Auto selbst darauf reagieren, zum Beispiel das Anzeigen der zuletzt ausgewählten Ziele im Navigationssystem, kann die Information wieder zurück über den CAN-Bus übertragen werden. Wird hingegen nach einer gewissen Zeit gar kein Fahrer erkannt, womöglich also eine unautorisierte Person, ist es denkbar eine SMS an den Fahrzeughalter zu versenden. Da die ALEN ein GSM Modul mit SIM-Karte integriert hat, ist es ohne weiteres umsetzbar. Über die Internetverbindung können auch Dienste des Herstellers oder von Drittanbietern angebunden werden, welche ein REST API zur Verfügung stellen. Es hängt demnach sehr stark davon ab, was mit der Erkenntnis passieren soll.

5.3.6. Sequenzdiagramm

Das folgende Diagramm (Abbildung 5.5) zeigt die Funktionen der einzelnen Komponenten. Dadurch, dass fast kein tatsächlicher Nachrichtenaustausch stattfindet, sondern meist nur Dateien abgelegt werden, gibt es wenige direkte Querverbindungen. Nach dem Hochfahren der ALEN, was in etwa zehn Sekunden dauert, wird zuerst der *CAN-Logger* und der *ML-Model Trainer* gestartet. Danach folgt der *Data-Preprocessor*, der *ML-Model Predictor* und der *Result Presenter*. Obwohl es aus logischer Sicht unabhängige und alleinstehende Komponenten sind, werden die letzten drei bei der Umsetzung zusammengeführt. Das hat den Grund, dass einerseits die Klassifizierungslogik das zuvor erstellte Modell benötigt und andererseits unmittelbar danach die Funktion zum Weiterleiten der Information, welche kein großes Ausmaß annimmt, aufgerufen wird.

5.4. Simulation

Das letzte Ziel der Masterarbeit ist, das entwickelte System in ein Fahrzeug zu integrieren. Da jedoch das Unternehmen, mit dem es umgesetzt wird, leider kein entsprechendes bereitstellen kann, muss es simuliert werden. Normalerweise würden vier Personen mehrere Testfahrten

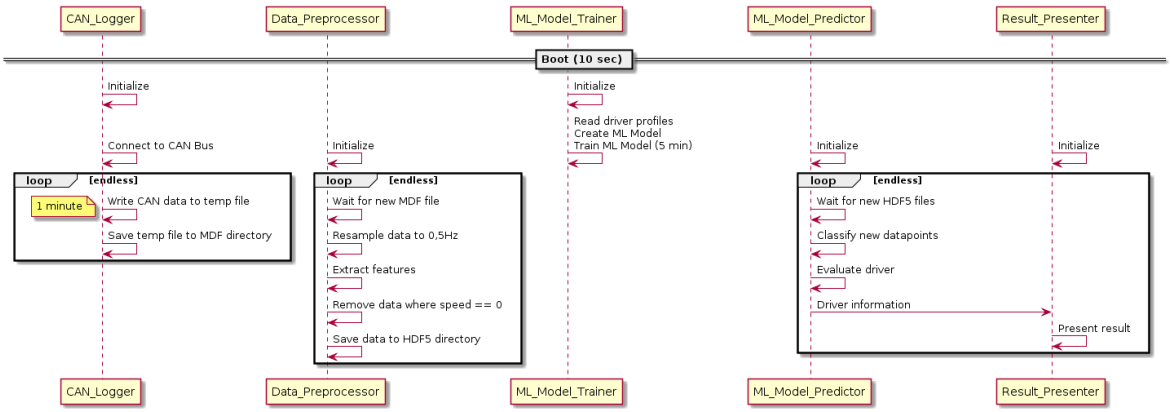


Abbildung 5.5.: Sequenzdiagramm Software Komponenten

durchführen, um ein Profil anzulegen. Bei weiteren Fahrten könnte das System in Folge dessen erprobt werden. Da dies eben nicht möglich ist, dienen hierzu zufällige vier der vorhandenen Datensätze. Für diesen Zweck werden die Start- und Endzeitpunkte jeder Fahrt extrahiert und abgespeichert. Im Durchschnitt enthält ein Datensatz 46 Fahrten, von dem per Zufall zehn Fahrten entnommen werden, sodass diese nicht mehr in der originalen Datei enthalten sind. Das ist insofern wichtig, damit der *Random Forest Classifier* die Testdaten nicht kennt und das Ergebnis verfälscht. Nach diesem Vorgang existiert daher pro Fahrer ein Trainingsset aus ungefähr 36 Fahrten und ein Testset aus zehn. Werden die Daten für einen Test, wie er zum Beispiel in Abschnitt 3 oder 4 gemacht wurde, verwendet, kommt eine Genauigkeit des Modells von ca. 94% heraus. Die Steigerung erklärt sich durch die geringere Anzahl an Fahrerprofilen.

Um die Simulation so realgetreu wie möglich zu implementieren, wird nur der *CAN-Logger* ersetzt. Stattdessen werden von einem eigenen Programm (*MDF-Simulator*) MDF Dateien in das Verzeichnis abgelegt, aus dem der *Data-Preprocessor* diese bezieht. Bei den MDF Dateien handelt es sich um jene, welche im 3. Kapitel vorgestellt wurden und bilden das Testset. Da sie jeweils eine Minute an Fahrtzeit beinhalten, wird auch immer eine Datei nach einer Minute in das dezidierte Verzeichnis hinzugefügt. Die Reihenfolge ist dabei chronologisch, beginnend mit der ersten Minute einer Fahrt. Durch die neue Komponente wird lediglich das erste Glied (CAN-Bus und -*Logger*) simuliert. Für die restlichen Software-Teile hat es keine Auswirkung. Da das ganze System auch auf der ALEN in Betrieb genommen wird, ergeben sich fast reale Bedingungen und die daraus resultierenden Ergebnisse haben eine hohe Aussagekraft.

Zusammenfassend lässt sich die Simulationsumgebung folgendermaßen beschreiben: Die Softwarekomponenten werden wie beschrieben implementiert, lediglich der *CAN-Logger* wird durch den *MDF-Simulator* ersetzt. Für die Identifikation eines Lenkers kommt der *Differenz*-Ansatz um Einsatz. Als Fahrerprofile werden vier der vorhandenen Datensätze herangezogen, bei denen zehn Fahrten gelöscht werden. Die original MDF Dateien von den gelöschten dienen hierzu als Eingabedaten. Bei einem Testdurchlauf verwendet der *MDF-Simulator* immer nur Dateien von einer Fahrerin, da es darum geht, nur einen zu identifizieren. Zur Benachrichtigung wird ein SMS-Versand mit dem integrierten Modul und SIM-Karte umgesetzt.

5.4.1. Fahrererkennung

Das Listing 5.1 zeigt einen Ausschnitt der letzten drei genannten Komponenten. Von Zeile zwei bis elf werden die abgelegten Fahrerprofile geladen, ein *Random Forest Classifier* erstellt

und trainiert. Ab Zeile dreizehn beginnt die Logik für die Fahrererkennung. Es beinhaltet das Einlesen neuer Messdaten sowie die Klassifizierung durch das ML-Modell. Danach werden die Häufigkeiten der Klassen gezählt und die zwei am meisten vorkommenden ermittelt. Anschließend wird die Überprüfung nach dem *Differenz*-Ansatz durchgeführt und schlussendlich eine SMS Versand, wenn der Fahrer oft genug bestätigt wurde.

```
1     [...]
2     # ML Model Trainer
3     profiles = get_profiles(config["profile_dir"])
4     features = config["features"][:21]
5     X = profiles[features].values
6     Y = profiles["class"]
7
8     clf = RandomForestClassifier(n_estimators=300, min_samples_leaf=1,
          min_samples_split=3, criterion="gini", max_depth=None)
9     clf.fit(X, Y)
10
11    # ML-Model Predictor
12    all_data = []
13    while True:
14        data = []
15        for file in os.listdir(config["input_dir"]):
16            df = pd.read_hdf(os.path.join(config["input_dir"], file))
17            data.append(df)
18
19        all_data = all_data + data
20
21        pdata = pd.concat(all_data, sort=False)
22        pdata = pdata.values
23
24        pred = clf.predict(pdata)
25
26        ids, counts = np.unique(pred, return_counts=True)
27        ids = dict(zip(ids, counts))
28        first = {"id": None, "count": 0}
29        second = {"id": None, "count": 0}
30        for id in t:
31            c = ids[id]
32            if c > first["count"]:
33                first["count"] = c
34                first["id"] = id
35            elif c > second["count"]:
36                second["count"] = c
37                second["id"] = id
38
39        first_conf = first["count"] / len(pred)
40        second_conf = second["count"] / len(pred)
41        diff = first_conf - second_conf
42        if diff >= 0.30 and first_conf >= 0.50:
43            if confirmed_id == first["id"]:
44                confirmation_count += 1
45            else:
46                confirmed_id = first["id"]
47                confirmation_count = 1
48
49        if confirmation_count == config["confirmation_count"]:
50            # Result Presenter
51            send_SMS("\nDriver with id %s is driving" % first["id"], config["
                  number"])
52            break
```

```
53      [ . . . ]
```

Listing 5.1: Ausschnitt Fahreridentifikation

5.4.2. Ergebnisse

In diesem Abschnitt werden kurz die Ergebnisse der Simulation auf der ALEN präsentiert. Bis das *Machine Learning Model* fertig trainiert ist, vergehen ca. fünf Minuten bei vier Profilen mit allen verfügbaren Fahrten (exklusive zehn). Aufgrund dessen beträgt die Anzahl wie oft eine Fahrerin hintereinander identifiziert werden muss fünf. Es liegen nämlich nachdem das Trainieren des Modells abgeschlossen ist, bereits fünf Dateien mit jeweils einer Minute an Fahrzeit vor. Für das Klassifizieren einer neuen Datei benötigt das Gerät unter eine Sekunde.

Bei einem Testdurchlauf wurde jeweils eine Fahrt herangezogen und festgestellt wie viele Minuten nötig sind, bis der Fahrer korrekt identifiziert wird. Um ein fundierteres Ergebnis zu bekommen, ist der Vorgang öfters mit permutierten Trainings- und Testfahrten durchlaufen worden. Der ganze Vorgang wurde auch mit allen vier Fahrern vollzogen. Dabei ist das Folgende herausgekommen.

- Der Algorithmus hat mit einer Wahrscheinlichkeit von 97% den Fahrer korrekt identifiziert.
- Lediglich drei Fahrten wurden falsch zugeordnet.
- Nur eine Fahrt konnte gar nicht zugeordnet werden.
- Für eine Identifikation wurden im Schnitt 5,73 Minuten an Fahrzeit benötigt.

6. Diskussion

Das letzte Kapitel vor der Zusammenfassung präsentiert die erzielten Ergebnisse der Masterarbeit und diskutiert die Erkenntnisse. Die Gliederung spiegelt die behandelten Forschungsfragen. Des Weiteren wird ein Ausblick über weiterführende Tätigkeiten gegeben.

6.1. Validierung der Methoden

Im Abschnitt 1.3 wurden vier Paper vorgestellt, welche sich mit dem gleichen Thema - Fahreridentifikation basierend am Fahrverhalten - beschäftigen. Diese haben hauptsächlich den *Machine Learning* Algorithmus *Random Forest* verwendet und damit gute Resultate erzielen können (teilweise über 85%). Es wurde jedoch sehr wenig über die Struktur, Eigenschaften und Herkunft der Daten erwähnt. Daher war die erste Frage: Sind die existierenden Methoden mit den vorhandenen Daten valide? Kapitel 3 hat gezeigt, dass dies zutrifft, da bei einem ersten Versuch eine Genauigkeit des Modells von 85% herausgekommen ist. Weitere Durchläufe haben hingegen eine hohe Schwankungsbreite - von 78,66% bis 92,53% - ergeben. Das lässt sich auf zwei Gründe zurückführen. In Sektion 4.3 bei der *Feature*-Analyse wurde entdeckt, dass das Signal *Gangposition* Unregelmäßigkeiten aufweist. Bei fünf von neun Datensets ist der Wert durchgehend *14*. Die guten Ergebnisse (über 90%) können zum Teil daher kommen, dass das Trainings- und Testset gleichverteilt Datenpunkte mit und ohne Gangposition *14* haben. Dadurch bekommt, wie in Abbildung 4.7a gezeigt, das *Feature* eine große Bedeutung und hilft ungemein bei der Klassifizierung. Der zweite Grund ist, dass die Auswahl der Daten einen Einfluss auf die Genauigkeit haben. Befinden sich viele Autobahnkilometer, welche wenig Individualität aufweisen und schwerer zuordenbar sind, unter den Sets, sinkt die Trefferquote. Das Gegenteil tritt bei vielen Datenpunkten von Lenk- oder Bremsmanöver auf, die Trefferquote steigt.

Das 3. Kapitel hat die vorliegenden Daten beschrieben. Im Gegensatz zu [GRDW18] sind sie direkt von mehreren CAN-Linien abgegriffen worden und nicht über den Diagnostik-Port (OBD II). Das bringt einige Vorteile. Zum Beispiel sind aufgrund dessen alle vorhandenen Signale am Bus verfügbar. Alleine am Fahrwerks-CAN sind bis zu 1500 und am Komfort-CAN über 2500 Signale vorhanden. Je nach Hardwarekapazitäten und Fahrzeugmodell können auch noch mehrere CAN-Linien miteinbezogen werden. Damit wird das ML-Modell genauer und aussagekräftiger, wenn es mehr Daten zum Verarbeiten gibt. Währenddessen über den OBD II Port [SAE17] laut Spezifikation maximal 196 Signale übertragen werden. Hier handelt es sich meist auch nur um verschiedenen Abgaswerte, Fehlerhistorie und Temperaturen. Hersteller können zudem noch weitere Signale in die Schnittstelle implementieren, welche aber nicht spezifiziert sind. Um die Werte über OBD II auslesen zu können, muss ein Dongle eingesetzt werden, der in einen dezidierten Anschluss, meist versetzt unterhalb des Lenkrads, eingesteckt wird. Er hat das OBD Protokoll implementiert und kann die Signale auswerten und in die physikalische Einheit umrechnen [OBD]. Für die Weiterverarbeitung wird gewöhnlich eine Bluetooth-Verbindung zu einem weiteren Gerät (z.B. Smartphone) benötigt. Für diese Anwendung wäre die ALEN notwendig, die die Daten empfängt und mit dem *Machine Learning* Algorithmus klassifiziert. Durch den OBD II Dongle ergeben sich zusätzlich einige Angriffs-

vektoren, mithilfe dieser die Privatsphäre verletzt, Eigentum beschädigt und sogar Leib und Leben gefährdet werden könnte. Forscher haben hierzu 77 am Markt verfügbare Dongles getestet und sind zum Entschluss gekommen, dass alle mindestens zwei Schwachstellen enthalten [USE20]. Beispielsweise sind keine Authentifizierungsmethoden realisiert, lassen das Extrahieren der Firmware ohne Sicherheitsmechanismus zu oder senden CAN-Nachrichten am Bus ohne jegliche Validierung. Das Abgreifen direkt vom Bus ist daher die klar bessere Methode und bietet mehr Signale, bessere Sicherheit und Schutz der Privatsphäre.

6.2. Optimierung

Bei der Beantwortung der zweiten Forschungsfrage wurde versucht, den *Random Forest* Algorithmus zu optimieren. Im ersten Schritt sind die Parameter analysiert worden. Dabei hat sich herausgestellt, dass die Standardwerte bereits sehr gute Ergebnisse liefern. Selbst nach 16128 Durchläufen, bei denen jede mögliche Kombination der Parameter überprüft wurde, konnte lediglich eine Verbesserung um fast ein Prozent erzielt werden. Auffallend war, dass eine Veränderung des Wertes bei dem Parameter *criterion* - von *gini* auf *entropy* - zwar keine Auswirkung auf die Trefferquote gehabt hat, jedoch sich die Durchlaufzeit erheblich vergrößert hat, nämlich um mehr als 300%. Der Grund hierfür ist die dahinterstehende Formel. *Entropy* wird mit dem Logarithmus der Basis 2 von der Wahrscheinlichkeit berechnet, wohingegen bei der *Gini Impurity* die Wahrscheinlichkeit mit sich selbst multipliziert wird (siehe 2.1.3). Da die Logarithmus-Funktion mehr Rechenleistung erfordert als eine Multiplikation, ist das Aufteilen eines Knotens mit *entropy* langsamer.

Durch eine detailliertere Analyse der vorhanden Datensätze hat sich eines verdeutlicht: Die Daten sind das ausschlaggebendste für die *Machine Learning* Anwendung. Diese Aussage wird auch immer wieder in der Literatur ([RRC19], [Con12]) bestätigt. Damit ist gemeint, dass zum einen die Struktur einheitlich und bekannt und zum anderen ausreichend viele Daten erhältlich sein müssen. Gibt es beispielsweise mehrere fehlende Werte oder Unregelmäßigkeiten (siehe *Gangposition*) können manche ML Algorithmen nicht zuverlässig Ergebnisse liefern. Das gleiche gilt wenn einfach zu wenig verfügbar sind. Das Regelset des Modells kann so nicht hinreichend gebildet werden, um die Vielfalt der existierenden Daten abzubilden.

Ein ML-*Model* stellt in gewisser Weise eine *Blackbox* dar, da nicht ohne einer näheren Untersuchung erkennbar ist, auf Grund welcher Daten eine Entscheidung getroffen wird. Diese Erfahrung hat auch der Autor von [Gah19] gemacht, der ebenfalls den Lenker anhand von CAN- und Smartphone-Daten identifizieren wollte. Er hat das Modell mit allen zugänglichen Daten trainiert, darunter auch Längen- und Breitengrade des GPS-Moduls. Der *Classifier* hat am Ende nicht basierend am Fahrverhalten Datenpunkte zugeordnet, sondern alleine mittels den GPS-Daten. In Abschnitt 4.3.2 wurde ähnliches aufgezeigt. Mit Hilfe der integrierten *Feature Importance* des *Random Forest*s ist bemerkt worden, dass es beim Signal *Gangposition* keine durchgängigen Werte gibt. Der Missstand wurde bereits oben erläutert und bekräftigt die genannte These, dass die Daten vor dem Einsatz mit *Machine Learning* genauestens analysiert und validiert werden müssen. Das verhindert eine falsche Vorgehensweise des ML-Modells als *Blackbox*.

6.3. Dauer der Trainings- und Identifikationsphase

Im Zuge der Optimierungen und der Integration wurde auch die Fragestellung behandelt, wie viele Trainingsdaten vonnöten sind, um mindestens 85% Genauigkeit des *Models* zu erreichen.

Des Weiteren war die Frage, wie lang eine Person fahren muss, damit sie zu 85% identifiziert werden kann. Hier kann keine pauschale Antwort gegeben werden, da viele Faktoren Einfluss darauf nehmen. Mitsicherheit die maßgeblichsten davon sind die Anzahl an Fahrprofilen in einem Set und welche Daten ausgewählt werden. Die Abbildungen 4.5a und 4.5b zeigen den Unterschied zwischen einer zufälligen und chronologischen Auswahl. Bei ersteren ist die maximale Genauigkeit mit 40 Minuten Trainingszeit pro Fahrer bei etwa 75% gelegen. Sind die Daten jeweils nach Fahrtbeginn für die Trainingsphase herangezogen worden, wurde nach ca. 15 Minuten knapp 86% erreicht. Mit der Zufallsauswahl konnte das Ziel somit nicht erreicht werden. Das bestätigt die Annahme, dass Daten am Beginn einer Fahrt, wo mehrere Lenk- und Bremsmanöver stattfinden, eine höhere Individualität aufweisen und dadurch besser zu klassifizieren sind. Die beschriebenen Tests sind mit neun Fahrprofilen durchgeführt worden. Während der Integration hat ein Test mit nur vier eine Genauigkeit von 94% ergeben. Das beweist, dass allein die Reduktion auf eine geringere Anzahl einen großen Einfluss hat.

Für die Dauer der Identifikationsphase gilt ebenso das gleiche. Da sich das Kapitel *Optimierung* an erster Stelle auf die allgemeine Verbesserung des Modells konzentriert hat, ist dieser Aspekt nicht beleuchtet worden. Es liegen daher keine Ergebnisse für ein Modell mit neun Profilen vor. Das vorherige Kapitel ist jedoch explizit auf die Fragestellung eingegangen. Allerdings kann wieder keine pauschale Aussage getroffen werden, da ein paar Gegebenheiten vorausgesetzt waren. Zum einen sind im ML-Modell nur vier Datensätze inkludiert und zum anderen gibt es eine Einschränkung durch die Zeit die benötigt wird, um den *Random Forest Classifier* zu trainieren. Des Weiteren ist die Art der Identifizierung durch den eingesetzten Algorithmus (siehe 5.3.4) gesteuert. So muss zum Beispiel fünfmal ($\hat{=}$ 5 Minuten) hintereinander die gleiche Fahrerin identifiziert werden. Mehreren Tests mit verschiedenen Daten haben hierbei ergeben, dass 97% davon im Durchschnitt nach 5,73 Minuten identifiziert sind.

6.4. Integration

Die letzte Forschungsfrage zielt auf die Integration des Systems zur Fahreridentifikation in ein Fahrzeug ab. Wie in dem Kapitel beschrieben, ist es nicht möglich gewesen, da ein tatsächliches Auto nicht zur Verfügung gestellt wurde. Stattdessen ist der CAN-Bus durch die bereits vorhandenen Daten simuliert worden. Die Beschreibungen und Ergebnisse haben gezeigt, dass dies sehr gut funktioniert hat und es auf diese Weise zu fast realen Bedingungen gekommen ist. Die ALEN als zentrale Einheit eignet sich prinzipiell gut für diesen Einsatz, da sie die notwendige Peripherie mitbringt, hat aber klar ihre Grenzen in Bezug auf *Machine Learning*. So dauert es ca. fünf Minuten, bis die CCU den *Random Forest Classifier* erstellt und trainiert hat. Einige Maßnahmen könnten die Zeit reduzieren. Beispielsweise kann die ALEN durch eine leistungsstärkere CCU mit sonst ähnlicher Hardwareausstattung ersetzt werden. Denkbar wäre zudem, das trainierte Modell zu speichern und bei Fahrtbeginn nur zu laden. Den Prozess komplett in die *Cloud* auszulagern, erfordert das Umsetzen von weitgehenden *Security*- und *Privacy*-Anforderungen, da sich die persönlichen Fahrprofile nicht mehr nur lokal im Auto befinden. Weiters würde dadurch das Konzept von *Edge Computing* verloren gehen. Die Vorteile davon wurden in den Grundlagen näher gebracht, hat aber eben auch einen Nachteil, nämlich der von fehlenden Ressourcen am *Edge Device*, vorausgesehen.

Selbst wenn die Zeit auf ein Minimum reduziert werden könnte, würde es die Identifizierung nicht zwangsläufig beschleunigen. Sie hängt hauptsächlich vom Zweck ab und wird von der Konfiguration des *ML-Model Predictor* bestimmt. Dem Serviceanbieter - sei es der OEM oder eine Drittpartei - muss klar sein, dass je kürzer die Identifikationsphase gewählt wird, desto leichter kann es zu einer Missidentifikation kommen. Es gibt durchaus Services, welche erst

gegen Ende einer Fahrt die Identifikation benötigen. Zum Beispiel ist in der Einleitung das digitale Fahrtbuch besprochen worden, das vollkommen ohne jeglicher Interaktion seitens der Fahrerin auskommt. Eine Implementierung auf der ALEN wäre ohne viel Zutun möglich, da dort bereits alle Daten aufliegen. Eine weitere Einsatzmöglichkeit liegt bei *Car-Sharing* Anbietern. Sie können mit Hilfe des Systems erkennen, ob wirklich die registrierte Person hinter dem Lenkrad sitzt. Diese Information genügt es auch erst am Ende zu haben. Das gleiche Prinzip lässt sich zudem bei Mietwagenunternehmen realisieren. Die Standardverträge erlauben nur einen Lenker, jede weitere kostet extra Geld (8,50 € bei *Sixt* [SIX]). Die Überprüfung kann hiermit durchgeführt und bei Bedarf der Vertragszusatz automatisiert ergänzt werden.

Es gibt überdies Anwendungen, die eine schnelle Identifikation erfordern. Eine davon, die Diebstahlwarnung, wurde ebenfalls schon in der Einleitung erwähnt und in diesen Kontext gesetzt. Für die Umsetzung wäre bloß eine kleine Adaption der vorhandenen Anwendung notwendig, bei der eine zusätzliche Abfrage zu unzureichenden Klassifizierungen eingebaut wird. Nachdem mehrere Versuche Dateien korrekt zuzuordnen gescheitert sind, kann eine SMS an den Fahrzeughalter gesendet werden, die darüber informiert, dass womöglich eine nicht autorisierte Person das Fahrzeug steuert. Hier muss aber klar geprüft werden, ab wann entschieden wird, dass es sich wirklich um einen unbekannten Fahrer handelt und nicht um einen bekannten, der noch nicht identifiziert wurde.

Durch die Ergebnisse des letzten Kapitels ist davon auszugehen, dass ein paar Minuten an Fahrzeit benötigt werden, damit eine Lenkerin ausreichend erkannt wird. Aus diesem Grund sind einige Services nicht umsetzbar. Allen voran jene, welche unmittelbar nach Fahrtbeginn in Kraft treten würden. Das automatische Einstellen des bevorzugten Radiosenders oder das Einrichten der Außenspiegel fallen somit weg. Ebenso die Möglichkeit, dass Fahrerabhängig bestimmte *Features* de- und aktiviert werden, ist nicht vorstellbar. Das gleiche gilt für eine Drosselung der Leistung, wenn beispielsweise noch unerfahrene Personen hinter dem Steuer sitzen. Dies könnte gegebenenfalls sogar eine Gefahr darstellen, da in der Anfangszeit die volle Leistung zu einem erhöhten Unfallrisiko beiträgt.

6.5. Limitierungen

Da nur eine geringe Anzahl an Fahrdaten vorliegen, kann keine Aussage über die Genauigkeit des Systems in einer diverseren Umgebung erfolgen. Zudem fehlen CAN-Nachrichten von mehr Fahrzeugtypen, um das Modell mit den Signalen validieren zu können. Des Weiteren sind keine Informationen über Geschlecht, Altersgruppen und Dauer des Führerscheinbesitzes von den vorhandenen Fahrprofilen bekannt. Es ist daher auch unklar, ob es sich dabei um eine repräsentative Gruppe an Testpersonen handelt. Bei nur neun ist es anzunehmen. Obwohl die Simulation den realen Bedingungen sehr nahe kommt, können trotzdem nicht alle Eventualitäten abgebildet werden. Das resultiert wahrscheinlich in einer Abweichung der Zuverlässigkeit der Anwendung.

6.6. Ausblick

Um das vorgestellte System zur Fahreridentifikation zu verbessern, gibt es mehrere Möglichkeiten. Neben dem *Random Forest* sind in der Literatur auch andere Algorithmen zur Klassifizierung ([RRC19], [ETKK16]) beschrieben. Bei einem Vergleich könnte sich ein anderer als performanter und genauer herausstellen. Das Modell kann zusätzlich optimiert werden, indem eine *Feedback*-Schleife implementiert wird. Nach einer erfolgreichen Identifikation werden die

neuen Messpunkte dem Modell als neue Trainingsdaten zur Verfügung gestellt. Dadurch lernt der *Classifier* durchgehend neue Fahrsituationen. Ein anderer Ansatz wäre nur Datenpunkte herzunehmen, welche leichter zu klassifizieren sind. Kapitel 3 und 4 haben aufgezeigt, dass solche existieren. Hier stellt sich die Frage, welche Fahrmanöver eine höhere Individualität aufweisen. Die Messdateien können folglich im *Data-Preprocessor* auf die relevanten Daten reduziert werden. Als Anhaltspunkt dienen hierfür die Arbeiten [GRDW18] und [HSS+16], die sich zum Teil schon damit beschäftigt haben.

Ein weiterer Punkt ist die Anwendung mit einem echten Fahrzeug zu erproben, da dies nicht möglich war. Im Zuge dessen ist eine Überprüfung der Simulationsergebnisse möglich. Außerdem wäre es interessant zu wissen, ob sich das System durch eine manipulierte Fahrweise austricksen lässt. Dadurch kann die Stabilität getestet werden und ob die Fahrweise von einer Person nachgeahmt werden kann.

7. Zusammenfassung

Die Masterarbeit hat sich mit dem Thema *Fahreridentifikation mittels Machine-Learning* beschäftigt. Es basiert darauf, dass Autolenker ein individuelles Fahrverhalten im Straßenverkehr haben und anhand von Fahrzeugdaten identifiziert werden können. Das darin vorgestellte System verwendet den *Random Forest* Algorithmus zur Klassifizierung von CAN-Nachrichten. Bei einem ersten Versuch konnten damit 85% der Testdaten korrekt klassifiziert und somit die Methoden validiert werden. Das zweite Kapitel hat die Grundlagen, die dafür notwendig sind, vermittelt. Dabei ist besonders die Idee von *Edge Computing* hervorzuheben, welches aufgrund des *Internet of Things* und den immer höheren Datenmengen eine Alternative zu *Cloud Computing* darstellt. Anstatt Sensorwerte in die *Cloud* zur Verarbeitung zu senden, erfolgt sie direkt am Gerät. Zugleich trägt es zum Schutz der Privatsphäre bei, da potentiell persönliche Daten nirgends hin übertragen werden. Aus diesem Grund verfolgt die beschriebene Anwendung das gleiche Konzept. Weiters wurde versucht das ML-Modell durch geschickte Parametrisierung zu optimieren. Es konnte jedoch nur ein Prozent gewonnen werden. Während der weiteren Analyse hat sich eine Unstimmigkeit mit einem CAN-Signal herausgestellt, dass ein Genauigkeitsverlust von 5% bedeutet hat. Das verdeutlicht, dass besonders für die *Machine Learning* Anwendungen eine genaue Begutachtung der Daten essentiell ist.

Im zweiten Teil der Arbeit wurde dargelegt, wie das System in ein Fahrzeug integriert werden kann. Die Vor- und Nachteile einer Architektur mit beziehungsweise ohne *Cloud*-Anbindung sind aufgezeigt worden und der Nutzen von *Edge Computing* in den Kontext gesetzt. Danach ist die Beschreibung der verwendeten *Car Connectivity Unit* und Softwarekomponenten, bestehend aus dem *CAN-Logger*, *Data-Preprocessor*, *ML-Model Trainer*, *ML-Model Predictor* und *Result Presenter* - gefolgt. Es wurden mehrere Ansätze zur Identifizierung besprochen und dabei die *Differenz*-Methode als die sinnvollste erachtet. Der nächste Punkt hätte mehrere Testfahrten mit dem Auto beinhalten sollen. Da jedoch keines zur Verfügung gestellt werden konnte, wurde es mit den Testdaten simuliert. Die Ergebnisse waren mit vier Fahrprofilen sehr vielversprechend, sodass 97% der Fahrten dem korrekten Fahrer zugeordnet wurden.

Zusammenfassend ist eine Fahreridentifikation basierend am Fahrverhalten mit den vorhandenen Daten und existierenden Methoden durchführbar. Eine generelle Aussage über den Einsatz unter realen Bedingungen kann nicht getroffen werden, da zum einen der Feldtest nicht stattgefunden hat und zum anderen auch zu wenig Daten von zu wenig Fahrerinnen vorhanden waren und somit sind sie nicht repräsentativ. Es ist trotzdem vorstellbar, das System für Anwendungen, welche eine limitierte Anzahl an Fahrer involviert und keine unmittelbare Identifizierung benötigt, einzusetzen.

Literaturverzeichnis

[Ama] Amazon machine learning. https://aws.amazon.com/de/machine-learning/, last accessed on 27/02/20. 12

[BLB+13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013. 23, 55

[Cho10] Nian Shong Chok. Pearson's versus spearman's and kendall's correlation coefficients for continuous data. September 2010. 27

[Col13] Andrew Collete. *Python and HDF5*. O'Reilly Media, Inc, USA, 2013. 20

[Con12] Drew Conway. *Machine Learning for Hackers*. O'Reilly Media, Inc, USA, 2012. 3, 46

[CW1] Self-driving cars will create 2 petabytes of data, what are the big data opportunities for the car industry? http://www.computerworlduk.com/news/data/boeing-787s-create-half-terabyte..., last accessed on 27/02/20. 17

[DD18] Li Du and Yuan Du. *Hardware Accelerator Design for Machine Learning*. 09 2018. 11

[EBG18] Saad Ezzini, Ismail Berrada, and Mounir Ghogho. Who is behind the wheel? driver identification and fingerprinting. *Journal of Big Data*, 5(1), feb 2018. 26

[ETKK16] Miro Enev, Alex Takakuwa, Karl Koscher, and Tadayoshi Kohno. Automobile driver fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2016(1):34–50, jan 2016. 4, 20, 21, 26, 48

[Eva11] Dave Evans. How the next evolution of the internet is changing everything. 2011. 16

[fSoAS14] Association for Standardization of Automation and Measuring Systems. Format specification mdf format v. 3.3.1. Standard, Association for Standardization of Automation and Measuring Systems, 2014. 16

[Gah19] Bernhard Gahr. *Towards the Personalized Car - Investigating Driver Identification using CAN Bus Data*. dissertation, University of St. Gallen, 2019. 46

[Gar19] Inc. Gartner. Gartner says 5.8 billion enterprise and automotive iot endpoints will be in use in 2020. Technical report, 2019. 1

[Goo] Google cloud documentation. https://cloud.google.com/docs, last accessed on 27/02/20. 12

[GRDW18] Bernhard Gahr, Benjamin Ryder, Andre Dahlinger, and Felix Wortmann. Driver identification via brake pedal signals — a replication and advancement of existing techniques. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, nov 2018. 4, 5, 45, 49

[GWBV02] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine Learning*, 46(1):389–422, 2002. 31

[HSS+16] David Hallac, Abhijit Sharang, Rainer Stahlmann, Andreas Lamprecht, Markus Huber, Martin Roehder, Rok Sosic, and Jure Leskovec. Driver identification using automobile sensor data from a single turn. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, nov 2016. 4, 5, 49

[KB+12] M Kleine-Budde et al. Socketcan–the official can api of the linux kernel. In *Proceedings of the 13th International CAN Conference (iCC 2012), Hambach Castle, Germany CiA*, pages 05–17, 2012. 38

[KEE+15] Konstantina Kourou, Themis P. Exarchos, Konstantinos P. Exarchos, Michalis V. Karamouzis, and Dimitrios I. Fotiadis. Machine learning applications in cancer prognosis and prediction. *Computational and Structural Biotechnology Journal*, 13:8 – 17, 2015. 5

[MNO+07] C. Miyajima, Y. Nishiwaki, K. Ozawa, T. Wakita, K. Itou, K. Takeda, and F. Itakura. Driver modeling based on driving behavior and its evaluation in driver identification. *Proceedings of the IEEE*, 95(2):427–437, Feb 2007. 4

[MS2] Azure machine learning studio. `https://docs.microsoft.com/en-us/azure/`, last accessed on 27/02/20. 12

[OBD] Obd2-de information. `https://www.obd-2.de/obd-2-allgemeine-infos.html`, last accessed on 16/04/20. 45

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 21

[RRC19] Gopinath Rebala, Ajay Ravi, and Sanjay Churiwala. *An Introduction to Machine Learning*. Springer International Publishing, 2019. 5, 10, 12, 23, 25, 46, 48

[SAE17] E/e diagnostic test modes, Feb 2017. 45

[SBZH07] Carolin Strobl, Anne-Laure Boulesteix, Achim Zeileis, and Torsten Hothorn. Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC Bioinformatics*, 8(1):25, 2007. 29

[SCZ+16] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016. 16, 17

[SIX] Sixt autovermietung. `https://www.sixt.de/`, last accessed on 20/04/20. 48

[Sta] Ergebnisse im Überblick: Familien 1985 - 2018. `https://www.statistik.at/wcm/idc/idcplg?IdcService=GET_PDF_FILE&RevisionSelectionMethod=LatestReleased&dDocName=023079`, last accessed on 11/03/20. 33

[USE20] Plug-n-pwned: Comprehensive vulnerability analysis of obd-ii dongles as a new over-the-air attack surface in automotive iot. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association. 46

[WOM+05] T. Wakita, K. Ozawa, C. Miyajima, K. Igarashi, K. Itou, K. Takeda, and F. Ita-

kura. Driver identification using driving behavior signals. In *Proceedings. 2005 IEEE Intelligent Transportation Systems, 2005.*, pages 396–401, Sep. 2005. 4

[YHQL15] S. Yi, Z. Hao, Z. Qin, and Q. Li. Fog computing: Platform and applications. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 73–78, Nov 2015. 17

[Yoca] Why the yocto project for my iot project? `https://www.embedded.com/why-the-yocto-project-for-my-iot-project/`, last accessed on 21/03/20. 36

[Yocb] Yocto project overview and concepts manual. `https://www.yoctoproject.org/docs/3.0.2/overview-manual/overview-manual.html`, last accessed on 21/03/20. 35

[ZS14] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik - Protokolle, Standards und Softwarearchitektur*. Springer-Verlag, Berlin Heidelberg New York, 2014. 13, 14, 15, 54, 55

Abbildungsverzeichnis

Tabellenverzeichnis

A. Anhang/Ergänzende Information

Im Folgenden werden alle Signale, welche in den original Dateien enthalten sind, gelistet. Zudem ist der Source-Code von den Software-Komponenten (siehe 5.3) abgebildet.

A.1. Signale

Signalname	CAN-Bus	Beschreibung
ESP_VR_Fahrtrichtung	F-CAN	Fahrtrichtung
ESP_HL_Fahrtrichtung	F-CAN	Fahrtrichtung
ESP_HR_Fahrtrichtung	F-CAN	Fahrtrichtung
ESP_VL_Fahrtrichtung	F-CAN	Fahrtrichtung
AB_Gurtschloss_Reihe2_MI	K-CAN	Gurtschloss hinten
AB_Gurtschloss_Reihe2_BF	K-CAN	Gurtschloss hinten
AB_Gurtschloss_FA	K-CAN	Gurtschloss vorne
AB_Gurtschloss_BF	K-CAN	Gurtschloss vorne
AB_Gurtschloss_Reihe2_FA	K-CAN	Gurtschloss hinten
BCM1_Aussen_Temp_ungef	F-CAN	Außentemperatur
BH_Lichthupe	F-CAN	Lichthupe aktiv
Wischer_vorne_aktiv	F-CAN	Wischer vorne aktiv
BH_Fernlicht	F-CAN	Fernlicht aktiv
BH_Blinker_li	F-CAN	Blinker links aktiv
BH_Blinker_re	F-CAN	Blinker rechts aktiv
ESP_v_Signal	F-CAN	Geschwindigkeit
MO_Drehzahl_01	F-CAN	Motordrehzahl
MO_Gangposition	F-CAN	Gangposition
MO_Kuppl_schalter	F-CAN	Kupplung
ESP_Bremsdruck	F-CAN	Bremsdruck
ESP_Fahrer_bremst	F-CAN	Fahrer bremst
ESP_HL_Radgeschw_02	F-CAN	Radgeschwindigkeit
ESP_HR_Radgeschw_02	F-CAN	Radgeschwindigkeit
ESP_VL_Radgeschw_02	F-CAN	Radgeschwindigkeit
ESP_VR_Radgeschw_02	F-CAN	Radgeschwindigkeit
FS_Luftfeuchte_rel	K-CAN	Luftfeuchtigkeit
ESP_Laengsbeschl	F-CAN	Längsbeschleunigung
ESP_Gierrate	F-CAN	Gierrate
ESP_VZ_Gierrate	F-CAN	Vorzeichen Gierrate
ESP_Querbeschleunigung	F-CAN	Querbeschleunigung
KBI_Tankfuellstand_Prozent	F-CAN	Tankfüllstand
LWI_Lenkradwinkel	F-CAN	Lenkradwinkel
LWI_VZ_Lenkradwinkel	F-CAN	Vorzeichen Lenkradwinkel
LWI_VZ_Lenkradw_Geschw	F-CAN	Vorzeichen Lenkradwinkelgeschwindigkeit
LWI_Lenkradw_Geschw	F-CAN	Lenkradwinkelgeschwindigkeit

ND_UTC	K-CAN	Zeit
MO_Fahrpedalrohwert_01	F-CAN	Fahrpedalrohwert
LV_Nebelschlusslicht_Anzeige	K-CAN	Nebelschlusslicht aktiv
LV_Tagfahrlicht_Anzeige	K-CAN	Tagfahrlicht aktiv
LV_Standlicht_Anzeige	K-CAN	Standlicht aktiv
LV_Abblendlicht_Anzeige	K-CAN	Abblendlicht aktiv
LV_Fernlicht_Anzeige	K-CAN	Fernlicht aktiv
LV_Nebellicht_Anzeige	K-CAN	Nebellicht aktiv
KBI_Aussen_Temp_gef	F-CAN	Außentemperatur
KBI_Kilometerstand_2	F-CAN	Kilometerstand

Tabelle A.1.: Signalbeschreibung

A.2. Software Komponenten

CAN-Logger

Da der *CAN-Logger* zum größten Teil aus *Bosch* interner Software besteht, wird dieser hier nicht angeführt.

Data-Preprocessor

```
1  import argparse
2  import pandas as pd
3  import numpy as np
4  import random
5  import os
6  import sys
7  import yaml
8  import time
9  from datetime import datetime
10 from asammdf import MDF
11
12
13 def filter_mdf(file, signals):
14     mdf_file = MDF(file)
15     # Lenkradwinkel
16     lenkradwinkel = mdf_file.get('can0_LWI_Lenkradwinkel')
17     samples = list(lenkradwinkel.samples)
18     vz_samples = list(mdf_file.get('can0_LWI_VZ_Lenkradwinkel').samples)
19     for i in range(0, len(samples)):
20         if not int(vz_samples[i]):
21             samples[i] = samples[i] * -1
22     lenkradwinkel.samples = np.asarray(samples)
23
24     # Lenkradwinkel geschw.
25     lenkrad_gesch = mdf_file.get('can0_LWI_Lenkradw_Geschw')
26     samples = list(lenkrad_gesch.samples)
27     vz_samples = list(mdf_file.get('can0_LWI_VZ_Lenkradw_Geschw').samples)
28     for i in range(0, len(samples)):
29         if not int(vz_samples[i]):
30             samples[i] = samples[i] * -1
31     lenkrad_gesch.samples = np.asarray(samples)
32
33     # Gierrate
34     gierrate = mdf_file.get('can0_ESP_Gierrate')
```

```
35      samples = list(gierrate.samples)
36      vz_samples = list(mdf_file.get('can0_ESP_VZ_Gierrate').samples)
37      for i in range(0, len(samples)):
38          if not int(vz_samples[i]):
39              samples[i] = samples[i] * -1
40      gierrate.samples = np.asarray(samples)
41
42      # filter file with VZ signals
43      filtered_mdf = mdf_file.filter(signals)
44      return filtered_mdf
45
46
47  def resample(data, window_size):
48      column_names = list(data.head())
49
50      data_mean = data.resample('%sL' % window_size).mean()
51      new_column_names = dict()
52      for n in column_names:
53          new_column_names[n] = '%s_mean' % n
54      data_mean.rename(columns=new_column_names, inplace=True)
55
56      data_std = data.resample('%sL' % window_size).std()
57      new_column_names = dict()
58      for n in column_names:
59          new_column_names[n] = '%s_std' % n
60      data_std.rename(columns=new_column_names, inplace=True)
61
62      data_min = data.resample('%sL' % window_size).min()
63      new_column_names = dict()
64      for n in column_names:
65          new_column_names[n] = '%s_min' % n
66      data_min.rename(columns=new_column_names, inplace=True)
67
68      data_max = data.resample('%sL' % window_size).max()
69      new_column_names = dict()
70      for n in column_names:
71          new_column_names[n] = '%s_max' % n
72      data_max.rename(columns=new_column_names, inplace=True)
73
74      data_median = data.resample('%sL' % window_size).median()
75      new_column_names = dict()
76      for n in column_names:
77          new_column_names[n] = '%s_median' % n
78      data_median.rename(columns=new_column_names, inplace=True)
79
80      data = pd.concat([data_mean, data_max, data_median, data_min, data_std],
81                       axis=1,
82                       sort=False)
83      data.dropna(inplace=True)
84      return data
85
86
87  def mdf_to_df(mdf_file, timestamp):
88      data = mdf_file.to_dataframe()
89      data.index = (data.index * 1000000000 + float(timestamp))
90      data.index = data.index.values.astype('datetime64[ns]')
91      data.index = pd.to_datetime(data.index)
92      return data
93
94
95  def main():
```

```
96
97      my_parser = argparse.ArgumentParser(
98          description='Preprocesses mdf data and stores it in hdf5 file')
99      my_parser.add_argument('-c',
100                             '--config',
101                             action='store',
102                             metavar='config',
103                             type=str,
104                             required=True,
105                             help='Configuration file')
106
107     my_parser.add_argument('-r',
108                             action='store',
109                             metavar='reset',
110                             type=bool,
111                             default=False,
112                             help='Remove files from output dir')
113
114     args = my_parser.parse_args()
115     config_file = args.config
116     reset = args.reset
117
118     print('Starting data-preprocessor.py')
119
120     config = dict()
121     with open(config_file, 'r') as stream:
122         try:
123             config = yaml.safe_load(stream)
124         except yaml.YAMLError as exc:
125             print(exc)
126
127     if not os.path.isdir(config['measured_output']):
128         print('Measured directory does not exist')
129         exit(1)
130
131     if not os.path.isdir(config['output_dir']):
132         print('Output directory does not exist')
133         exit(1)
134
135     if reset:
136         os.system('rm -rf %s/*', config['output_dir'])
137         os.system('rm -rf %s/*', config['measured_output'])
138
139     while True:
140         for file in os.listdir(config['measured_output']):
141             print('New data available...')
142             if not file.endswith('.mf4'):
143                 continue
144             timestamp = file.split('_')[0]
145             mdf_file = filter_mdf(
146                 os.path.join(config['measured_output'], file),
147                 config['signals'])
148             df = mdf_to_df(mdf_file, timestamp)
149             df = resample(df, config['window_size'])
150             df = df[df['can0_ESP_v_Signal_max'] != 0]
151             outfile = os.path.join(config['output_dir'], timestamp, '.hdf')
152             df.to_hdf(outfile, 'driverX', append=True)
153
154             print('Processed %s' % outfile)
155             # move file to backup
156             os.rename(os.path.join(config['measured_output'], file),
```

```
157                         os.path.join(config['backup_output'], file))
158
159         time.sleep(5)
160
161
162 main()
```

Listing A.1: Ausschnitt Fahreridentifikation

ML-Model Trainer, ML-Model Predictor, Result Presenter

```
1  import argparse
2  import pathlib
3  import pandas as pd
4  import os
5  import sys
6  import h5py
7  import yaml
8  import numpy as np
9  import random
10 import time
11 from sklearn.utils import shuffle
12 from sklearn.model_selection import train_test_split
13 from sklearn.ensemble import RandomForestClassifier
14
15
16 def send_SMS(message, number):
17     os.system('./sendSMS "%s" %s' % (message, number))
18
19
20 def main():
21
22     my_parser = argparse.ArgumentParser(
23         description='Random forest classifier for driver identification')
24     my_parser.add_argument('-c',
25                            '--config',
26                            action='store',
27                            metavar='config',
28                            type=str,
29                            required=True,
30                            help='Config file for rf parameter')
31
32     my_parser.add_argument('-s',
33                            '--simulation',
34                            action='store',
35                            metavar='simulation',
36                            type=bool,
37                            default=False,
38                            help='Enables simulation mode')
39
40     args = my_parser.parse_args()
41     config_file = args.config
42     simulation = args.simulation
43
44     print('Starting random-forest.py')
45
46     config = dict()
47     with open(config_file, 'r') as stream:
48         try:
```

```
49              config = yaml.safe_load(stream)
50          except yaml.YAMLError as exc:
51              print(exc)
52
53      if not os.path.isdir(config['input_dir']):
54          print('Input directory does not exist')
55          exit(1)
56
57      if not os.path.isdir(config['profile_dir']):
58          print('Directory with profiles does not exist')
59          exit(1)
60
61      if not os.path.isdir(config['sim_profile_dir']):
62          print('Directory with simulation profiles does not exist')
63          exit(1)
64
65      # read driver profiles
66      prfile_dir = config['profile_dir']
67      if simulation:
68          prfile_dir = config['sim_profile_dir']
69
70      frames = []
71      for file in os.listdir(prfile_dir):
72          if not file.endswith('.hdf'):
73              continue
74
75          data = pd.read_hdf(os.path.join(prfile_dir, file))
76          frames.append(data)
77
78      print('Loaded driver profiles')
79      profiles = pd.concat(frames, sort=False)
80      features = profiles.columns
81      features.drop('class')
82      X = profiles[features]
83      Y = profiles['class']
84
85      X = X.values
86      X = np.nan_to_num(X)
87      Y = Y.squeeze()
88      Y = Y.astype(int)
89
90      # Train RF Classifier
91      clf = RandomForestClassifier(n_estimators=config['n_estimators'],
92                                   n_jobs=-1,
93                                   random_state=1,
94                                   min_samples_leaf=config['min_samples_leaf'
                                         ],
95                                   criterion=config['criterion'],
96                                   max_depth=config['max_depth'])
97      clf.fit(X, Y)
98      print('Created and trained ML Model')
99
100     all_data = []
101     while True:
102         data = []
103         for file in os.listdir(config['input_dir']):
104             if not file.endswith('.hdf'):
105                 continue
106
107             time.sleep(1)
108             df = pd.read_hdf(os.path.join(config['input_dir'], file))
```

```
109             data.append(df)
110
111             # move file to backup
112             os.rename(os.path.join(config['input_dir'], file),
113                       os.path.join(config['backup_dir'], file))
114
115         if len(data) == 0:
116             continue
117
118         print('New data available...')
119         all_data = all_data + data
120
121         pdata = pd.concat(all_data, sort=False)
122         pdata = pdata.values
123
124         print('Predicting...')
125         pred = clf.predict(pdata)
126
127         print('Result:\n')
128         ids, counts = np.unique(pred, return_counts=True)
129         ids = dict(zip(ids, counts))
130         first = {'id': None, 'count': 0}
131         second = {'id': None, 'count': 0}
132         for id in t:
133             c = ids[id]
134             if c > first['count']:
135                 first['count'] = c
136                 first['id'] = id
137             elif c > second['count']:
138                 second['count'] = c
139                 second['id'] = id
140
141         first_conf = first['count'] / len(pred)
142         second_conf = second['count'] / len(pred)
143         diff = first_conf - second_conf
144         if diff >= 0.30 and first_conf >= 0.50:
145             if confirmed_id == first['id']:
146                 confirmation_count += 1
147             else:
148                 confirmed_id = first['id']
149                 confirmation_count = 1
150         if confirmation_count == config['confirmation_count']:
151             send_SMS('\nDriver with id %s is driving' % first['id'],
152                      config['number'])
153             break
154
155         time.sleep(5)
156
157
158 main()
```

Listing A.2: Ausschnitt Fahreridentifikation