HOCHSCHULE COBURG

Hochschule für angewandte Wissenschaften Coburg

Fakultät Elektrotechnik und Informatik

Studiengang: Informatik-Bachelor

Bachelorarbeit

Klassifikation von Verkehrsteilnehmern auf Basis realer Positionszeitreihen mit Verfahren des maschinellen Lernens

Lennart Köpper

Abgabe der Arbeit: 10.09.2023

Betreut durch:

Prof. Dr. Thomas Wieland, Hochschule Coburg

Abstract (Deutsch)

Im Kontext des autonomen Fahrens ist die Wahrnehmung des Fahrzeugumfeldes, einschließlich anderer Verkehrsteilnehmer, von entscheidender Bedeutung. Diese erfolgt üblicherweise mit Hilfe verschiedener Sensorsysteme, welche jedoch auf das unmittelbar Fahrzeugumfeld beschränkt sind. Als möglicher Ansatz zur Entwicklung einer erweiterten Umfeldwahrnehmung untersucht diese Arbeit die Klassifikation von Verkehrsteilnehmern auf Basis hochfrequenter (1 Hertz) Positionssequenzen, welche unter Realbedingungen im Straßenverkehr über das *Global Navigation Satellite System* (GNSS) aufgezeichnet wurden. Hierbei werden zunächst vier Typen von Verkehrsteilnehmern unterschieden: *Fußgänger*, *Fahrräder*, *Motorräder* und *Autos*. In Vorbereitung auf die Klassifikation wird im Zuge der Arbeit aufgezeigt, wie aus den reinen Positionssequenzen relevante Bewegungsmerkmale (Geschwindigkeit, Beschleunigung etc.) abgeleitet werden können. Außerdem werden verschiedene Möglichkeiten untersucht, wie Verrauschungsfehler und Ausreißer in den GNSS-Sequenzen bereinigt bzw. abgemildert werden können. Hierzu gehören bspw. eine nachträgliche Erhöhung des Aufnahmeintervalls und das Angleichen der GNSS-Sequenzen an die Daten des *OpenStreetMap*-Projektes mittels *Map-Matching*. Die Wahl des primär in dieser Arbeit genutzten Klassifikationsverfahrens fällt auf *rekurrente neuronale Netze* (RNNs) mit *Long-Short-Term-Memory-Zellen*, welche unter den Verfahren des maschinellen Lernens den heutigen Standard für die Verarbeitung sequenzieller Daten setzen. Es wird eine mögliche Architektur für RNNs zur Klassifikation von Verkehrsteilnehmern auf Basis ausgewählter sequenzieller Bewegungsmerkmale vorgestellt und untersucht, wie die Länge der Sequenzen die Klassifikationsqualität beeinflusst. Die Ergebnisse der Arbeit zeigen, dass RNNs dazu in der Lage sind, Verkehrsteilnehmer und insbesondere auch verschiedene motorisierte Fahrzeuge auf Basis zwei- bis vierminütiger Sequenzen mit einem verdoppelten Aufnahmeintervall gut zu klassifizieren. Ein Map-Matching der Sequenzen führt dabei im Allgemeinen zu keiner Verbesserung der Klassifikationsqualität.

Abstract (English)

In the field of autonomous driving, perceiving the vehicle's environment, including other road users, is of crucial importance. This is typically achieved using various sensors systems, which, however, are limited to the line of sight of the vehicle. As a possible approach to develop some form of extended environmental perception, this bachelor thesis aims to investigate road user classification based on high-frequency (1 hertz) positional sequences, that were recorded under real-world conditions via the *Global Navigation Satellite System* (GNSS). As a starting point, there are four types of road users to be distinguished, namely*: pedestrians*, *bicycles*, *motorcycles* and *cars*. In preparation for the classification, this work demonstrates how relevant motion-based features (velocity, acceleration, etc.) can be derived from the raw positional sequences. Furthermore, various methods are examined to clean up or at least reduce inaccuracies like noise and outliers in the GNSS-sequences. This includes, for example, a subsequent increase of the recording interval and the adjustment of the GNSS-sequences according to *OpenStreetMap* data using *Map-Matching*. The classification method of choice, which is primarily used in this work, are *Long Short-Term Memory recurrent neural networks* (RNNs) that set the current state of the art in Machine Learning for processing sequential data. An RNN architecture for road user classification based on selected sequential motion-based features is presented and the influence of sequence length on classification quality is examined. The results of the work show that RNNs are capable of efficiently classifying road users, and especially different motorized vehicles, based on two- to four-minute sequences with a doubled sampling interval. In general, map matching the sequences does not affect classification quality at all.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Programmcodeverzeichnis

Symbolverzeichnis

Symbol	Bedeutung
b	Bias-Term eines künstlichen Neurons
C	Regularisierungsparameter zur Bestrafung von Fehlklassifikationen beim Training von SVMs
d_{max}	Maximale Tiefe eines Decision Trees
$f_{max}(x)$	Funktion zur Bestimmung der maximalen Anzahl an Merkmalen, die für das Aufteilen eines Knotens in einem Decision Tree in Betracht gezogen werden
FN/ FN_k	Anzahl falsch negativer Vorhersagen eines Klassifikators, ggf. bezogen auf die Klasse k
FP/ FP_k	Anzahl falsch positiver Vorhersagen eines Klassifikators, ggf. bezogen auf die Klasse k
h	In Abhängigkeit des Kontexts: a) Ausgabe eines künstlichen Neurons b) Interner Zustand einer Gedächtniszelle
K	Anzahl der Klassen für ein gegebenes Klassifikationsproblem
$K(x, y)$	Kernel-Funktion einer SVM
l_{lstm}	Anzahl der LSTM-Schichten eines RNNs auf Basis der in 8.2.1 vorgestellten Basisarchitektur
l_{pre}	Anzahl vorbereitender Schichten eines RNNs auf Basis der in 8.2.1 vorgestellten Basisarchitektur
l_{post}	Anzahl nachbereitender Schichten eines RNNs auf Basis der in 8.2.1 vorgestellten Basisarchitektur
$n_{Neurons}$	Anzahl der Neuronen pro Schicht in einem RNN auf Basis der in 8.2.1 vorgestellten Basisarchitektur
n_{Trees}	Anzahl der Decision Trees in einem Random Forest
RN / RN_k	Anzahl richtig negativer Vorhersagen eines Klassifikators, ggf. bezogen auf die Klasse k
RP/ RP_k	Anzahl richtig positiver Vorhersagen eines Klassifikators, ggf. bezogen auf die Klasse k
s_{min}^{leaf}	Minimale Anzahl an Trainingsdatenpunkten, die durch ein Blatt eines Decision Trees abgedeckt werden müssen
s_{max}	Anteil der Trainingsdatenpunkte, der beim Training eines Random Forests für das Training der einzelnen Decision Trees zur Verfügung steht
s_{min}^{split}	Minimale Anzahl an Trainingsdatenpunkten, die ein Knoten eines Decision

Symbol	Bedeutung
	Trees abdecken muss, um weiter aufgeteilt werden zu dürfen
$\boldsymbol{w}$	Gewichtsvektor eines künstlichen Neurons
$\boldsymbol{x}$	Eingabevektor eines künstlichen Neurons oder KNNs
z	Gewichtete Summe eines künstlichen Neurons
$\boldsymbol{z}$	Vektor der gewichteten Summen aller Ausgabeneuronen eines KNNs
γ	Regularisierungsparameter des RBF-Kernels für SVMs.
$\varphi(z)$	Aktivierungsfunktion eines künstlichen Neurons

Abkürzungsverzeichnis

4G	Vierte Generation des Mobilfunks
5G	Fünfte Generation des Mobilfunks
API	Application Programming Interface
CPU	Central Processing Unit
DQLN	Deep-Q-Learning-Netz
FFN	Feed-Forward-Netz
GNSS	Global Navigation Satellite System
HMM	Hidden Markov Model
HTTP	Hypertext Transfer Protocol
IZK	Innovations-Zentrum Kronach e.V.
JSON	JavaScript Object Notation
KNN	Künstliches neuronales Netz
LIDAR	Light Detection and Ranging
LKW	Lastkraftwagen
LSTM	Long Short-Term Memory
PKW	Personenkraftwagen
RADAR	Radio Detection and Ranging
RAM	Random Access Memory
RBF	Radial Basis Function
ReLU	Rectified Linear Unit
RNN	rekurrentes neuronales Netz
STS	Sequence to Sequence
STV	Sequence to Vector
SVM	Support Vector Machine

1 Einleitung

1.1 Kontext und Projekthintergrund

Schon heute besitzen Mobilität und Konnektivität einen großen Einfluss auf den Alltag und die Lebensqualität vieler Menschen. Zwei bedeutende Trends, die im Zusammenhang mit diesen beiden Begriffen immer wieder Erwähnung finden, sind das *autonome Fahren* und *5G*. Dass diese beiden Trends eng miteinander in Verbindung stehen, legt ein Bericht des Beratungsunternehmens Gartner nahe, nach welchem der Ausbau von 5G-Netzwerken von großer Bedeutung für die weiteren Entwicklungen des autonomen Fahrens ist. Der wichtigste Grund hierfür ist demnach die immer weiter steigende Menge von Fahrzeug- und Sensordaten, welche durch zunehmend autonomisierte Fahrzeuge nicht nur verarbeitet, sondern auch zeitkritisch mit anderen Akteuren geteilt werden muss. Hierdurch ist davon auszugehen, dass 5G durch die bis zu zehnfach höheren Übertragungsraten (im Vergleich mit dem heutigen 4G-LTE) in Zukunft nicht nur Einfluss auf die allgemeine Konnektivität und das Infotainment von Fahrzeugen nehmen, sondern auch ganz konkret zur weiteren Steigerung der Sicherheit im autonomen Fahren beitragen wird [1].

In diesem Kontext wurde das Projekt *5GKC* ins Leben gerufen. Das Ziel dieses Projektes besteht darin, in der oberfränkischen Stadt Kronach ein 5G-basiertes Testfeld zu schaffen, in welchem verschiedene Anwendungen von 5G im autonomen Fahren erforscht, entwickelt und im öffentlichen Verkehr erprobt werden können. Der wesentliche Fokus liegt dabei auf den Bereichen Steuerung, Überwachung, Mensch-Fahrzeug-Interaktion und Kommunikation. Kooperationspartner des Projektes ist dabei neben dem *Fraunhofer-Institut für integrierte Schaltungen*, dem *Innovations-Zentrum Region Kronach e.V.* und *Valeo*, dem Weltmarktführer für Fahrerassistenzsysteme, auch die *Hochschule für angewandte Wissenschaften Coburg*. Letztere dient als wichtiger Impulsgeber und Partner bei verschiedenen Forschungsprojekten. Zu diesen zählt auch ein Projekt zur *erweiterten Umfeldwahrnehmung autonomer Fahrzeuge*, welchem die vorliegende Arbeit zuzuordnen ist [2].

1.2 Motivation und Zielsetzung

Im Bereich des autonomen Fahrens und der Fahrerassistenzsysteme spielt die Wahrnehmung des Fahrzeugumfeldes, zu welchem neben der unbelebten Umgebung eines Fahrzeuges vor allem auch andere Verkehrsteilnehmer gehören, eine entscheidende Rolle. Üblicherweise

kommen hierfür verschiedenste Sensorsysteme, wie *Ultraschall-*, *RADAR-* und *LIDAR*-Sensoren, sowie kamerabasierte Verfahren, wie das *Stereo-* und das *maschinelle Sehen* zum Einsatz, welche allesamt spezifische Vor- und Nachteile mit sich bringen. Die Vorteile können hierbei durch die sogenannte *Sensordatenfusion*, also durch die Kombination von Informationen aus verschieden Sensorsystemen, oft gut zusammengeführt werden, was auch häufig zum Ausgleich spezifischer Nachteile einzelner Sensoren führt [3, S. 440]. Ein wesentlicher Nachteil bleibt bei den heute verbreiteten Systemen jedoch in jedem Fall bestehen, denn alle beschränken sich auf ein stark limitiertes Umfeld des Fahrzeuges, welches im Fall von kamerabasierten Verfahren bspw. auf das Sichtfeld der Kameras begrenzt ist. Dieses Sichtfeld kann insbesondere im städtischen Verkehr sehr stark eingeschränkt sein, woraus sich ein Bedarf nach Systemen begründet, welche auch zur erweiterten Umfeldwahrnehmung eingesetzt werden können, um die bestehenden Systeme zu ergänzen und das autonome Fahren somit sicherer zu machen.

Im Zuge dieser Bachelorarbeit soll hierbei die Möglichkeit erforscht werden, Verkehrsteilnehmer auf Basis von sequenziell bereitgestellten Positionsdaten durch den Einsatz *maschineller Lernverfahren* in verschiedene Typen zu klassifizieren. Hierbei soll die Klassifikation möglichst realitätsnah, also insbesondere unter der Nutzung von realen Positionssequenzen aus dem Straßenverkehr, umgesetzt werden. Aus vorangegangenen Forschungsarbeiten ist dabei bekannt, dass reale Positionsdaten, die über das *Global Navigation Satellite System* (GNSS) ermittelt werden, häufig ungenau und zu einem gewissen Grad verrauscht sind, weshalb im Verlauf dieser Arbeit zunächst die folgende Frage untersucht wird:

Wie können reale Positionsdaten, die Ungenauigkeiten und Rauschen aufweisen, so vorverarbeitet werden, dass sie sich gut für den Einsatz maschineller Lernverfahren eignen?

Im Anschluss daran ergibt sich dann die primäre Forschungsfrage der Arbeit:

Welche Verfahren des maschinellen Lernens sind für die Klassifikation von Verkehrsteilnehmern auf Basis von sequenziellen Positionsdaten geeignet?

Sollten die Ergebnisse dieser Arbeit als positiv zu bewerten sein, ließe sich darauf basierend ein System entwickeln, welches in der Lage ist, Informationen zu anderen Verkehrsteilnehmern abzuleiten, während sich diese noch deutlich außerhalb der Reichweite der bisher üblichen Sensorsysteme von autonomen Fahrzeugen befinden.

1.3 Aufbau der Arbeit

Um die im letzten Abschnitt formulierten Forschungsfragen zu beantworten, soll im nachfolgenden Kapitel 2 *Theoretischer Hintergrund* zunächst auf die wesentlichen Grundlagen eingegangen werden, die für ein Verständnis der weiteren Arbeit notwendig sind. Anschließend wird im Kapitel 3 *Verwandte Arbeiten* auf den bisherigen Stand der Forschung Bezug genommen. Es wird aufgezeigt welche verwandten Ansätze bei der Klassifikation von Verkehrsteilnehmern aus Positionsdaten bereits existieren und worin sich diese von dem Ansatz der vorliegenden Arbeit unterscheiden. In Kapitel 4 *Datengrundlage* wird die Datenbasis vorgestellt, welche die Grundlage für die in dieser Arbeit umgesetzten Klassifikationen bildet. In Kapitel 5 *Anforderungen und Gesamtkonzept* folgt basierend darauf die Vorstellung des ausgearbeiteten Konzeptes zur Untersuchung der Forschungsfragen und Umsetzung der Klassifikation. Die darauffolgenden Kapitel 6 *Datensatzgenerierung und -erweiterung*, 7 *Umsetzung des Map-Matchings* und 8 *Umsetzung der Klassifikation* beschreiben die Realisierung der vielversprechendsten Verfahren zur Datenvorverarbeitung bzw. zur Klassifikation der Positionssequenzen. Die Ergebnisse werden im Anschluss daran im Kapitel 9 *Evaluierung und Diskussion* auf Basis unabhängiger Testdaten miteinander verglichen und hinsichtlich ihrer Qualität bewertet. Den Abschluss bildet das Kapitel 10 *Zusammenfassung und Ausblick*, in welchem die wichtigsten Erkenntnisse der Arbeit zusammengefasst werden und auf weiteres Forschungspotenzial hingewiesen wird.

2 Theoretischer Hintergrund

2.1 Global Navigation Satellite System

Global Navigation Satellite System (GNSS; dt. globales Navigationssatellitensystem) ist der Sammelbegriff für die zahlreichen satellitengestützten Systeme, welche für den Zweck der Navigation und Standortbestimmung eingesetzt werden. Hierzu zählen neben den global verfügbaren Systemen, wie dem *Global Positioning System* (GPS) der Vereinigten Staaten, dem *GLONASS* der russischen Föderation, dem *Galileo*-System der Europäischen Union und dem *BeiDou*-System der Volksrepublik China auch regionale Unterstützungssysteme [4, S. 2].

Die Standortbestimmung basiert bei allen Systemen auf dem sogenannten *Time of Arrival Ranging*. Hierbei werden ausgehend von den Satelliten Signale (*Ranging Codes)* gesendet, welche die aktuelle Position des sendenden Satelliten und den Sendezeitpunkt umfassen. Nachdem die Signale durch ein GNSS-fähiges Gerät empfangen wurden, werden die Signallaufzeiten errechnet. Diese werden im Anschluss mit der Signalgeschwindigkeit multipliziert, um die Entfernung zu den Satelliten zu bestimmen [4, S. 19ff.]. Ist die Entfernung zu vier Satelliten bekannt, kann daraus, wie in Abb. 1 veranschaulicht, der Standort des empfangenden Geräts in Form von Koordinaten und Höhe ermittelt werden. Theoretisch wären dabei drei Satelliten ausreichend, wenn davon ausgegangen werden kann, dass die Uhren der Satelliten und des Empfängers perfekt synchronisiert sind, was in der Praxis jedoch meist nicht der Fall ist [4, S. 2].

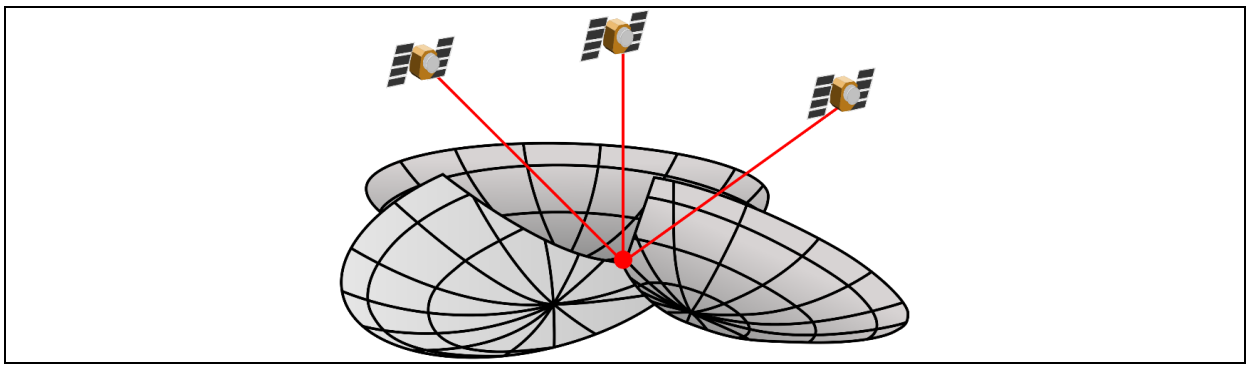

Abb. 1: Ermittlung der GNSS-Position als Schnittpunkt mehrerer Kugeln
Quelle: Entnommen aus [5]

Die Genauigkeit der Positionsbestimmung mittels GNSS ist durch viele Faktoren abhängig. Bspw. sollten die Satelliten für eine genaue Bestimmung in Relation zum Empfänger möglichst gleichmäßig verteilt sein. Sind die genutzten Satelliten eng beisammen oder ungleichmäßig

verteilt, kann die Positionsermittlung ungenau werden [6]. Hinzu kommen weitere Fehlerquellen, wie bspw. Signalbrechungen in der Atmosphäre oder die Reflektion von Signalen an der Erdoberfläche oder an Gebäuden. Letztere können das Signal einzelner Satelliten auch vollständig blockieren, und hiermit die Auswahl der Satelliten zur Standortbestimmung einschränken. Einige dieser Fehlerfaktoren, insbesondere die atmosphärisch bedingten, können durch satellitenbasierte Ergänzungssysteme wie dem *European Geostationary Navigation Overlay Service* korrigiert werden. Diese senden über geostationäre Satelliten Korrektursignale, welche bei der Berechnung des Standortes miteinbezogen werden [7]. Hierdurch sind durchschnittliche Genauigkeiten von etwa 5-15 Metern erreichbar [6].

2.2 Map-Matching

Wie im letzten Abschnitt bereits erwähnt, sind Positionen und somit auch Positionssequenzen, welche über das GNSS ermittelt werden im Regelfall fehlerbelastet. Zum einen durch Abtastfehler, also den Verlust von Information zwischen zwei aufgenommenen Koordinatenpunkten, und zum anderen durch Messungenauigkeiten. Diese Ungenauigkeiten zu ignorieren kann potenziell zu falschen Analysen und entsprechend auch zu falschen Schlussfolgerungen in den Klassifikationsmodellen führen. *Map-Matching* ist ein möglicher Ansatz derartige Fehler zu minimeren. Gemeint ist hiermit das Abbilden von aufgenommenen Positionssequenzen auf eine digitale Repräsentation eines Straßen- und Wegenetzes, das üblicherweise in Form eines *Graphen* vorliegt [8]. Abb. 2 zeigt ein Beispiel für ein solches Map-Matching.

Abb. 2: Map-Matching von aufgenommenen GNSS-Punkten auf das Straßennetz
Quelle: Entnommen aus [8]

Beim Versuch eine Positionssequenz auf ein Straßen- und Wegenetz zu projizieren, handelt es sich im Grunde um ein Suchproblem. Gesucht wird diejenige Abfolge an Straßensegmenten,

welche am wahrscheinlichsten zu der gegebenen Sequenz an GNSS-Punkten passt. Hierfür nutzen die meisten modernen Map-Matching-Services einen komplexen Algorithmus, der auf einem *Hidden Markov Model* (HMM*)* basiert und im Folgenden vereinfacht erläutert wird. Im ersten Schritt werden, wie in Abb. 3 veranschaulicht, für alle GNSS-Punkte passende Straßensegmente, sogenannte *Kandidaten*, ermittelt. Diese werden in Abhängigkeit verschiedener Parameter in einem gewissen Umkreis um die Punkte herum bestimmt [8].

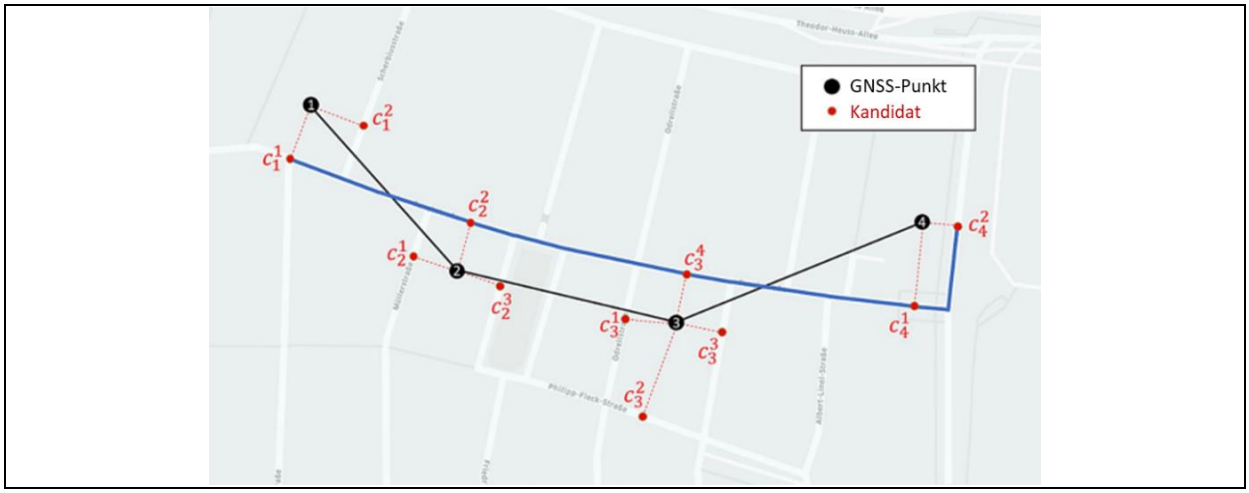

Abb. 3: Koordinatenpunkte (GNSS-Messung) und zugehörige Kandidaten
Quelle: Entnommen aus [8]

Im nächsten Schritt wird das HMM eingesetzt. Dabei werden die gegebenen GNSS-Punkte als *beobachtbare Zustände* (*measurements*) betrachtet. Die zugehörigen Kandidaten bilden die Menge der *unbeobachtbaren Zustände* (*hidden states*). Gesucht wird nun die wahrscheinlichste Sequenz an Kandidaten, als Folge von Zuständen mit den höchsten Übergangswahrscheinlichkeiten, wobei die möglichen Zustandsübergänge durch die gegebenen Verbindungen des Straßennetzwerks definiert sind. Bei der Berechnung dieser Übergangswahrscheinlichkeiten kommen verschiedene Metriken zum Einsatz, die unter anderem dazu führen, dass Kandidaten, welche sich näher an den aufgezeichneten Punkten befinden, als wahrscheinlicher angenommen werden. Außerdem werden Folgen von Kandidaten bevorzugt, die möglichst wenig komplexe Bewegungs-Manöver voraussetzen. Wurden die Übergangswahrscheinlichkeiten berechnet, so definieren diese einen Suchraum, in welchen nun der optimale Pfad mit Hilfe eines Suchalgorithmus bestimmt wird. Abb. 4 zeigt alle möglichen Übergänge zwischen den Kandidaten der in Abb. 3 gegebenen Positionssequenz. Der durch den Suchalgorithmus gefundene (optimale) Pfad und die zugehörigen Kandidaten sind hervorgehoben [8].

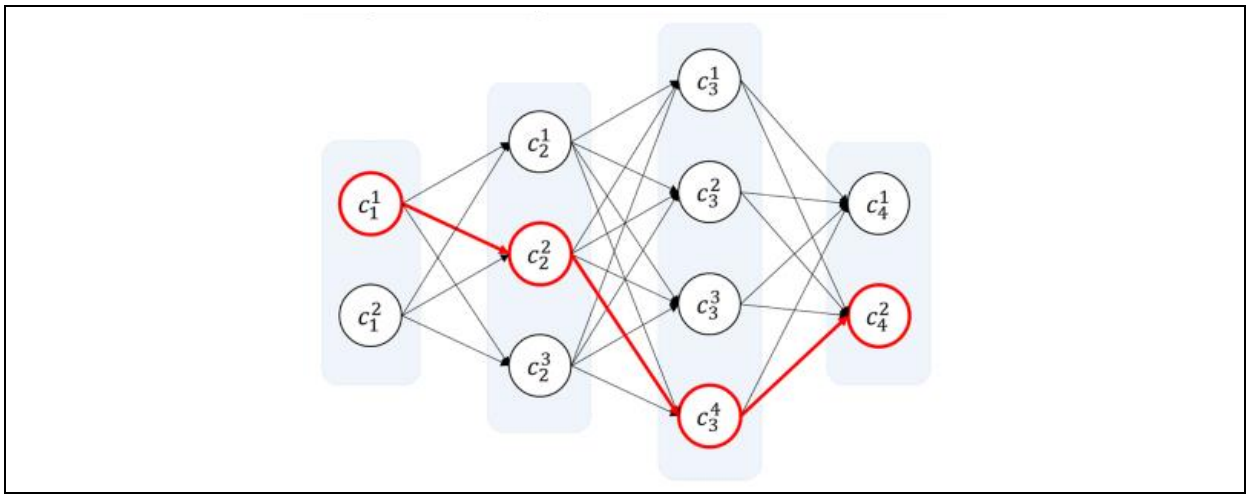

Abb. 4: Mögliche Zustandsübergänge und optimale Kandidaten
Quelle: Entnommen aus [8]

Ein großer Vorteil der HMM-basierten Services ist, dass sie sich sowohl für den Einsatz in *Online*- als auch in *Offline-Szenarien* eignen [9]. Unter Ersterem versteht man das Map-Matching unter Echtzeit-Bedingungen. Das heißt die Punkte werden bereits während der Aufnahme auf das Straßennetz abgebildet, wobei natürlich nur die bisher bekannten Punkte genutzt werden können, um die wahrscheinlichsten Kandidaten für den aktuellen Punkt zu ermitteln. Im Gegensatz dazu können in Offline-Szenarien auch alle nachfolgenden Punkte miteinbezogen werden, da das Map-Matching erst dann stattfindet, wenn die Aufnahme abgeschlossen wurde [8].

In Bezug auf die zugrundeliegenden Positionssequenzen zeigt eine Untersuchung von Pingfu Chao et al. [9], dass die Aufnahmerate einen wesentlichen Einfluss auf die Qualität des Map-Matchings besitzt. Demnach führen nicht nur zu niedrige, sondern auch zu hohe Aufnahmeraten ab einem Hertz zu potenziell schlechten Resultaten.

2.3 Maschinelles Lernen und Klassifikation

Eine weit verbreitete Definition des Begriffs *maschinelles Lernen* (engl. *Machine Learning*) wurde 1959 durch Arthur Lee Samuel, einem amerikanischen Pionier auf dem Gebiet der Computerspiele und künstlichen Intelligenz, geprägt [10]. Er definierte das maschinelle Lernen als Fachgebiet, welches Computern die Fähigkeit verleiht, Probleme zu lösen, ohne explizit dafür programmiert zu werden. Um dies zu erreichen, werden beim maschinellen Lernen verschiedene Algorithmen eingesetzt, welche dazu in der Lage sind, sich im Hinblick auf eine gegebene Aufgabe selbstständig zu verbessern, indem sie durch die Verarbeitung von Beispieldaten Erfahrungen sammeln. Die Menge der verwendeten Beispieldaten nennt man dabei

Trainingsdatensatz. Ein einzelnes Beispiel wird als *Trainingsdatenpunkt* bezeichnet. Entsprechend nennt man den Lernprozess *Training*. [11, S. 4] Die meisten Algorithmen sind modellbasiert, das heißt, sie stützen sich auf – je nach Algorithmus und gewählten *Hyperparametern* mehr oder weniger komplexe – mathematische Modelle, deren *Modellparameter* im Zuge des Trainings so angepasst werden, dass sie im besten Fall Muster in den Daten widerspiegeln. Das trainierte *Modell* kann anschließend auf die konkrete Aufgabenstellung angewandt werden, wobei es dazu in der Lage ist, auch mit neuen Datenpunkten umzugehen [11, S. 21ff.].

Maschinelles Lernen kommt meist dann zum Einsatz, wenn die konventionellen Methoden der Informatik an ihre praktischen Grenzen stoßen. Dies ist insbesondere der Fall, wenn Aufgabenstellungen vorliegen, deren Lösungen nur durch einen sehr komplexen Regelsatz realisiert werden können. Dann ist es oftmals einfacher zu versuchen, über große Datenmengen und maschinelle Lernalgorithmen Erkenntnisse zu gewinnen oder gar Problemlösungen zu finden [11, S. 6]. Eine für das maschinelle Lernen typische Problemstellung ist hierbei die *Klassifikation*. In diesem Kontext versteht man unter einem Klassifikationsproblem ein Vorhersageproblem, bei welchem die gesuchte Vorhersage verschiedene Klassen repräsentiert [12]. Vereinfacht beschrieben werden bei einer Klassifikation also Objekte in verschiedene Kategorien eingeteilt. Im Zuge dieser Arbeit sollen bspw. Verkehrsteilnehmer in verschiedene Typen, wie *Fußgänger*, *Fahrradfahrer* oder *Autos*, eingeteilt werden.

Algorithmen und Modelle, welche für die Vorhersage von Klassen eingesetzt werden können, nennt man *Klassifikationsverfahren* bzw. *Klassifikatoren*. Beim Training werden den Klassifikationsverfahren zusätzlich zu den Trainingsdatenpunkten auch die zugehörigen Lösungen, also die vorherzusagenden Klassen, auch *Labels* genannt, mitgegeben. Da die Algorithmen somit während des Lernens angeleitet werden, gehören Klassifikationsverfahren im Allgemeinen dem Teilbereich des *Überwachten Lernens* (engl. *Supervised Learning*) an. Neben diesem unterscheidet man noch die Bereiche *Halbüberwachtes* (engl. *Semi-Supervised*), *Unüberwachtes* (engl. *Unsupervised*) und *Bestärkendes Lernen* (engl. *Reinforcement Learning*) [11, S. 9ff.]. Diese sind jedoch für die vorliegende Arbeit nicht von Relevanz.

2.4 Eingesetzte Klassifikationsverfahren

Es existieren zahlreiche maschinelle Lernverfahren, welche sich für den Einsatz als Klassifikationsverfahren eignen. Im Folgenden soll lediglich auf die Klassifikationsverfahren eingegangen werden, welche im Zuge der praktischen Umsetzung dieser Arbeit zum Einsatz kommen:

Support Vector Machine und *Random Forest*. Da das Random-Forest-Verfahren dabei auf sogenannten *Decision Trees* basiert, werden auch diese kurz erläutert. Der Klassifikation mit Hilfe neuronaler Netze wird ein eigener Abschnitt dieser Arbeit gewidmet.

2.4.1 Support Vector Machine

Die *Support Vector Machine* (SVM) ist ein weit verbreitetes, mächtiges und flexibles maschinelles Lernverfahren, welches sich hervorragend für die binäre Klassifikation eignet. Basis der Klassifikation ist hierbei die lineare Separierung zweier Klassen im Merkmalsraum durch das Modellieren einer hierfür möglichst optimalen Hyperebene [13]. Zur besseren Vorstellung, was darunter zu verstehen ist, zeigt Abb. 5 beispielhaft eine binäre Klassifikation zweier Arten von Schwertlilien anhand der Merkmale Länge und Breite der Kronblätter. Diese spannen einen zweidimensionalen Merkmalsraum auf. Links sind suboptimale Hyperebenen eingezeichnet, die zum Teil nicht einmal zu einer Separierung der Klassen führen. Rechts ist eine Hyperebene zu sehen, wie sie eine SVM fände.

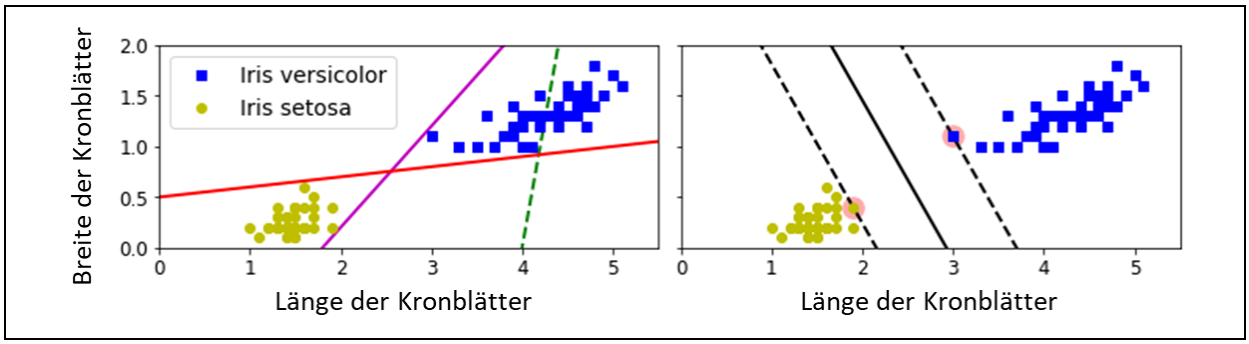

Abb. 5: Lineare Separierung durch suboptimale und optimale Hyperebene
Quelle: Entnommen aus [11, S. 156]

Zu beachten ist, dass die optimale Hyperebene die Daten nicht nur separiert, sondern auch den größtmöglichen Abstand zwischen den Trainingsdatenpunkten der einzelnen Klassen hält. Dabei stützt sich die Hyperebene vollständig auf diejenigen Datenpunkte der beiden Klassen, die dieser am nächsten sind. Diese Datenpunkte werden als *Stützvektoren* (engl. *Support Vectors*) bezeichnet. Da alle anderen Datenpunkte für die Platzierung der Hyperebene irrelevant sind, können sie nach dem Training aus dem Modell verworfen werden, wodurch SVMs sehr speichereffizient sind [13].

In den meisten Anwendungsfällen ist es nicht möglich eine Hyperebene zu finden, die zwei Klassen fehlerfrei linear separieren kann. Außerdem sind die Hyperebenen von klassischen

SVMs oftmals durch Ausreiser gestützt, welche dazu führen, dass die SVM die Daten schlecht verallgemeinert. Deswegen kommen heutzutage vor allem sogenannte *Soft-Margin-SVMs* zum Einsatz, die in Abhängigkeit von einem Hyperparameter in einem gewissen Maße Fehlklassifikationen zulassen. Dieser Hyperparameter wird als C bezeichnet. Wie in Abb. 6 zu sehen, führt ein höheres C im Allgemeinen zu weniger Fehlklassifikationen auf den Trainingsdaten. Allerdings werden die Daten dadurch auch potenziell weniger verallgemeinert, was zu einer schlechteren Qualität des Modells auf den Testdaten führen kann [11, S. 156f.]. Man spricht in solch einem Fall von einem *Overfitting* in Bezug auf die Trainingsdaten.

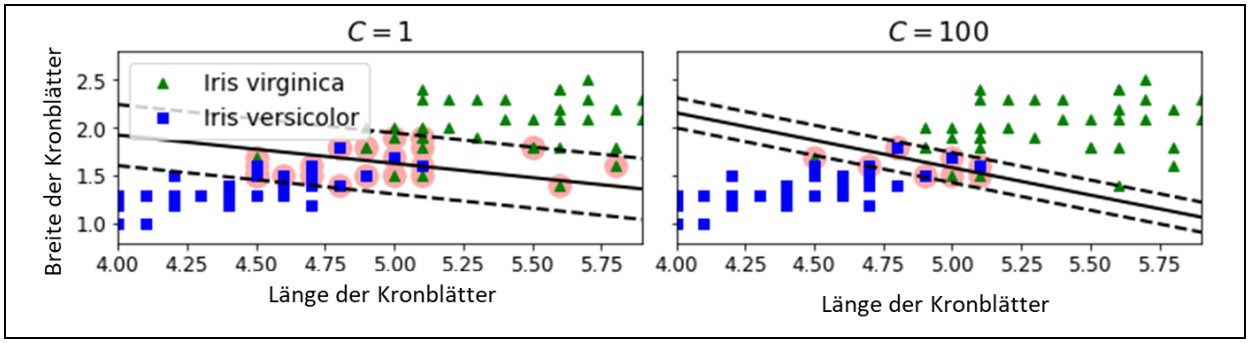

Abb. 6: Effekt des Regularisierungsparameters C bei Soft-Margin-SVMs
Quelle: Entnommen aus [11, S. 157]

Oftmals benötigen SVMs zum Finden einer linear separierenden Hyperebene einen Raum, der mehr Dimensionen als der gegebene Merkmalsraum aufweist. Um diesen zu erhalten, können die Datenpunkte durch die Anwendung einer sogenannten *Kernel-Funktion* in einen solchen höherdimensionalen Raum abgebildet werden [13]. Abb. 7 dient erneut der Veranschaulichung. Der nicht separierbare Datensatz links, der lediglich das Merkmal x_1 enthält, wird durch das Hinzufügen des Merkmals $x_2 = (x_1)^2$ linear separierbar.

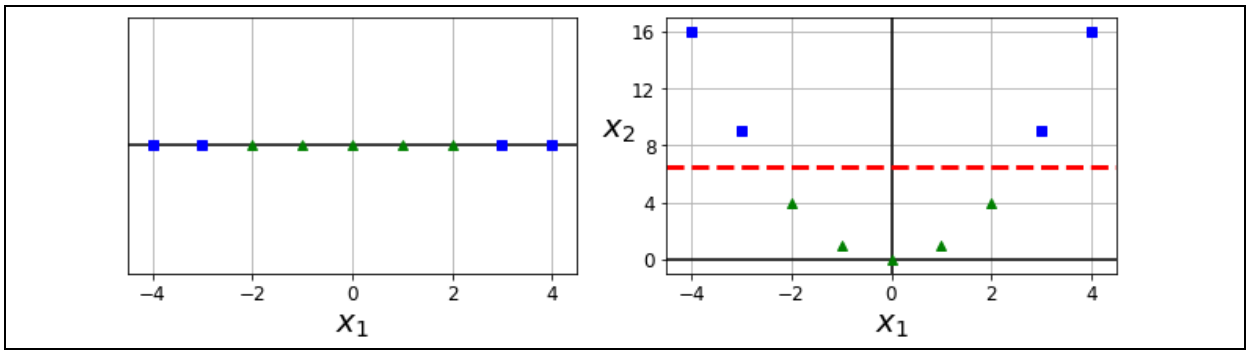

Abb. 7: Hinzufügen von Merkmalen, um Datenpunkte separierbar zu machen
Quelle: Entnommen aus [11, S. 159]

Durch die Abbildung über die Kernel-Funktion müssen die Merkmale anders als im hier gezeigten Beispiel jedoch nicht explizit zum Merkmalsraum hinzugefügt werden. Die Transformation in diesen erfolgt implizit. Die Wahl der Kernel-Funktion stellt hierbei einen weiteren Hyperparameter dar. Verbreitet sind neben dem *linearen Kernel* vor allem verschiedengradige *polynomielle* und der *Gaussche RBF-Kernel*:

$$K_\gamma(x, y) = e^{-\gamma\|x-y\|^2} \tag{2.1}$$

Dieser geht mit einem weiteren Hyperparamter γ (Gamma) einher [11, S. 160ff.]. Genau wie C ist auch γ ein Regularisierungsparameter, der die Fähigkeit zur Verallgemeinerung einer SVM maßgeblich beeinflusst. Ein hohes γ führt dazu, dass die Hyperebene unregelmäßiger und um einzelne Datenpunkte herum verläuft. Ein niedriges γ führt zu einer weicheren und somit oft besser verallgemeinernden Hyperebene [11, S. 162].

2.4.2 Decision Tree und Random Forest

Decision Trees sind mächtige und dabei dennoch gut interpretierbare Klassifikationsverfahren. Sie basieren auf einer Baumstruktur, bestehend aus der *Wurzel* sowie einer Menge an *Knoten*, *Zweigen* und *Blättern*. Der Baum bildet dabei ausgehend vom Wurzelknoten aufeinanderfolgende merkmalsbasierte Entscheidungsregeln ab, welche ggf. über Teilbäume zu den Blättern des Baumes führen. Diese repräsentieren schließlich die verschiedenen Kategorien des zugehörigen Klassifikationsproblems [14]. Abb. 8 zeigt einen Decision Tree, welcher das Beispiel der Schwertlilienklassifikation aus dem letzten Abschnitt erneut aufgreift.

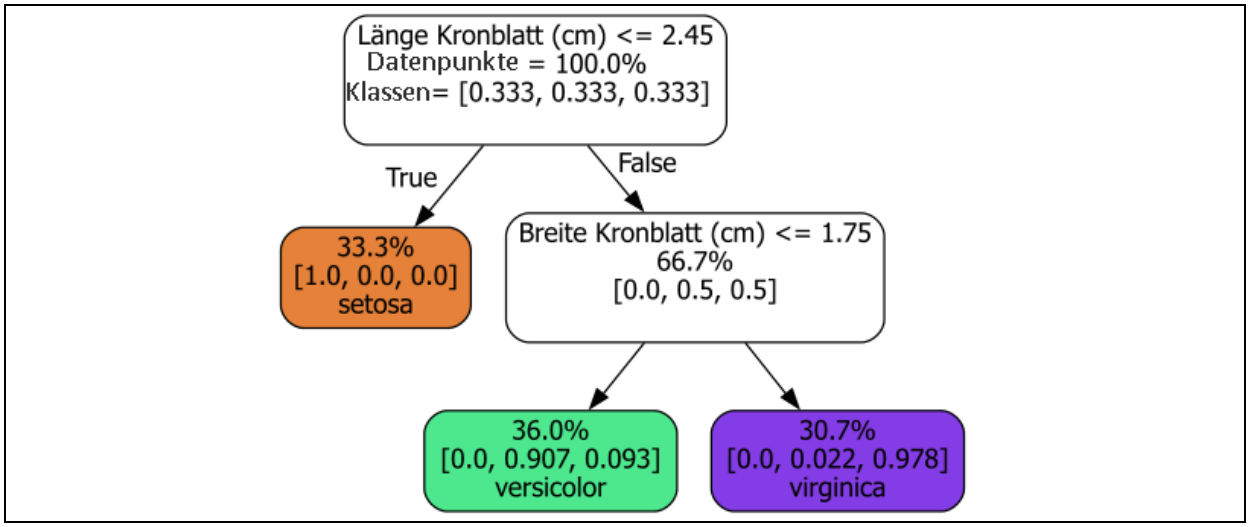

Abb. 8: Beispiel für einen Decision Tree
Quelle: Entnommen aus [11, S. 178]

Ein häufig verwendetes Verfahren zum Aufbau von Decision Trees ist der *CART* (*Classification and Regression Tree*) *Algorithmus*. Die Grundidee dieses Algorithmus ist dabei simpel: Er teilt die Menge der Trainingsdaten, basierend auf einem Merkmal k und einem Schwellenwert t_k, in zwei Untermengen auf. Dabei werden k und t_k so gewählt, dass die resultierenden Untermengen möglichst *rein* sind, also im besten Fall nur eine Klasse enthalten. Die Untermengen werden schließlich rekursiv nach demselben Prinzip aufgeteilt, bis die festgelegte maximale Tiefe (d_{max}) oder eine andere Abbruchbedingung basierend auf weiteren Hyperparametern erreicht wurde. Zu diesen gehören bspw. die minimale Anzahl an Datenpunkten, die ein Knoten enthalten muss, um weiter aufgeteilt werden zu dürfen (s_{min}^{split}), oder die minimale Menge an Datenpunkten, die ein Blatt mindestens enthalten muss (s_{min}^{leaf}). Alle drei erwähnten Hyperparameter sind Regularisierungsparameter, die dazu genutzt werden können, ein Overfitting der Daten durch den Decision Tree zu reduzieren. Auch das genutzte Reinheitsmaß ist ein Hyperparameter. Verbreitet ist die Nutzung der *Entropie* [11, S. 181ff.].

Decision Trees bilden die Basis für ein weiteres Klassifikationsverfahren: den *Random Forest*. Bei einem Random Forest handelt es sich um ein *Ensemble* an Decision Trees, welche jeweils auf Basis einer zufälligen Auswahl an Trainingsdatenpunkten erstellt werden. Außerdem werden auch die zur Aufteilung genutzten Merkmale zufällig ausgewählt. Eine Vorhersage trifft der Random Forest, indem er alle Vorhersagen des Ensembles sammelt und anschließend diejenige Klasse vorhersagt, welche im Mehrheitsentscheid siegt. Das führt im Allgemeinen zu einer deutlich besseren und allgemeineren Vorhersage im Vergleich zu einem einzelnen Decision Tree [14]. Wichtige Hyperparameter eines Random Forests sind die Anzahl der enthaltenen Bäume (n_{Trees}), die Größe des Trainingsdatensatzes für einen einzelnen Baum (s_{max}) und die Anzahl der zufällig auszuwählenden Merkmale, die für das Aufteilen einzelner Knoten in Betracht gezogen werden sollen (oft Nutzung einer Funktion $f_{max}(x)$; x entspricht der Anzahl aller Merkmale) [11, S. 199f.]. Hinzu kommen die bereits erwähnten Hyperparameter für die Decision Trees.

2.5 Sequenzielle Klassifikation mit künstlichen neuronalen Netzen

Künstliche neuronale Netze (KNNs) sind komplexe maschinelle Lernmodelle, deren Funktionsweise in den Ursprüngen durch die Vernetzung von biologischen Neuronen in menschlichen Gehirnen inspiriert ist. Sie gelten als flexibel, mächtig sowie gut skalierbar und bilden die Grundlage für das heutzutage vielseitig eingesetzte *Deep Learning* zur Bewältigung äußerst

komplexer Problemstellungen des maschinellen Lernens [11, S. 281]. Im Folgenden soll zunächst ein Überblick über den Aufbau und die prinzipielle Funktionsweise von gewöhnlichen KNNs, sogenannten *Feed-Forward-Netzen* (FFNs), gegeben werden. Darauf aufbauend wird anschließend aufgezeigt, wie diese zu sogenannten *rekurrenten neuronalen Netzen* (RNNs) erweitert werden können, um mit Hilfe von Gedächtniszellen sequenzielle Daten zu klassifizieren.

2.5.1 Aufbau eines künstlichen neuronalen Netzes

Im Allgemeinen bestehen künstliche neuronale Netze aus mehreren Verarbeitungseinheiten, welche in diesem Kontext als *Neuronen* bezeichnet werden. Diese Neuronen sind, wie in Abb. 9 dargestellt, in mehreren Schichten angeordnet, wobei die Ausgabe eines jeden Neurons einer Schicht an die Eingabe der Neuronen in der nächsten Schicht weitergereicht wird [15].

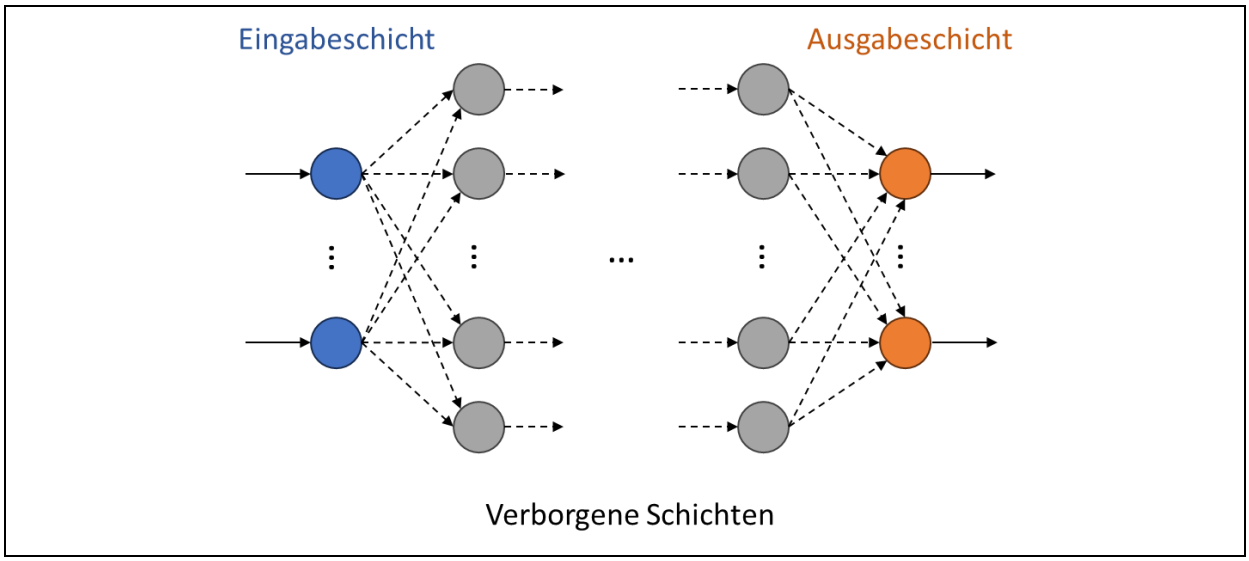

Abb. 9: Aufbau eines künstlichen neuronalen Netzes

Die erste Schicht eines KNNs bezeichnet man als *Eingabeschicht* (engl. *Input Layer*) und die zugehörigen Neuronen als *Eingabeneuronen* (engl. *Input Neurons*). Die Anzahl der Eingabeneuronen eines Netzes wird durch die Anzahl an gegebenen Merkmalen in den Trainingsdatenpunkten festgelegt. Analog zur Eingabeschicht bezeichnet man die letzte Schicht eines KNNs als *Ausgabeschicht* (engl. *Output Layer*). Sie besteht aus den *Ausgabeneuronen* (engl. *Output Neurons*) [16, S. 11]. Die Anzahl der Ausgabeneuronen ist abhängig von der Problemstellung. Bei einem mehrklassigen Klassifikationsproblem verwendet man jedoch üblicherweise ein Neuron pro vorherzusagender Kategorie [11, S. 295f.].

Etwas komplizierter ist der Aufbau des Netzes zwischen der Ein- und Ausgabeschicht. Die dort befindlichen Schichten bezeichnet man als *verborgene Schichten* (engl. *Hidden Layers*). Sowohl die Anzahl der verborgenen Schichten als auch die Anzahl der Neuronen in den einzelnen Schichten folgen keiner Faustregel [16, S. 12]. Sie sind Hyperparameter, welche die Fähigkeit des Netzes beeinflussen, komplexe Muster und Konzepte in den Daten zu erlernen und abzubilden. Ein KNN mit mehr als zwei verborgenen Schichten gilt als tiefes neuronales Netz (engl. *Deep Neural Network*) [16, S. 37]. Komplexe Funktionen können sowohl durch tiefe als auch durch weite (viele Neuronen pro Schicht) KNNs modelliert werden. Im Allgemeinen besitzt das Erhöhen der Anzahl an Schichten jedoch eine deutlich höhere Parametereffizienz, wodurch eine vergleichbare Leistung durch deutlich weniger Neuronen erreicht werden kann [11, S. 326f.].

2.5.2 Künstliche Neuronen und Aktivierungsfunktionen

Mathematisch betrachtet implementiert ein einzelnes künstliches Neuron eine Funktion, welche einen Eingabevektor $\boldsymbol{x} = [x_1, \dots, x_m]$ auf eine einzelne numerische Ausgabe abbildet. Diese Funktion ist wie folgt definiert:

$$h_{\boldsymbol{w},b}(\boldsymbol{x}) = \varphi(b + \sum_{i=1}^{m} w_i \times x_i) \tag{2.2}$$

Hierbei stellt $\boldsymbol{w} = [w_1, \dots, w_m]$ einen Vektor dar, der jeder Eingabe des Neurons ein Gewicht zuordnet. b ist der sogenannte *Bias-Term* und φ (Phi) eine nicht lineare *Aktivierungsfunktion*. Die Ausgabe eines Neurons entspricht also der Ausgabe einer Aktivierungsfunktion in Abhängigkeit von der gewichteten Summe der Eingaben und dem Bias-Term. Dabei ist die Eingabe, wie in Abb. 10 veranschaulicht, in der Regel durch die Ausgaben der Neuronen der vorherigen Schicht bestimmt [15].

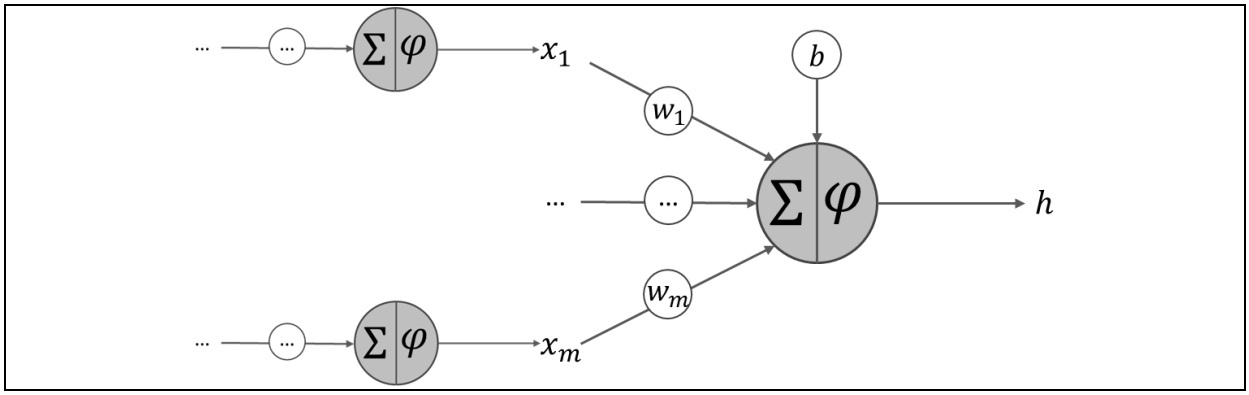

Abb. 10: Veranschaulichung eines künstlichen Neurons

Würde die Aktivierungsfunktion φ aus der obigen Funktion (2.2) entfallen, dann könnte ein künstliches Neuron lediglich lineare Transformationen realisieren. Da eine Verkettung von linearen Transformationen ebenfalls linear ist, könnten KNNs somit lediglich lineare Zusammenhänge abbilden und nicht zur Lösung komplexer Probleme eingesetzt werden. Deshalb werden bestimmte Aktivierungsfunktionen verwendet, um die Nichtlinearität der Neuronen zu garantieren. Dies ermöglicht KNNs theoretisch, beliebig komplexe (stetige) Funktionen zu approximieren [11, S. 293f.].

Die beliebtesten Aktivierungsfunktionen sind die *Sigmoid-Funktion* $\sigma(z)$, der *Tangens-Hyperbolicus* $tanh(z)$ und die *Rectified-Linear-Unit-Funktion* $ReLU(z)$. Die Funktionen sind wie folgt definiert:

$$\sigma(z) = \frac{1}{1+e^{z}} \tag{2.3}$$

$$tanh(z) = 2\sigma(2z) - 1 \tag{2.4}$$

$$ReLU(z) = \max(0, z) \tag{2.5}$$

Wie auch in Abb. 11 zu erkennen ist, verlaufen sowohl die Sigmoid-Funktion als auch der Tangens-Hyperbolicus s-förmig innerhalb eines nach unten und oben begrenzten Wertebereichs von (0, 1) bzw. (-1, 1). Der Wertebereich der ReLU-Funktion lautet $[0, \infty)$. Sie ist somit lediglich nach unten begrenzt. In der Praxis hat sich die ReLU-Funktion u. a. auf Grund ihrer schnellen Berechenbarkeit inzwischen als Standard durchsetzen können.

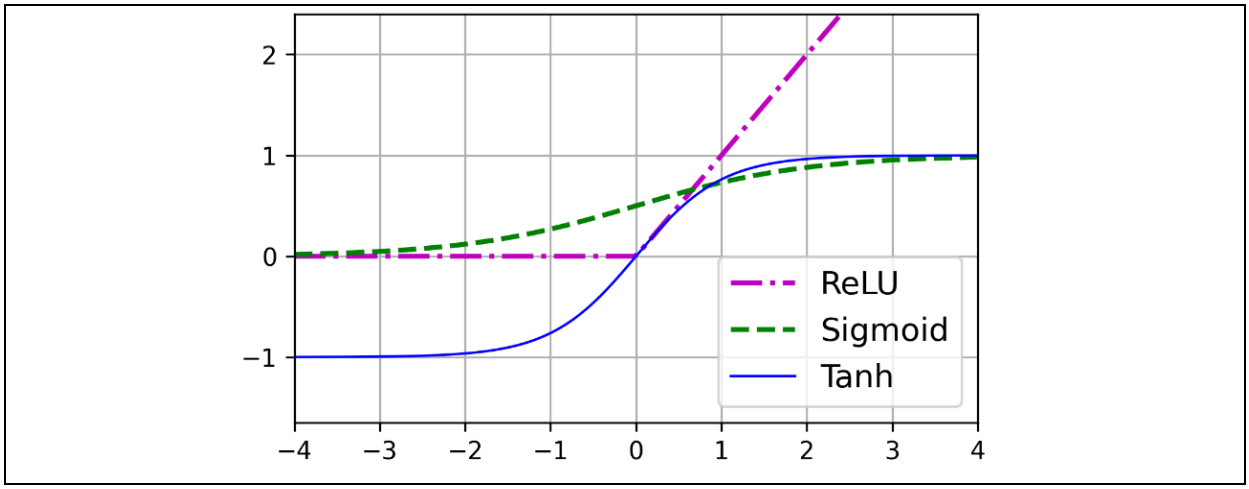

Abb. 11: Aktivierungsfunktionen: tanh(z), $\sigma(z)$ und $ReLU(z)$

Für die Ausgabeschicht wird oftmals eine andere Aktivierungsfunktion als in den anderen Schichten des KNNs verwendet. Diese ist abhängig von der Aufgabenstellung und der

angestrebten Ausgabe des Netzes. Bei mehrklassigen Klassifikationsproblemen nutzt man üblicherweise die *Softmax-Funktion* [15]. Diese ist für das Ausgabeneuron j $(1 \leq j \leq K)$ wie folgt definiert:

$$softmax(\boldsymbol{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{2.6}$$

Hierbei entspricht K der Anzahl der Ausgabeneuronen bzw. Klassen und $\boldsymbol{z} = [z_1, \dots, z_K]$ einem Vektor, der die gewichteten Summen aller Ausgabeneuronen enthält. Durch Anwendung der Softmax-Funktion summieren sich alle Ausgaben des KNNs zu eins auf, wodurch man die Ausgabe des Neurons j als Wahrscheinlichkeit interpretieren kann, dass der eingegebene Datenpunkt der zu j zugeordneten Klasse angehört [11, S. 149f.].

2.5.3 Training neuronaler Netze: Backpropagation und Early-Stopping

Beim Training der künstlichen Neuronen kommt es darauf an, die Gewichte und den Bias-Term, welche in der Regel zufällig initialisiert werden, so anzupassen, dass sie in Kombination mit der gewählten Aktivierungsfunktion zu der gewünschten Ausgabe führen. Hierbei kommt ein Verfahren zum Einsatz, welches sich *Backpropagation*, nennt.

Der Backpropagation-Algorithmus basiert auf dem sogenannten *Gradientenabstieg*, ein Verfahren, welches dazu genutzt wird, die Eingabeparameter einer *Kostenfunktion* (*Loss-Function*) iterativ so anzupassen, dass ihre Ausgabe minimiert wird. Ohne auf die konkrete Mathematik einzugehen, berechnet der *Gradientenabstieg* hierbei in jedem Zeitschritt ausgehend von der aktuellen Position im Parameterraum die Richtung des steilsten Abstiegs, welche durch den sogenannten *lokalen Gradienten* definiert wird. Darauf basierend werden die Parameter so angepasst, dass die Position in Richtung dieses Abstiegs schrittweise verschoben wird, bis der lokale Gradient null wird, also ein Minimum gefunden wurde. Ein wichtiger Hyperparameter beim Gradientenabstieg ist die Größe der Schritte, welche durch die sogenannte *Lernrate* festgelegt wird. Ist diese zu klein, kann es sehr viele Iterationen in Anspruch nehmen, bis der Algorithmus konvergiert. Eine größere Lernrate verschnellert in der Regel das Training, kann aber auch dazu führen, dass Minima übersprungen werden und der Algorithmus im schlimmsten Fall divergiert [11, S. 121]. Abb. 12 veranschaulicht das Gradientenabstiegsverfahren zur Minimierung einer Verlustfunktion auf Basis zweier Parameter.

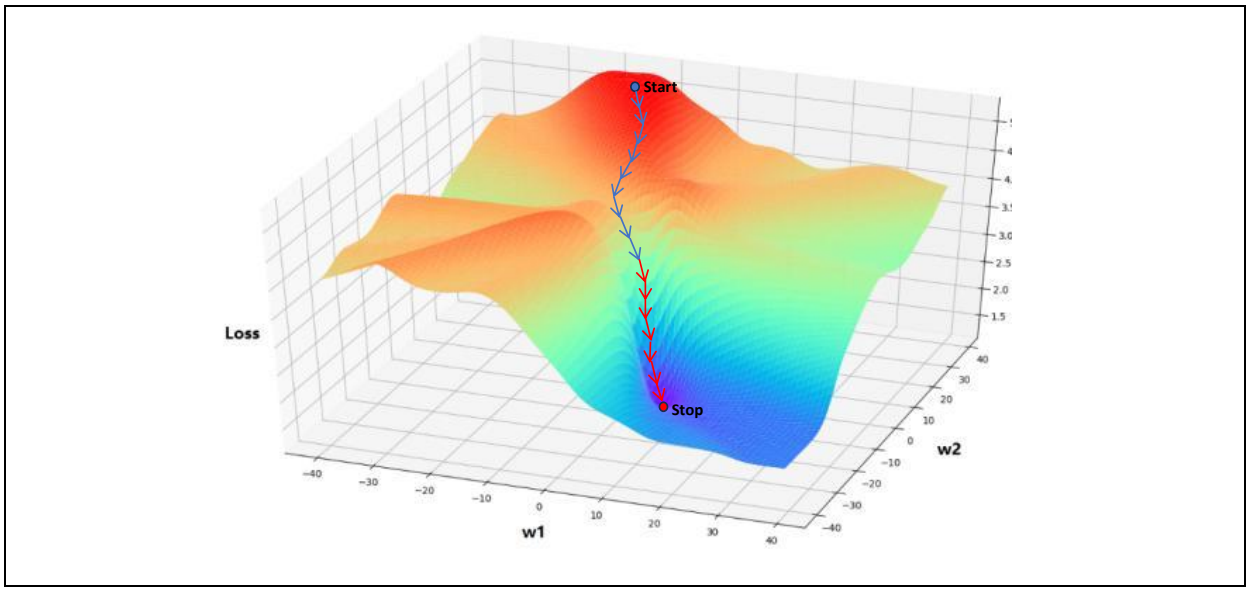

Abb. 12: Darstellung des Gradientenabstiegs
Quelle: Entnommen aus [17]

Beim Backpropagation-Algorithmus für das Training neuronaler Netze können hierbei die Gradienten zur Anpassung aller Modellparameter (Gewichte und Bias-Terme der Neuronen) für jede Iteration in nur zwei Durchgängen durch das Netz berechnet werden. Hierzu werden in der Regel sogenannte *Mini-Batches* (z. B. eine Teilmenge von 32 Datenpunkten) aus den Trainingsdatensatz genutzt, um die zugehörige Ausgabe der einzelnen Neuronen und des gesamten Netzes zu berechnen. Dies bezeichnet man als *Vorwärtsdurchlauf* oder *Forward Propagation*. Als nächstes wird der Ausgabefehler des Netzes auf Basis einer Verlustfunktion bestimmt, welche die Labels mit der berechneten Ausgabe vergleicht und das Ergebnis dieses Vergleichs in Form eines Messwertes zurückliefert. Nun wird berechnet, wie viel jedes Gewicht der Ausgabeschicht zum Fehler beigetragen hat. Anschließend wird bestimmt, wie hoch der Beitrag zum Fehler der Gewichte der darunterliegenden Schicht war. Der Algorithmus arbeitet sich somit rückwärts durch die Schichten des Netzes, was als *Rückwärtsdurchlauf* oder *Backpropagation* bezeichnet wird. Hierdurch werden die Fehlergradienten der einzelnen Gewichte und Bias-Terme effizient über das gesamte Netz berechnet. Schließlich müssen diese dann nur noch entsprechend angepasst werden, wodurch die Verlustfunktion minimiert bzw. die Ausgabe des Netzes korrigiert wird. Während des Trainingsprozesses wird der gesamte Trainingsdatensatz meist mehrfach durchlaufen. Einen einzelnen Durchlauf bezeichnet man hierbei als *Epoche* [11, S. 291f.].

Unter der Voraussetzung, dass das Netz mächtig genug ist (in Bezug auf Tiefe und Breite), führt ein ausreichend langes Training in jedem Fall zu einer sehr starken Anpassung des Netzes

an die Trainingsdaten, welches häufig ein Overfitting, also eine schlechte Verallgemeinerung auf unbekannte Daten wie den Testdaten, zur Folge hat. Um dies zu vermeiden, wird oftmals das Regularisierungsverfahren *Early-Stopping* eingesetzt. Hierbei wird das Training nicht in Abhängigkeit des Vorhersagefehlers auf den Trainingsdaten gestoppt, sondern dann, wenn der sogenannte *Validierungsfehler* aufhört zu sinken. Dieser wird am Ende einer jeden Epoche auf Basis eines dritten Datensatzes, dem *Validierungsdatensatz*, bestimmt. Wichtig ist, dass der Validierungsdatensatz nicht am Training und somit an der Anpassung des Netzes beteiligt ist [11, S. 143].

2.5.4 Rekurrente neuronale Netze

Die bisher beschriebenen Feed-Forward-Netze bilden einen gerichteten azyklischen Graphen, in welchem die Aktivierung der einzelnen Neuronen(-schichten) in nur eine Richtung, ausgehend von der Eingabe- bis hin zur Ausgabeschicht, erfolgt. Bei *rekurrenten neuronalen Netzen* ist dies anders. Sie enthalten neben den üblichen Neuronen und vorwärtsgerichteten Verbindungen sogenannte *rekurrente Elemente*, bspw. in Form von Verbindungen, welche die Neuronen einer Schicht mit sich selbst oder mit vorangegangenen Schichten verbinden [15]. RNNs enthalten somit Rückkopplungen, die dazu führen, dass die Aktivierung der rekurrenten Elemente und damit letztlich auch die Ausgabe des gesamten Netzes nicht nur von der aktuellen Eingabe, sondern auch von allen vorangegangenen Eingaben abhängig ist. Diese Eigenschaft erlaubt es RNNs, beliebig lange Eingabesequenzen bestehend aus mehreren Zeitschritten zu verarbeiten und hierbei auf zeitlich codierte Informationen zurückzugreifen [16, S. 202f.].

Es gibt eine Vielzahl an Möglichkeiten, ein Feed-Forward-Netz zu einem rekurrenten neuronalen Netz zu erweitern (also rekurrente Elemente hinzuzufügen). In dieser Arbeit werden hierfür Gedächtniszellen eingesetzt. Im Allgemeinen versteht man darunter Teile eines neuronalen Netzes, welche einen internen Zustand im nächsten Zeitschritt an sich selbst weitergeben. Bei einfachen Gedächtniszellen ist dieser Zustand mit der Ausgabe identisch. Abgesehen von der Rückkopplung unterscheiden sie sich nicht von herkömmlichen Neuronen, weshalb man sie auch als *rekurrente Neuronen* bezeichnet. Da einfache rekurrente Neuronen jedoch nur dazu in der Lage sind, kurze Muster über etwa zehn Zeitschritte zu erlernen, wurden auch komplexere Gedächtniszellen wie die *Long-Short-Term-Memory-Zellen (LSTM-Zellen)* entwickelt, welche dazu in der Lage sind, etwa zehnfach längere Muster abzubilden [11, S. 504f.]. LSTM-Zellen nutzen hierfür zwei interne Zustände, welche als Langzeit- und Kurzzeitgedächtnis aufgefasst werden können. Die Steuerung dieser Zustände erfolgt über komplexe Mechanismen, an

welchen verschiedene interne Neuronenschichten beteiligt sind. Diese erlauben es LSTM-Zellen zu erlernen, wichtige Eingaben zu erkennen, diese – solange wie benötigt – im Langzeitzustand abzuspeichern und wichtige Informationen daraus zu entnehmen [11, S. 520f.]. Für einen genaueren Einblick in die Funktionsweise von LSTM-Zellen sei an dieser Stelle auf die zugehörige Literatur, wie dem Paper von Hochreiter und Schmidhuber [18], in welchem LSTMs erstmalig vorgestellt wurden, verwiesen.

In Abb. 13 wird eine einzelne Gedächtniszelle veranschaulicht. Links ist hierbei die Zelle statisch und rechts über die Zeit aufgerollt dargestellt. Im Aufrollen über die Zeit liegt auch der Schlüssel zum Training von RNNs mittels *Backpropagation Through Time*, dies soll hier jedoch nicht weiter ausgeführt werden [11, S. 504ff.].

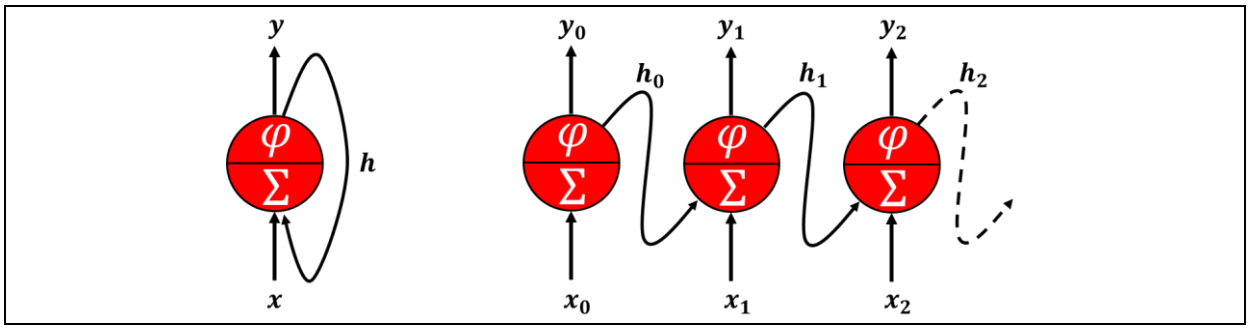

Abb. 13: Gedächtniszelle: statisch und über die Zeit aufgerollt

Grundsätzlich kann man RNNs entsprechend ihrer Ausgabe in zwei Kategorien einteilen. RNNs, welche basierend auf der Eingabesequenz selbst eine Sequenz ausgeben, werden *Sequence to Sequence* (STS) RNNs genannt. Sie eignen sich bspw. dann, wenn sich eine vorherzusagende Klasse über den Zeitverlauf der Sequenz verändern kann. Ist im Gegensatz dazu nur die letzte Ausgabe eines RNNs von Relevanz, kommen in der Regel *Sequence to Vector* (STV) RNNs zum Einsatz, welche alle Ausgaben des Netzes – mit Ausnahme der letzten – ignorieren. Eine Darstellung dieser beiden Netzarchitekturen zeigt Abb. 14 [11, S. 505].

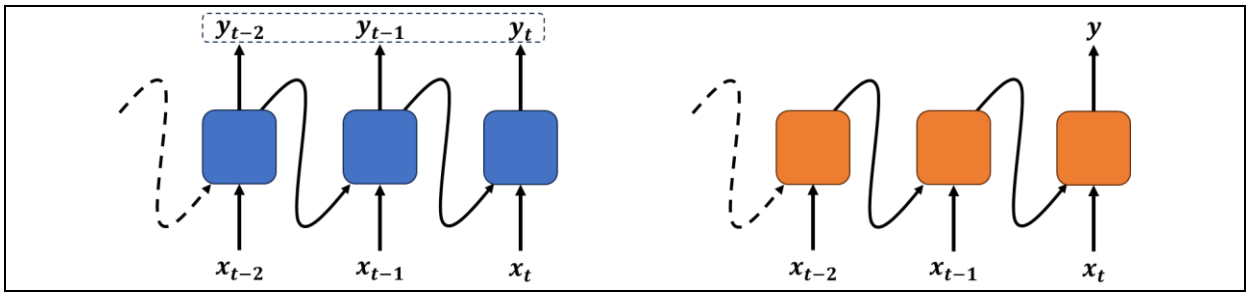

Abb. 14: Sequence to Sequence und Sequence to Vector RNN

2.6 Bewertungsmaße für Klassifikatoren

Um die Qualität eines Klassifikationsverfahrens messbar zu machen und durch Training und Parameteroptimierung zu steigern, ist es wichtig, Bewertungsmaße zu definieren, die die Vorhersageleistung eines Klassifikators widerspiegeln. In diesem Abschnitt sollen solche Maße definiert und erläutert werden, wobei zum besseren Verständnis in allen Fällen schrittweise von einer binären Klassifikation auf ein mehrklassiges Problem verallgemeinert wird.

2.6.1 Konfusionsmatrizen

Ein verbreitetes Werkzeug, um die Vorhersageleistung eines Klassifikators zu visualisieren, ist die *Konfusionsmatrix*. Dabei handelt es sich um eine Tabelle, bei welcher für eine gegebene Menge an Datenpunkten die Schnittmengen aller vorhergesagten und tatsächlichen Klassen ausgezählt werden. Die Zeilen der Matrix stehen hierbei typischerweise für die tatsächlichen Klassen, wohingegen die Spalten die Vorhersagen repräsentieren [11, S. 94]. Abb. 15 zeigt ein Beispiel für eine Konfusionsmatrix zur Auswertung eines binären Klassifikators auf Basis von 65 Datenpunkten.

		65	**Vorhergesagte Klasse**	
			Positiv	**Negativ**
Tatsächliche Klasse	**Positiv**		**Richtig positiv (RP)** *12*	**Falsch negativ (FN)** *10*
	Negativ		**Falsch positiv (FP)** *3*	**Richtig negativ (RN)** *40*

Abb. 15: Beispiel für eine zweiklassige Konfusionsmatrix

Konfusionsmatrizen können eine beliebige Anzahl von Klassen umfassen. Wichtig ist lediglich, dass deren Anordnung in den Zeilen und Spalten identisch ist, damit alle korrekt klassifizierten Datenpunkte auf der Hauptdiagonalen der Matrix widergespiegelt werden. Ein guter Klassifikator enthält entsprechend außerhalb dieser Hauptdiagonalen eine möglichst geringe Anzahl an Datenpunkten [12].

2.6.2 Genauigkeit

Die *Genauigkeit* (engl. *Accuracy*) ist eine häufig genutzte Metrik bei Klassifikationsproblemen. Sie misst den Anteil der korrekt klassifizierten Datenpunkte an der Gesamtzahl aller betrachteten Datenpunkte. Für eine binäre Klassifikation ergibt sich somit die folgende Formel:

$$Accuracy_2 = \frac{RP+RN}{RP+FP+FN+RN} \tag{2.7}$$

Bei K-klassigen Klassifikationsproblemen kann man die Genauigkeit berechnen, indem man zunächst die Genauigkeitswerte für alle Klassen im Einzelnen und anschließend das arithmetische Mittel dieser Werte berechnet. Hiermit ergibt sich die erweiterte Formel:

$$Accuracy_K = \frac{1}{K}\sum_{k=1}^{K}\frac{RP_k+RN_k}{RP_k+FP_k+FN_k+RN_k} \tag{2.8}$$

Die Genauigkeit ist ein sehr intuitives Maß. Unabhängig von der Anzahl der Klassen liegt sie – wie auch alle nachfolgenden Maße – immer im Intervall [0, 1], wobei ein Klassifikator umso besser ist, desto näher der Wert an eins liegt. Allerdings sollte man vorsichtig sein, die Genauigkeit einzusetzen, falls der zugrundeliegende Datensatz unausgeglichen ist, da sie dazu tendiert, große Klassifikationsfehler in unterrepräsentierten Klassen zu verbergen [12]. Für die Matrix in Abb. 15 berechnet sich eine Genauigkeit von 0,8.

2.6.3 Relevanz

Die *Relevanz* (engl. *Precision*) misst die Genauigkeit der positiven Vorhersagen. Für eine binäre Klassifikation berechnet sie sich aus der folgenden Formel [11, S. 95]:

$$Precision_2 = \frac{RP}{RP+FP} \tag{2.9}$$

Für K-klassige Klassifikationsprobleme haben sich verschiedene Herangehensweisen herausgebildet, die Relevanz zu berechnen. Die in dieser Arbeit verwendete ist die sogenannte *Macro-Precision*, welche sich analog zur mehrklassigen Genauigkeit berechnet [12]:

$$Precision_K = \frac{1}{K}\sum_{k=1}^{K}\frac{RP_k}{RP_k+FP_k} \tag{2.10}$$

Für die Matrix in Abb. 15 beträgt die Relevanz ebenso wie die Genauigkeit 0,8. Da die Relevanz durch die ausschließliche Beachtung der positiven Vorhersagen für sich allein stehend sehr einseitig ist, geht sie in der Regel mit der Sensitivität einher.

2.6.4 Sensitivität

Die *Sensitivität* (engl. *Recall*), auch als Trefferquote bezeichnet, ist der Anteil der tatsächlich positiven Datenpunkte, die vom Klassifikator als solche vorhergesagt wurden. Sie berechnet sich aus der folgenden Formel [11, S. 95]:

$$Recall_2 = \frac{RP}{RP+FN} \tag{2.11}$$

Auch für die Sensitivität haben sich im Hinblick auf K-klassige Klassifikationsprobleme verschiedene Herangehensweisen gebildet. In dieser Arbeit wird der *Macro-Recall* verwendet, der sich analog zur mehrklassigen Genauigkeit und Relevanz berechnet [12]:

$$Precision_K = \frac{1}{K}\sum_{k=1}^{K}\frac{RP_k}{RP_k+FN_k} \tag{2.12}$$

Auch die Sensitivität ist für sich allein genommen sehr einseitig. Deshalb wird sie in der Regel gemeinsam mit der Relevanz ausgewertet. Für die Matrix in Abb. 15 beträgt die Sensitivität lediglich ca. 0,55.

2.6.5 F1-Score

Da Relevanz und Sensitivität oft gemeinsam ausgewertet werden, kann es bequem sein diese zu einer einzigen Metrik zusammenzufassen: dem *F1-Score*. Dieser berechnet sich aus dem harmonischen Mittelwert von Relevanz und Sensitivität bzw. Macro-Precision und -Recall bei K-klassigen Klassifikationsproblemen:

$$F1Score = \frac{2}{\frac{1}{Precision}+\frac{1}{Recall}} \tag{2.13}$$

Im Gegensatz zur Genauigkeit eignet sich der F1-Score auch zur Beurteilung von unausgeglichenen Datensätzen. Bei der Interpretation des Ergebnisses sollte lediglich beachtet werden, dass das harmonische Mittel niedrigeren Werten eine höhere Gewichtung gibt [11, S. 96]. Für die Matrix in Abb. 15 beträgt der F1-Score ca. 0,65.

2.7 Eingesetzte Technologien und Programmbibliotheken

Für die im Rahmen dieser Bachelorarbeit realisierten Versuche und praktischen Umsetzungen, kamen zahlreiche externe Technologien und Programmbibliotheken zum Einsatz. In den nachfolgenden Unterabschnitten werden die Wichtigsten aufgeführt und deren Zweck kurz beschrieben.

2.7.1 Valhalla Map-Matching-Service

Zur praktischen Umsetzung des Map-Matchings wird im Zuge dieser Arbeit die http-basierte Map-Matching-API der Open-Source-Routing-Engine *Valhalla* in der Version 3.3.0 eingesetzt. Diese Arbeit folgt damit der Empfehlung von Hagen und Saki, die in ihrem Paper [8] die Nutzung dieses Services als performante und gut skalierbare Alternative zu kommerziellen oder in der Nutzung limitierten Map-Matching-Services, wie bspw. dem von *Google Maps*, nahelegen.

Die API von Valhalla umfasst hierbei zwei verschiedene Map-Matching-Endpunkte, die ausgehend von der gleichen Eingabesequenz an GNSS-Punkten verschiedene Operationen ausführen. Der `trace_route` Endpunkt liefert dabei im Wesentlichen die ans Straßennetz angepasste Sequenz mit einigen wenigen Zusatzinformationen wie der Matching-Distanz zurück, wohingegen der `trace_attributes` Endpunkt dazu genutzt werden kann, um detaillierte Informationen zu den zugeordneten Straßensegmenten, wie bspw. gültige Geschwindigkeitsbegrenzungen oder die Art der Straße, zu erhalten. Für beide Endpunkte muss dabei explizit der *Costing-* bzw. *Matching-Modus* gesetzt werden. Die möglichen Modi umfassen u.a. ein Map-Matching für Fußgänger (*pedestrian*), Fahrradfahrer (*bicycle*) und motorisierte Straßenfahrzeuge (*auto*). Darüber hinaus können zahlreiche zusätzliche Parameter für das Map-Matching gesetzt werden. Weitere Informationen hierzu sind der Dokumentation von Valhalla unter [19] zu entnehmen.

2.7.2 Python

Der gesamte Code, der im Rahmen dieser Arbeit verfasst wurde, ist in der Programmiersprache *Python* in der Version 3.10 geschrieben. Bei Python handelt es sich um eine universell einsetzbare, üblicherweise interpretierte, höhere Programmiersprache, die verschiedene Programmierparadigmen, wie die objektorientierte und funktionale Programmierung, unterstützt. Dabei ist Python dynamisch typisiert, wodurch sie sich auch als *Skriptsprache* eignet [20]. Python besitzt eine ausführliche Dokumentation [21], welcher weitere Informationen zur Programmiersprache entnommen werden können.

Zu den Gründen, warum Python für die zu dieser Arbeit zugehörigen Implementierungen gewählt wurde, zählen neben der einfachen Syntax und damit oft auch guten Lesbarkeit des resultierenden Codes insbesondere die zahlreichen nützlichen Standard- und Drittanbieter-Bibliotheken, die für diese Sprache existieren. Für das Verständnis des umgesetzten Codes werden dabei insbesondere grundlegende Kenntnisse zu den folgenden zwei Bibliotheken

vorausgesetzt: *pandas* und *NumPy*. Diese können bei Bedarf durch ein Studium der referenzierten Dokumentationen erlangt werden.

Bei pandas handelt es sich um eine Bibliothek zur Datenanalyse und -verarbeitung. Die Bibliothek erweitert Python um zusätzliche Datenstrukturen, wie die sogenannten *Dataframes*, um große Datensätze effizient zu manipulieren [22]. Verwendete Funktionen und Objekte von pandas werden im Code über die *Namespace-Referenz* `pd` gekennzeichnet.

NumPy ist eine Bibliothek für effiziente numerische Berechnungen. NumPy enthält neben zahlreichen Funktionen auch eigene Datenstrukturen, wie *Arrays*, die für performante Operationen auf höherdimensionalen Vektoren und Matrizen eingesetzt werden können. Diese Datenstrukturen bilden die Grundlage für andere Bibliotheken, wie bspw. pandas [23]. Im Code werden NumPy Funktionen und Objekte über die Namespace-Referenz `np` gekennzeichnet.

2.7.3 Scikit-learn

Scikit-learn gilt als umfangreichste Open-Source-Bibliothek für das maschinelle Lernen in Python. Die bereitgestellten Funktionen und Klassen umfassen dabei insbesondere die Bereiche Datentransformation und -vorverarbeitung, überwachte Lernverfahren, unüberwachte Lernverfahren sowie Modell-Evaluierung und -Auswahl. Dabei sind alle Implementierungen stark auf ihre Berechnungseffizienz optimiert. [24]. Besonders hervorzuheben ist außerdem das gute Schnittstellendesign, welches über all diese Bereiche hinweg einheitliche Konventionen festlegt. Dabei werden alle implementierten Klassen auf drei grundlegende Oberklassen generalisiert: *Estimatoren*, *Transformer* und *Prädiktoren*. Zu ersteren zählen alle Objekte, die interne Parameter auf Basis eines Datensatzes erlernen oder abschätzen können. Dies wird über die Methode `fit` angestoßen, welche den Datensatz entgegennimmt. Alle Parameter können anschließend über öffentliche Attribute des Objekts abgerufen werden. Transformer sind Estimatoren, welche einen Datensatz zusätzlich transformieren können. Die Transformation wird mit der Methode `transform` ausgeführt, die einen Datensatz entgegennimmt und den transformierten Datensatz zurückliefert. Die Transformation beruht dabei auf den gelernten Parameter der `fit`-Methode. Alternativ kann auch die Methode `fit_transform` genutzt werden, die dem aufeinanderfolgenden Aufruf der beiden Methoden entspricht. Zuletzt sind Prädiktoren Estimatoren, die in der Lage sind, auf Basis gegebener Datenpunkte Vorhersagen zu treffen. Alle Prädiktoren besitzen hierfür die Methode `predict`, welche einen Satz an Datenpunkten entgegennimmt und einen Satz entsprechender Vorhersagen zurückliefert. Außerdem besitzen

alle Prädiktoren die Methode `score`, über welche die Vorhersagequalität des Prädiktors über einen Testdatensatz und unter der Nutzung verschiedener Metriken ermittelt werden kann. Nötige Hyperparamter werden bei allen Klassen über den Konstruktor gesetzt. [11, S. 66].

Eine Abfolge von Transformern und ein abschließender beliebiger Estimator können außerdem als wiederverwendbare *Pipeline* definiert und abgespeichert werden, wobei sich diese anschließend wie ein einziger Estimator verhält. Damit können bspw. alle benötigten Vorverarbeitungsschritte und eine anschließende Klassifikation zusammengefasst werden. Bei Bedarf können weitere Informationen zu Funktionen und Klassen von scikit-learn über die Dokumentation unter [25] bezogen werden.

2.7.4 Keras und TensorFlow

Keras ist eine Open-Source-Deep-Learning-Bibliothek für Python, die es erlaubt, alle möglichen Arten von neuronalen Netzen zu entwerfen, zu trainieren, auszuwerten und auszuführen. Seit Veröffentlichung der Referenzimplementierung im Jahr 2015 zählt Keras aufgrund ihrer Benutzerfreundlichkeit und Flexibilität zu den verbreitetsten Bibliotheken für das Deep Learning. Einsteigern kommt hierbei zugute, dass die API von Keras an vielen Stellen durch die scikit-learn-API beeinflusst ist. Für die aufwendigen Berechnungen, welche insbesondere mit dem Training tiefer neuronaler Netze einhergehen, ist die Referenzimplementierung von Keras auf ein zusätzliches Rechen-Backend angewiesen. Ein solches wird beispielsweise durch *TensorFlow* bereitgestellt [11, S. 297f.].

Bei *TensorFlow* handelt es sich um eine Bibliothek für numerische Berechnungen und datenstromgetriebene Programmierung, die sich durch ihre hohe Effizienz und Anpassbarkeit auszeichnet. TensorFlow wurde durch das Google-Brain-Team entwickelt und ist inzwischen Basis für eine Vielzahl an Google-Services mit umfangreichen Rechenanforderungen [11, S. 379]. Seit Version 2 bringt TensorFlow seine eigene Keras-Implementierung mit sich, welche die Referenzimplementierung um einige nützliche Zusatzfeatures, unter anderem zur Datenvorverarbeitung, erweitert. Außerdem bietet TensorFlow unter bestimmten Voraussetzungen auch die Möglichkeit, Berechnungen auf einer oder mehreren GPUs durchzuführen, wodurch insbesondere das Training von neuronalen Netzen beschleunigt werden kann [11, S. 277f.].

Für einen tieferen Überblick über TensorFlow sei an dieser Stelle auf die offizielle Webseite unter [26] verwiesen. In Bezug auf Keras können weitere Informationen zu den verwendeten Funktionen und Klassen bei Bedarf über die API-Dokumentation unter [27] bezogen werden.

3 Verwandte Arbeiten

3.1 Vorangegangene Abschlussarbeiten

Dieser Arbeit gehen drei Abschlussarbeiten voraus, welche sich ebenfalls mit der Klassifikation von Verkehrsteilnehmern auf Basis von Positionsdaten im Rahmen der erweiterten Umfeldwahrnehmung autonomer Fahrzeuge befasst haben. Nachfolgend werden die wichtigsten Erkenntnisse, aber auch die Grenzen dieser Arbeiten aufgezeigt.

Die Bachelorarbeit von Recep Furgan Torlak [28] aus dem Jahr 2022 bildet im Wesentlichen den Grundstein für die späteren Arbeiten. Die Arbeit setzt sich mit verschiedenen Möglichkeiten der Datengewinnung auseinander, wobei der Fokus auf der Abfrage von Positionsdaten aus der Verkehrssimulation *CARLA* liegt, für die im Zuge der Arbeit auch eine Python-API umgesetzt wurde. Auch Möglichkeiten reale Daten zu gewinnen und simulierte Daten realistischer zu machen, werden in der Arbeit aufgeführt. Außerdem beschäftigte sich Torlak mit der Berechnung von Bewegungsinformationen aus den ermittelten Positionsdaten, welche später zur Klassifikation genutzt werden sollen. Die Klassifikation selbst ist jedoch kein Teil der Arbeit.

Maximilian Sohl knüpft in seiner Bachelorarbeit [29], ebenfalls aus dem Jahr 2022, direkt an die Arbeit von Torlak an. Er nutzt die aus der Verkehrssimulation CARLA gewonnenen Daten, um die Leistung von drei verschiedene Klassifikationsverfahren zu evaluieren. Bei diesen handelt es sich um die Verfahren *Support Vector Machine, Decision Tree* und *K Nearest Neighbours.* Hierbei sollten die Klassifikatoren auf Basis verschieden stark verrauschter Positionsdaten bis zu fünf Verkehrsteilnehmertypen zu unterscheiden lernen, allerdings stellte sich im Verlauf der Arbeit heraus, dass auf Basis der Simulation höchstens drei Typen (*Fußgänger*, *Fahrradfahrer* und *motorisiertes Fahrzeug*) unterschieden werden können. Vor der Klassifikation wurden die über ein festes Zeitintervall von einer Minute gesammelten Positionsdaten in lange Eingabevektoren weiterverarbeitet, die die Bewegung der Verkehrsteilnehmer innerhalb dieses Intervalls repräsentieren. Eine Reduktion dieser Eingabevektoren fand nicht statt. Bei der anschließenden Klassifikation wurden die besten Ergebnisse durch die SVM erzielt, welche einen Genauigkeitswert von 93,9% auf den unverrauschten Daten erreichen konnte. Ein Verrauschen der Daten, um diese vermeintlich realistischer zu machen, führte in allen Fällen zu einem starken Abfall der Qualität der Klassifikation.

Auch Dietmar Fischer fokussiert sich in seiner Masterarbeit [30] von 2023 primär auf die Nutzung von Daten, die ohne Verfälschung der Verkehrssimulation CARLA entnommen wurden,

um darauf basierend ein System zu entwickeln, welches durch den Einsatz von neuronalen Netzen dazu in der Lage ist, Verkehrsteilnehmer erneut in drei Typen und zusätzlich entsprechend ihrer Relevanz für den *Hero* (das eigene Fahrzeug) zu klassifizieren. Dabei untersucht Fischer im Wesentlichen die Performanz zweier Arten von neuronalen Netzen: Zum einen von Feed-Forward-Netzen (FFN) und zum anderen von *Deep-Q-Learning-Netzen* (DQLN). Letztere zählen anders als die anderen erprobten Ansätze nicht mehr zu den überwachten, sondern zu den bestärkenden Lernverfahren. Im Zuge der Datenvorverarbeitung setzte Fischer anders als Sohl auf eine Reduktion der Eingabedimension auf 16 deskriptive Statistiken, welche die Bewegungsmerkmale des Verkehrsteilnehmers über ein festes Intervall von einer Minute zusammenfassen. Die Ergebnisse der Arbeit zeigen, dass der DQLN-Ansatz bei der Typklassifikation einen Genauigkeitswert von 90% auf den Simulationsdaten erreichen konnte. Das FFN erreichte 93%. Darüber hinaus erprobte Fischer die Modelle mit einem kleinen Datensatz an realen Positionssequenzen, wobei zumindest das beste FFN einen guten Genauigkeitswert von etwa 90% erzielen konnte. Der unternommene Versuch auch die Relevanz von anderen Fahrzeugen aus Sicht des Heros in Risikokategorien zu klassifizieren, hat zu keinen nennenswerten Ergebnissen geführt.

Anders als bei Fischer beschränkt sich die vorliegende Arbeit erneut lediglich auf die Klassifikation von Verkehrsteilnehmern in verschiedene Typen. Im Gegensatz zu allen vorangegangenen Arbeiten legt diese jedoch den Fokus nicht mehr auf Simulationsdaten, die innerhalb von CARLA gewonnen wurden, sondern komplett auf reale Daten, welche durch Teilnehmer des Straßenverkehrs erzeugt werden und potenziell weiterer Vorverarbeitungen bedürfen. Des Weiteren soll erstmalig eine Unterscheidung verschiedener motorisierter Fahrzeuge untersucht und hierbei auf bisher nicht erprobte maschinelle Lernverfahren zurückgegriffen werden.

3.2 Vehicle Classification from Low-Frequency GPS Data with Recurrent Neural Networks

Die Arbeit *Vehicle Classification from Low-Frequency GPS Data with Recurrent Neural Networks* von Matteo Simoncini et al. [15], die im Jahr 2018 veröffentlicht wurde, untersucht die Klassifikation von motorisierten Fahrzeugen in drei Kategorien: *leichte, mittelschwere* und *schwere Fahrzeuge*. Basis hierfür bilden niederfrequente GNSS-Sequenzen mit einem variierendem Aufnahmeintervall von durchschnittlich 90 Sekunden.

Der zugrundeliegende Datensatz ist äußerst umfangreich. Er umfasst etwa eine Million Sequenzen, die durch insgesamt 55 Millionen GNSS-Punkte eine Strecke von 56 Millionen gefahrenen Kilometern abbilden. Die Sequenzen wurden so vorverarbeitet, dass für alle paarweise aufeinanderfolgenden GNSS-Punkte die *gefahrene Strecke*, die *Luftlinie*, die *verstrichene Zeit*, die *Geschwindigkeit* und *Intervallgeschwindigkeit*, die *Beschleunigung* und *Intervallbeschleunigung* sowie der *Straßentyp* vorlag. Globale Merkmale oder deskriptive Statistiken wurden nicht berechnet. Die für die Klassifikation verwendeten Sequenzen umfassten mindestens 20 Zeitschritte, was einem Zeitfenster von etwa 30 Minuten entspricht.

Für die Klassifikation wurden verschiedene rekurrente neuronale Netze mit LSTM-Schichten entworfen. Dabei kamen auch Feed-Forward-Schichten zum Einsatz, die vor und nach den LSTM-Schichten platziert wurden, was den Autoren zufolge die Vorhersageleistung der Netze deutlich verstärkte. Das beste Modell erreichte auf den Testdatensatz eine Sensitivität von 85% für leichte und 93% für schwere Fahrzeuge. Allerdings hat das Modell Probleme in der Unterscheidung von mittelschweren und leichten Fahrzeugen, wodurch es für mittelschwere Fahrzeuge lediglich eine Trefferquote von 48% erzielen konnte.

Nichtsdestotrotz zeigt die Arbeit, dass der Einsatz von RNNs durch ihre Fähigkeit, sequenzielle Daten zu verarbeiten, ein vielversprechendes Verfahren für die Klassifikation von Fahrzeugen auf Basis von GNSS-Sequenzen ist. Die Autoren heben dabei in ihrem Fazit hervor, dass die Nutzung von höherfrequenten GNSS-Sequenzen bei einem ähnlichen Ansatz das Potenzial besitzt, die aufgezeigten Schwächen zu überwinden und Fahrzeuge zuverlässiger und auch schneller zu klassifizieren.

3.3 Vehicle Classification using GPS Data

Im Zuge ihrer Arbeit *Vehicle Classification using GPS Data* setzten sich Zhanbo Sun und Xuegang Ban [31] bereits im Jahr 2013 mit der Klassifikation von Fahrzeugen auf Basis von GNSS-Sequenzen auseinander. Untersucht wird hierbei ein binäres Klassifikationsproblem: Die Unterscheidung zwischen PKWs und LKWs, basierend auf einem Datensatz, der 52 PKW- und 84 LKW-Sequenzen mit Längen von 15-20 Minuten umfasst. Alle genutzten Sequenzen bilden lediglich Fahrten auf Zubringerstraßen ab, die mit einem Aufnahmeintervall von etwa drei Sekunden aufgenommen wurden.

Aus den Sequenzen wurden im Zuge der Arbeit verschiedene globale Merkmale mit Geschwindigkeits- sowie Beschleunigungs- und Verzögerungsbezug extrahiert. Anschließend wurden

verschiedene Modelle zur Merkmalsauswahl (engl. *Feature Selection*) und Klassifikation entwickelt und deren Ergebnisse veranschaulicht und ausgewertet. Die Klassifikationsmodelle basierten hierbei auf Support Vector Machines mit quadratischen Kernel-Funktionen.

Die Autoren heben hervor, dass Merkmale, die das Beschleunigungs- und Verzögerungsverhalten der Fahrzeuge abbilden, sich besser zur Klassifikation verschiedener motorisierter Fahrzeuge eignen als geschwindigkeitsbasierte Merkmale. Am effektivsten waren hierbei der Anteil von Beschleunigungs- und Verzögerungswerten über 1 $m/s\hat{}2$ und die Standardverteilungen der Beschleunigung und Verzögerung. Unter der Nutzung dieser vier Merkmale konnte durch das beste SVM-Modell eine Fehlklassifizierungsrate von nur 4,2% (bzw. eine Genauigkeit von 95,8%) auf den Testdaten erreicht werden.

4 Datengrundlage

4.1 Anforderungen

Das Ziel dieser Arbeit ist die Erforschung der Klassifikation von Verkehrsteilnehmern auf Basis von sequenziellen Positionsdaten durch den Einsatz von maschinellen Lernverfahren. Grundlage für die erfolgreiche Umsetzung dieser Klassifikation ist, wie in allen Projekten des maschinellen Lernens, eine für die vorliegende Aufgabenstellung geeignete Datenbasis.

Entsprechend der Zielsetzung muss die Datenbasis eine möglichst umfangreiche Sammlung an gelabelten GNSS-Sequenzen aufweisen, welche die Bewegung verschiedener Teilnehmer des Straßenverkehrs unter Realbedingungen repräsentiert. Um sinnvoll mit diesen Sequenzen arbeiten zu können, sollten diese dabei mindestens eine Minute, idealerweise jedoch eine deutlich größere Zeitspanne umfassen. Zwischen zwei aufeinanderfolgenden Positionsaufnahmen einer einzelnen Sequenz sollte außerdem ein geringes und möglichst immer gleich großes Zeitintervall liegen. Gut geeignet ist beispielsweise eine Aufnahmefrequenz im Bereich von 0,5-2 Hertz, was einem Aufnahmeintervall von 2 bis 0,5 Sekunden entspricht.

Eine weitere nicht zu vernachlässigende Anforderung ist eine möglichst hohe Repräsentativität der Datenbasis. Dies umfasst verschiedene Aspekte. Zum Beispiel die Anzahl an Personen, welche an der Datengewinnung beteiligt waren, oder die gegebenen Klassen von Verkehrsteilnehmern und deren Anteil an den Gesamtdaten. Auch die Abbildung von verschiedenen und möglichst vielseitigen Verkehrssituationen und -umständen spielt eine entscheidende Rolle für die Qualität der umzusetzenden Klassifikatoren.

4.2 Datengewinnung

Der naheliegendste Ansatz der Datengewinnung liegt in der Beschaffung und Nutzung bereits bestehender Datensätze. Allerdings konnte kein frei verfügbarer Datensatz gefunden werden, der sich entsprechend der im letzten Abschnitt erhobenen Anforderungen auf die vorliegende Aufgabenstellung anwenden ließe. Während der Suche nach einem solchen Datensatz wurde insbesondere ein Mangel an Datensätzen, welche die Bewegung vulnerablerer Verkehrsteilnehmer (Fußgänger, Fahrradfahrer, Motorradfahrer) abbilden und hochfrequente GNSS-Sequenzen enthalten, festgestellt. Dies machte es notwendig, die Datenbasis für das vorliegende Klassifikationsproblem selbst zu erzeugen, was bei strenger Beachtung aller Anforderungen den praktischen Rahmen dieser Arbeit gesprengt hätte. Deshalb wurden einige Einschränkungen

festgelegt, um den Aufwand der Datenerhebung auch für eine kleine Gruppe an freiwilligen Teilnehmern umsetzbar zu halten.

Die wichtigste Einschränkung ist dabei die Begrenzung der Klassifikation auf lediglich vier Verkehrsteilnehmertypen: *Fußgänger*, *Fahrräder*, *Motorräder* und *Autos*. Hierdurch ist die Komplexität des Klassifikationsproblems für eine Umsetzbarkeit im Zuge dieser Arbeit hinreichend eingeschränkt, allerdings werden dennoch wichtige vulnerable und vor allem verschiedene motorisierte Verkehrsteilnehmertypen betrachtet. Zusätzlich wurden auch örtliche Einschränkungen festgelegt. Diese umfassten zum einen die Vorgabe, zunächst lediglich innerörtlichen Verkehr zu betrachten, und zum anderen die Beschränkung des Gebietes der Datenerhebung auf die Region *Südthüringen* und *Oberfranken*, in welcher die Hochschule Coburg lokalisiert ist.

Für die Erhebung der Daten kam die Mobile App *MotionTrace* zum Einsatz, welche im Zuge des Forschungsprojektes zur erweiterten Umfeldwahrnehmung autonomer Fahrzeuge an der Hochschule Coburg entwickelt wurde. Die Kernfunktion der App besteht in der Aufnahme von gelabelten GNSS-Sequenzen, welche die Bewegung – oder vielmehr die Positionsänderung – des ausführenden Mobilgeräts über einen beliebigen Zeitraum abbilden können. Hierbei wird ein Aufnahmeintervall von einer Sekunde zwischen zwei aufeinanderfolgenden Positionsaufnahmen genutzt. Abb. 16 zeigt einen Screenshot der Benutzeroberfläche von MotionTrace während einer Aufnahme, die durch den Nutzer mit dem Label für die Klasse *Auto* versehen wurde.

Die durch MotionTrace aufgezeichneten Sequenzen können synchron zur Aufnahme als CSV-Dateien auf dem Gerät gespeichert oder im JSON-Format an einen Server gesendet und von dort abgerufen werden. Eine einzelne Positionsaufnahme (Zeile) in der zu einer Aufnahme zugehörigen CSV-Datei umfasst dabei die in Tab. 1 aufgeführten Merkmale (Spalten).

Bezeichner	**Datentyp**	**Erläuterung**
`idx`	Integer	Laufender Index, der die Reihenfolge der Positionsaufnahmen innerhalb einer Sequenz eindeutig festlegt
`type`	Integer	Label/Klasse des aufzeichnenden Verkehrsteilnehmers: 0 = Fußgänger, 1 = Fahrrad, 2 = Motorrad, 3 = Auto (, …)
`lat`	Double	Die geographische Breitenkoordinate in Dezimalgrad
`lon`	Double	Die geographische Längenkoordinate in Dezimalgrad
`accuracy`	Double	Die geschätzte radiale Genauigkeit der aufgezeichneten Koordinaten in Meter

Bezeichner	Datentyp	Erläuterung
`time`	Integer	Zeitstempel der Koordinatenaufzeichnung in Millisekunden seit der Unix-Epoche (01.01.1970 00:00:00 Uhr)

Tab. 1: Spalten der von MotionTrace erzeugten CSV-Dateien (Aufnahmen)

Insgesamt konnten sechs freiwillige Teilnehmer gewonnen werden, welche sich an der Datenerhebung beteiligten und ihre Bewegungen als Fußgänger, Fahrradfahrer, Motorradfahrer oder Autofahrer im Straßenverkehr über einen Zeitraum von etwa zwei Monaten regelmäßig mittels MotionTrace aufgenommen haben.

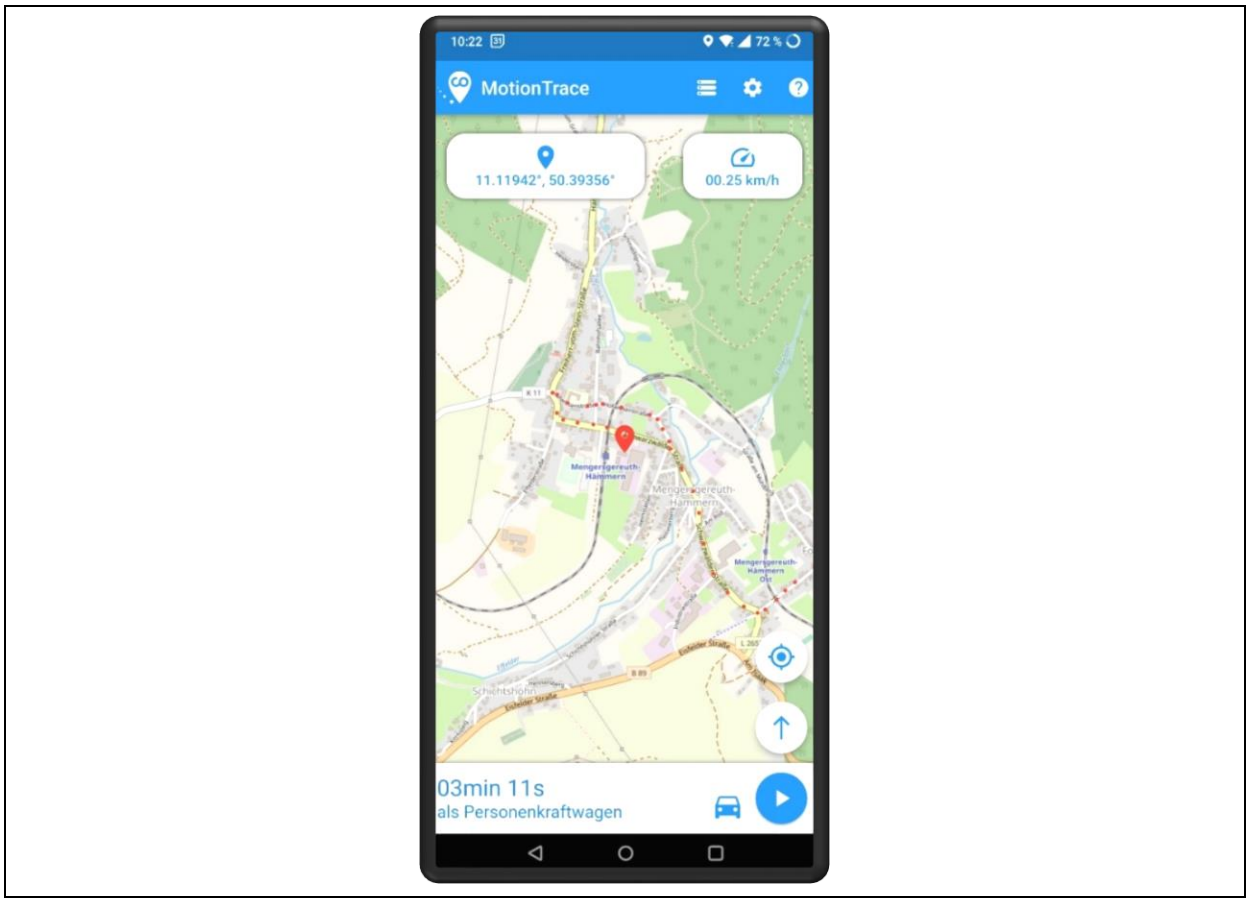

Abb. 16: Screenshot der MotionTrace-App

4.3 Betrachtung der Ausgangsdaten

Insgesamt konnten 165 Aufnahmen mit einer Gesamtlänge von ca. 32 Stunden als Datenbasis für diese Arbeit gesammelt werden. Die Verteilung dieser 32 Stunden auf die einzelnen Verkehrsteilnehmertypen ist hierbei Abb. 17 zu entnehmen. Demnach sind in etwa 34,6% der Ausgangsdaten der Klasse *Auto*, 24,5% der Klasse *Fahrrad*, 20,3% der Klasse *Motorrad* und 20,6%

der Klasse *Fußgänger* zugeordnet. Die Daten sind somit, insbesondere in Bezug auf die Klasse *Auto*, nicht ausgeglichen.

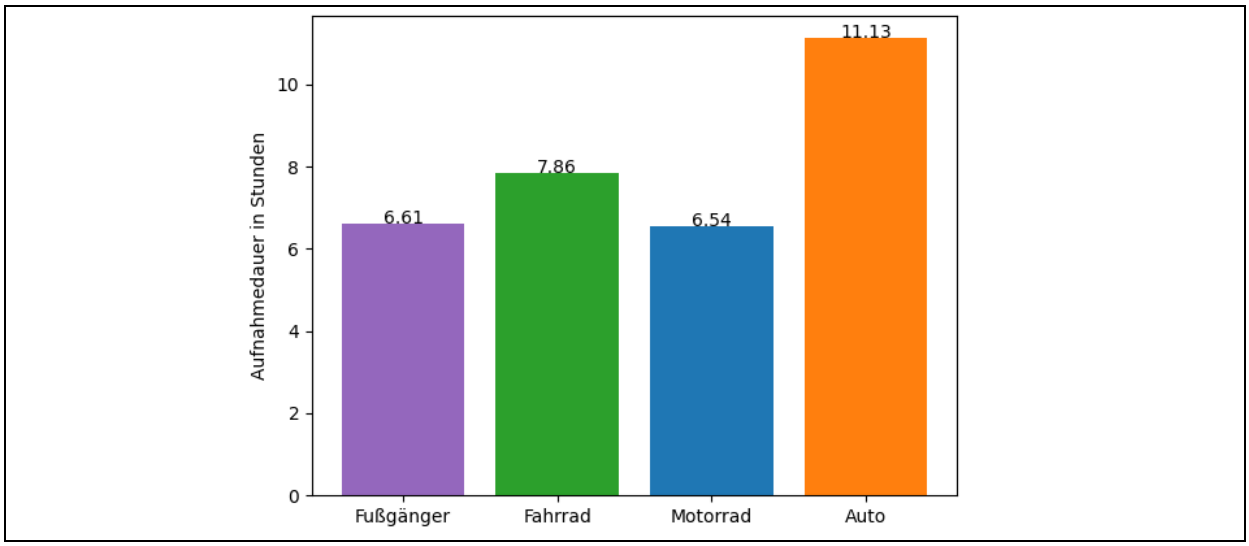

Abb. 17: Umfang der Datenbasis (in Stunden) nach Verkehrsteilnehmerklasse

Visualisiert man die gesammelten Aufnahmen auf einer Karte, so fällt auf, dass ein großer Teil der Daten nur wenigen Städten und Gemeinden entstammt. Zu diesen gehören die Mittelstädte *Sonneberg* und *Coburg*, die Kleinstädte *Kronach*, *Neustadt bei Coburg* und *Rödental* sowie das Dorf *Mengersgereuth-Hämmern*. Vereinzelte Aufnahmen entstammen anderen Ortschaften aus der Region Südthüringen und Oberfranken. Konkret ist das Aufnahmegebiet geographisch in der Breite auf das Intervall von 49,8894° bis 50,4174° und in der Länge auf das Intervall von 10,8821° bis 11,3786° begrenzt. Entsprechend beschränken sich die gesammelten Daten auf mittelstädtischen, kleinstädtischen und dörflichen Verkehr. Großstadtverkehr wird nicht abgebildet.

Die Mehrheit der Aufnahmen wurde im Raum Sonneberg und Neustadt bei Coburg aufgezeichnet. Die in Abb. 18 dargestellte Visualisierung zeigt dabei, dass die Ausgangsdaten nicht nur verschiedene städtische Bereiche, wie bspw. Wohn- und Industriegebiete, sondern auch vielfältige Verkehrswege, wie Haupt- und Nebenstraßen, aber auch Fußgänger- und Fahrradwege sowie Parks abdecken. Nicht in der Abbildung zu erkennen, aber dennoch hervorzuheben, ist, dass durch einige der gesammelten Aufnahmen auch besondere Verkehrsumstände, wie bspw. Staus zu Stoßzeiten oder stark gefüllte Fußgängerzonen, in den Ausgangsdaten abgebildet werden.

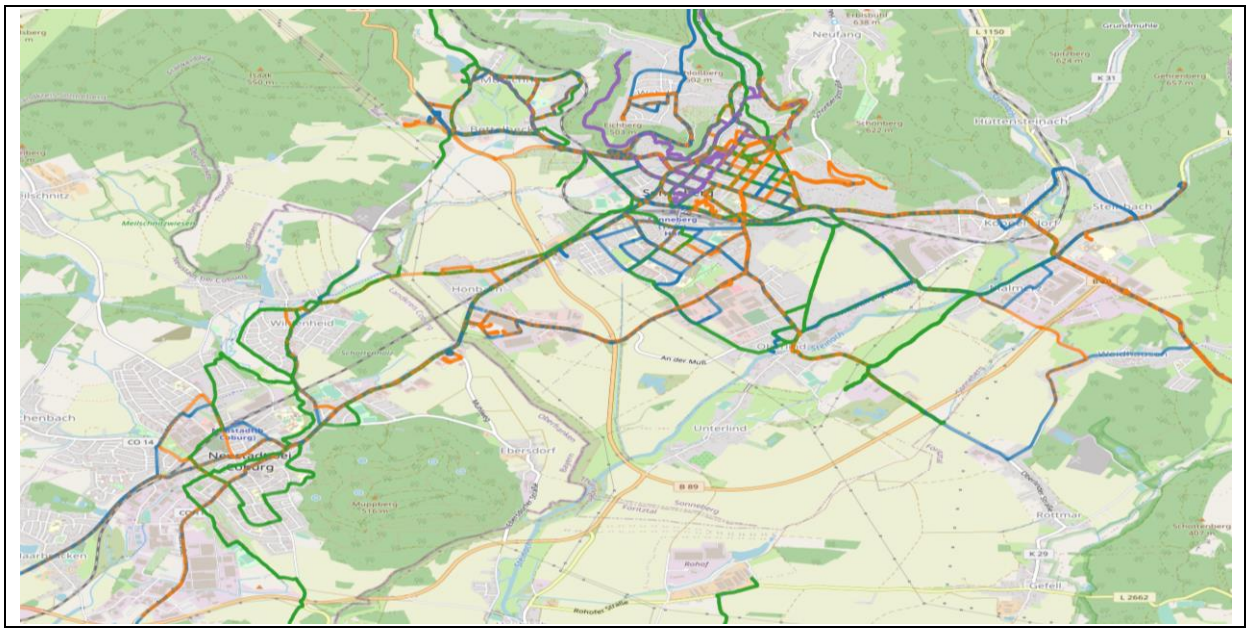

Abb. 18: Visualisierung der gesammelten Aufnahmen im Raum Sonneberg und Neustadt bei Coburg

Die einzelne Betrachtung der Aufnahmen bestätigt die Erkenntnis, die zum Teil bereits aus den vorangegangen Forschungsarbeiten gewonnen werden konnte: Weite Teile der gesammelten GNSS-Sequenzen sind zu einem gewissen Grad fehlerbehaftet. Dabei werden im Zuge dieser Arbeit insbesondere vier Arten von Fehlern differenziert. Abb. 19 veranschaulicht diese Fehlerarten schematisch im Vergleich zu einer fehlerfreien Sequenz, die einer Fahrt auf einer geraden Straße bei konstanter Geschwindigkeit entspricht. Eine Betrachtung konkreter Beispiele würde an dieser Stelle den Rahmen sprengen.

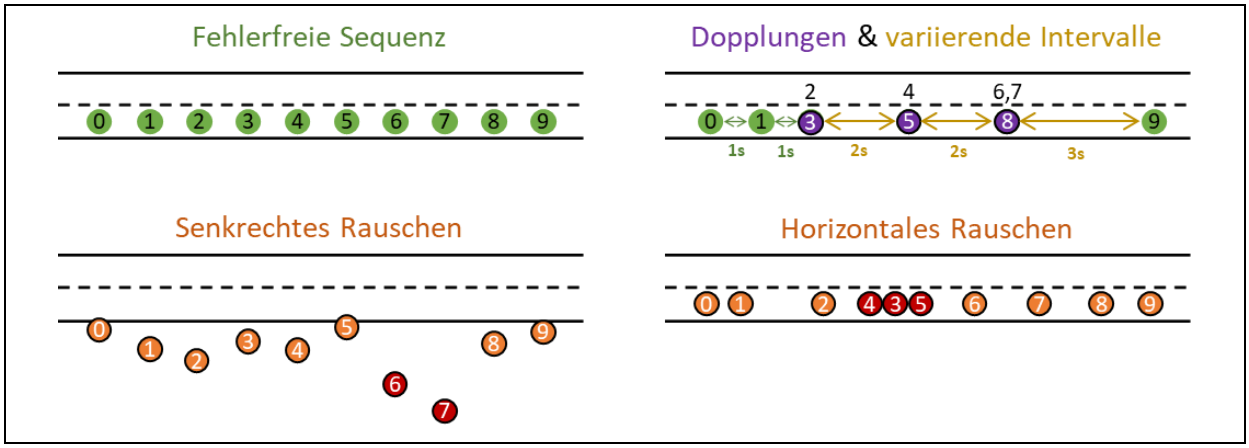

Abb. 19: Primäre Fehlerarten innerhalb der GNSS-Sequenzen

Die in den Ausgangsdaten am häufigsten auftretenden Fehler sind Ungenauigkeiten in der Positionserfassung, welche im Folgenden zusammenfassend als *Rauschen* bezeichnet werden. Unterschieden werden hierbei zwei Formen des Rauschens: das *senkrechte Rauschen* und das *horizontale Rauschen*. Das Erstere führt dazu, dass erfasste Koordinatenpunkte abseits der eigentlich zurückgelegten Strecke liegen. Dieses Rauschen scheint dabei allerdings in der Regel nicht für jeden Punkt unabhängig zu sein, denn aufeinanderfolgende Punkte weisen oft ähnliche Abweichungen auf. Sie liegen also, wie beispielhaft unten links in Abb. 16 dargestellt, alle rechts von der tatsächlich zurückgelegten Strecke. Das horizontale Rauschen äußert sich dadurch, dass Koordinatenpunkte nicht abseits, sondern entlang der zurückgelegten Strecke vor oder hinter der tatsächlichen Position erfasst werden. In seltenen Fällen sind einzelne Punkte dabei so stark horizontal verrauscht, dass sie trotz einer konstanten Vorwärtsbewegung hinter zuvor aufgenommenen Punkten erfasst werden. Wichtig zu beachten ist, dass beide Arten des Rauschens in der Regel gleichzeitig und in allen aufgenommenen Sequenzen auftreten. Der Schweregrad des Rauschens variiert dabei jedoch sehr stark.

Eine Möglichkeit, um das Rauschen in den Ausgangsdaten abzuschätzen, ist durch die `accuracy`-Werte der Positionsaufnahmen gegeben, deren Mittelwert über alle Aufnahmen bei ca. 6,86 Meter liegt. Die zugehörige Standardabweichung beträgt in etwa 4,26 Meter. Damit liegt die Genauigkeit der meisten Koordinatenpunkte am unteren Ende der erwarteten Genauigkeit des GNSS, welche als 5-15 Meter angegeben wurde. Einige Aufnahmen weißen jedoch auch `accuracy`-Werte von weit über 20 Meter auf. Nichtsdestotrotz lässt sich die Genauigkeit der Ausgangsdaten somit als verhältnismäßig hoch beurteilen.

Eine weitere Fehlerart, die in vielen Aufnahmen auftritt, sind *Dopplungen*. Hiermit ist das mehrfache Erfassen derselben Position und in der Regel auch desselben zugehörigen Zeitstempels gemeint. Dopplungen treten vermehrt auf, wenn langsame Bewegungen oder Stillstand aufgenommen wurden. Hierbei kann es vorkommen, dass der Standort-Service des Mobilgeräts bei der Aufzeichnung mittels MotionTrace keine Änderung der Position feststellt und diese entsprechend nicht innerhalb des Aufnahmeintervalls aktualisiert. Eine direkte Folge von Dopplungen sind, wie auch in Abb. 19 veranschaulicht, *variierende Zeitintervalle*. Diese können jedoch auch unabhängig von Dopplungen auftreten. Neben Dopplungen ist ein weiterer Grund für variierende Zeitintervalle möglicherweise die fehlende Kopplung zwischen der Aktualisierung des Standort-Services und der Standortanfrage und -speicherung durch MotionTrace.

Die Zeitdifferenzen der aufeinanderfolgenden Positionsaufnahmen in den Ausgangsdaten (berechnet aus den Zeitstempeln) weisen eine Standardabweichung von ca. 305 Millisekunden und einige Ausreißer von bis zu 15 Sekunden auf. Ausreiser sind vermutlich die Folge von Störungen des GNSS-Signals unter bestimmten Umständen, wie bspw. einer Tunnelfahrt. In der Regel entspricht das Aufnahmeintervall jedoch in etwa der angedachten Sekunde, was sich aus dem 25%-Quantil von 998 Millisekunden und dem 75%-Quantil von 1011 Millisekunden der Zeitdifferenzen schlussfolgern lässt.

Für eine weitere Betrachtungen der Ausgangsdaten sind alle gesammelten und für die nachfolgenden Umsetzungen verwendeten Aufnahmen im zu dieser Arbeit zugehörigen Projekt zu finden. Die einzelnen Aufnahmen liegen hierbei als separate CSV-Dateien vor, welche die in Tab. 1 aufgeführten Spalten umfassen.

5 Anforderungen und Gesamtkonzept

Zur Bewältigung der u. a. in Abschnitt 1.1 erläuterten Problemstellung und zur Beantwortung der hieraus abgeleiteten Forschungsfragen wurde das in Abb. 20 veranschaulichte Gesamtkonzept entwickelt. Dieses bildet die Basis der in den nachfolgenden Kapiteln beschriebenen Implementierungen zu dieser Arbeit.

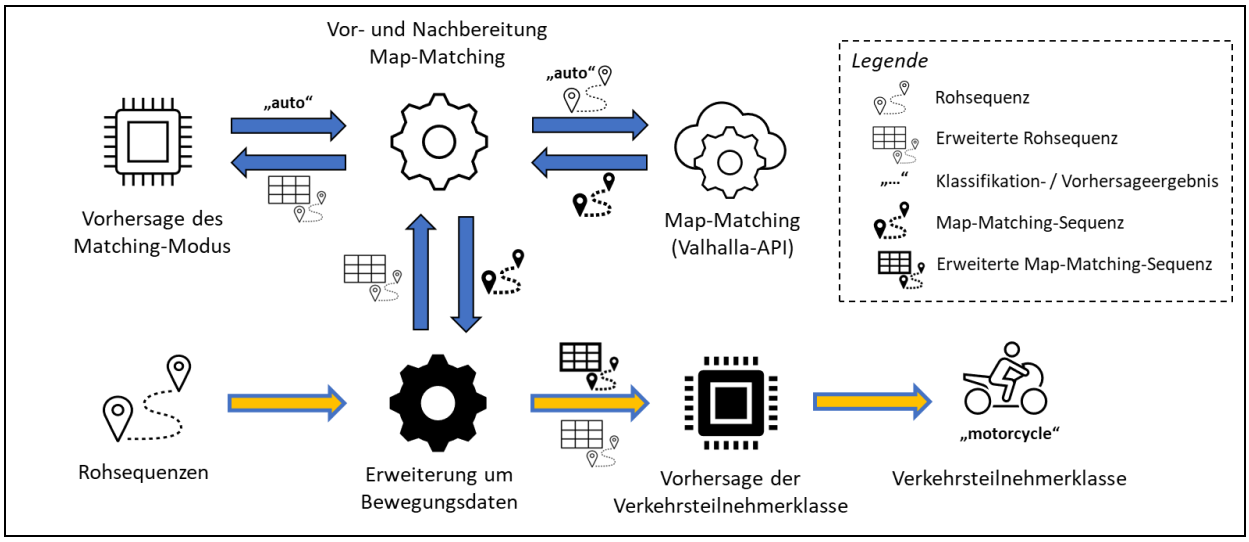

Abb. 20: Veranschaulichtes Gesamtkonzept der Verkehrsteilnehmerklassifikation

Gelb hervorgehoben ist dabei das Basiskonzept der Verkehrsteilnehmerklassifikation. An dessen Beginn stehen zunächst die Rohsequenzen, welche im Grunde Zeitreihen aus Positionsaufnahmen sind. Jede Positionsaufnahme enthält dabei die Koordinaten eines Verkehrsteilnehmers und deren geschätzte Genauigkeit zu einem bestimmten Zeitpunkt, welcher über einen Zeitstempel abgebildet wird. In dieser Form eignen sich die Sequenzen nicht für die Klassifikation, da sie einem starken *Bias* (dt. etwa *Verzerrung*) unterliegen. Dieser ergibt sich vor allem aus der starken lokalen und temporalen Beschränktheit der Datenbasis sowie aus der geringen Menge an Datenerzeugern. Hierdurch könnten die zu trainierenden Klassifikatoren fälschlicherweise lernen, Fahrzeugklassen innerhalb der Datenbasis allein auf Basis von Ort (z. B. zurückgelegten Strecken) und Zeit (Aufnahmen eines Nutzers entstanden oft mit kurzem zeitlichem Abstand zueinander) statt über Bewegungsmuster zu unterscheiden. In jedem Fall kann davon ausgegangen werden, dass die Klassifikatoren so nur sehr schwer auf neue Daten außerhalb dieser lokalen und temporalen Beschränktheit verallgemeinern könnten. Deswegen soll keines der oben genannten Merkmale (Koordinaten, Genauigkeit und Zeitstempel) direkt in die Klassifikation einfließen. Stattdessen werden hieraus verallgemeinernde Merkmale berechnet,

welche die Bewegung von Verkehrsteilnehmern unabhängig von Ort und Zeit repräsentieren sollen. Konkret werden die Positionsaufnahmen innerhalb der Sequenzen um Merkmale, wie Geschwindigkeit, Beschleunigung und Winkelgeschwindigkeit erweitert. Die Klassifikation der Verkehrsteilnehmerklasse erfolgt anschließend allein auf Basis dieser Bewegungsmerkmale. Als Klassifikatoren zur Vorhersage der Verkehrsteilnehmerklasse kommen hierbei basierend auf den Erkenntnissen von Matteo Simoncini et al. [15] rekurrente neuronale Netze zum Einsatz, deren Alleinstellungsmerkmal es ist, Sequenzen zu verarbeiten, ohne diese vorher auf einen einzelnen Datenpunkt zusammenfassen zu müssen. Dadurch können die Sequenzen durch die Klassifikatoren auch tatsächlich als solche betrachtet und ein Verlust potenziell wichtiger zeitcodierter Informationen vermieden werden. Im Zuge dieser Arbeit wird hierbei auch untersucht, inwieweit die Länge der eingegebenen Sequenzen einen Einfluss auf die Vorhersagequalität durch die RNNs besitzt, indem der Klassifikation ein-, zwei- und vierminütige Sequenzen zugrunde gelegt werden.

Die Betrachtung der Ausgangsdaten im vorangegangenen Kapitel hat gezeigt, dass Positionssequenzen verschiedene Arten von Fehlern enthalten. Da diese möglicherweise die Qualität der Klassifikation beeinflussen, werden basierend auf dem Basiskonzept verschiedene Möglichkeiten der Datenbereinigung untersucht. Zur Behandlung der Verrauschungsfehler, insbesondere des senkrechten Rauschens, kommt hierbei der Map-Matching-Service von Valhalla zum Einsatz. Dafür ist jedoch eine Erweiterung des Gesamtkonzeptes notwendig, welche in Abb. 20 blau hervorgehoben ist. Anders als beim Basiskonzept werden die um Bewegungsdaten erweiterten Sequenzen nicht direkt zur Klassifikation der Verkehrsteilnehmerklasse eingesetzt, sondern zunächst vorklassifiziert. Die Vorklassifikation dient der Vorhersage des von Valhalla einzusetzenden Matching-Modus. Wie bereits in Unterabschnitt 2.7.1 erwähnt, werden hierbei drei verschiedene Modi unterschieden, welche das Map-Matching durch Valhalla entweder für Fußgänger, Fahrradfahrer oder motorisierte Straßenfahrzeuge optimieren. Da es sich bei der Vorhersage des Matching-Modus um ein vereinfachtes Klassifikationsproblem (keine Unterscheidung motorisierter Fahrzeuge) handelt, kommt für diese ein weniger komplexes Klassifikationsverfahren als bei der Vorhersage der Verkehrsteilnehmertypen zum Einsatz. Als Eingabe dient dabei eine Zusammenfassung der Bewegungsdaten der vorliegenden Sequenz auf Basis verschiedener deskriptiver Merkmale. Die Wahl des eingesetzten Klassifikationsverfahrens basiert auf einer vorausgehenden Evaluierung des SVM- und Random-Forest-Verfahrens.

Nach der Bestimmung des voraussichtlich optimalen Matching-Modus, werden auf Basis der erweiterten Sequenz einige weitere Parameter zur Optimierung des Map-Matchings bestimmt.

Diese werden gemeinsam mit dem Modus und der Rohsequenz über eine Webschnittstelle an eine Valhalla-Instanz übergeben. Zurückgeliefert wird eine Sequenz, welche durch Map-Matching auf ein digitales Straßen- und Wegenetzt abgebildet wurde. Für diese Sequenz müssen anschließend die Bewegungsdaten neu berechnet werden, bevor sie schlussendlich zur finalen Vorhersage der Verkehrsteilnehmerklasse eingesetzt werden kann.

Zur Behandlung von horizontalem Rauschen wird zusätzlich ein weiterer Ansatz untersucht: Die künstliche Erhöhung des Aufnahmeintervalls auf zwei Sekunden, indem jede zweite Positionsaufnahme in den Ausgangssequenzen verworfen wird. Die Idee dahinter ist, dass durch die Erhöhung des Aufnahmeintervalls höhere zeitliche und örtliche Abstände zwischen den einzelnen Positionsaufnahmen entstehen, die dazu führen, dass das relative Rauschen in den Sequenzen und damit dessen Einfluss auf die berechneten Bewegungsmerkmale ggf. geringer wird. Eine Erhöhung auf mehr als zwei Sekunden wird jedoch nicht untersucht, da davon ausgegangen werden kann, dass dann der Informationsgehalt der berechneten Bewegungsmerkmale, insbesondere der Beschleunigung und Winkelgeschwindigkeit, reduziert wird.

Dopplungen und variierende Aufnahmeintervalle sollen ebenfalls behandelt werden. Dies geschieht jedoch nicht im Zuge der Datenvorverarbeitung, sondern indem neben den bewegungsbasierten Merkmalen auch zeit- bzw. intervallbasierte Merkmale in die Eingabe der Klassifikatoren aufgenommen werden.

6 Datensatzgenerierung und -erweiterung

Als Grundlage zur Untersuchung der Auswirkungen verschiedener Sequenzlängen und Aufnahmeintervalle auf die Qualität der Verkehrsteilnehmerklassifikation werden aus der Datenbasis, also allen gesammelten MotionTrace-Aufnahmen, mehrere Datensätze erzeugt. Der prinzipielle Ablauf bei der Erzeugung der einzelnen Datensätze ist dabei immer gleich. Zunächst werden alle gesammelten Aufnahmen als Dataframe eingelesen, in gleich lange Sequenzen aufgeteilt und ggf. durch eine Reduzierung auf jeden zweiten Datenpunkt heruntergetaktet. Aufnahmen, welche kürzer als die geforderte Sequenzlänge sind, werden an dieser Stelle verworfen. Dasselbe gilt für zu kurze Sequenzen, welche sich aus der Aufteilung längerer Aufnahmen ergeben (i. d. R. am Ende der Aufnahme). Alle übrigen Sequenzen werden in der Liste `df_list` gespeichert. Diese wird im nächsten Schritt dazu genutzt, die Sequenzen stratifiziert, also unter Beibehaltung des Klassenverhältnisses, auf Basis der zugehörigen Labels im Verhältnis 75:25 in Trainings- und Testdaten aufzuteilen. Zur Aufteilung wird die in Code 1 verwendete scikit-learn Funktion `train_test_split` verwendet. Anschließend werden die Trainings- und Testdaten in jeweils eine CSV-Datei geschrieben, wobei einzelne Sequenzen über die zusätzliche Spalte `recording_nr` indiziert werden. Abgesehen davon unterscheiden sich die Spalten der CSV-Dateien zunächst nicht von denen der Aufnahmen (siehe Abschnitt 4.2).

```
train_dfs, test_dfs,_,_= train_test_split(df_list, labels,
  test_size=0.25, stratify=labels, random_state=42)
```

Code 1: Stratifizierte Aufteilung der Sequenzen in Trainings- und Testdaten

Insgesamt werden auf diese Weise sechs Datensätze generiert, welche in Tab. 2 aufgelistet sind. Diese decken die Kombinationen der im letzten Kapitel festgelegten Sequenzlängen von einer, zwei oder vier Minuten und der Aufnahmeintervalle von einer oder zwei Sekunden ab.

Bezeichner	**Intervall**	**Datenpunkte pro Sequenz**	**Sequenzen in den Trainingsdaten**	**Sequenzen in den Testdaten**
1s_1min	1s	60	1368	456
1s_2min	1s	120	652	218
1s_4min	1s	240	294	99
2s_1min	2s	30	1368	456

Bezeichner	Intervall	Datenpunkte pro Sequenz	Sequenzen in den Trainingsdaten	Sequenzen in den Testdaten
2s_2min	2s	60	652	218
2s_4min	2s	120	294	99

Tab. 2: Datensätze mit verschiedenen Aufnahmeintervallen und Sequenzlängen

Die einzelnen Datensätze bzw. die darin enthaltenen Trainings- und Testsequenzen werden anschließend erweitert. Hierfür wird im Wesentlichen über alle gegebenen Sequenzen der verschiedenen Datensätze iteriert, wobei währenddessen schrittweise die in Tab. 3 aufgeführten Bewegungsmerkmale berechnet werden. Die zugrundeliegenden Funktionen zur Berechnung der einzelnen Merkmale basieren auf der Arbeit von Torlak [28].

Bezeichner	Datentyp	Erläuterung
`timediff`	Integer	Die Zeitdifferenz in ms zwischen der betrachteten und der letzten Positionsaufnahme, welche sich aus den zugehörigen `time`-Werten berechnet.
`velocity`	Double/NULL	Die Geschwindigkeit in m/s, welche sich aus der Distanz zwischen den Koordinaten der aktuellen und letzten Positionsaufnahme sowie dem `timediff`-Wert berechnet.
`acceleration`	Double/NULL	(Positive) Beschleunigung in m/s^2, welche sich aus dem aktuellen und letzten `velocity`-Wert und dem Mittelwert der zugehörigen `timediff`-Werte berechnet.
`deceleration`	Double/NULL	Verzögerung (negative Beschleunigung) in m/s^2, welche sich aus dem aktuellen und letzten `velocity`-Wert und dem Mittelwert der zugehörigen `timediff`-Werte berechnet.
`heading`	Double/NULL	Die Bewegungsrichtung in Grad (0° ≙ Norden, 90° ≙ Osten, 180° ≙ Süden und 270° ≙ Westen), welche sich aus den Koordinaten der aktuellen und letzten Positionsaufnahme berechnet.
`angular_speed`	Double/NULL	Winkel- bzw. Drehgeschwindigkeit in $Grad/s$, welche sich aus dem aktuellen und letzten `heading`-Wert und dem Mittelwert der zugehörigen `timediff`-Werte berechnet.

Tab. 3: Bewegungsmerkmale, um welche die Sequenzen erweitert werden

Zu beachten ist, dass die meisten der Bewegungsmerkmale unter bestimmten Umständen NULL-Werte annehmen können. Beispielsweise ist der erste Wert aller Merkmale einer Sequenz (mit Ausnahme von `timediff`) immer NULL. Bei `acceleration`, `deceleration` und `angular_speed` sind es sogar die ersten beiden Werte. Der Grund hierfür ist, dass der Berechnung der jeweiligen Merkmale eine Mindestanzahl von zwei bzw. drei Positionsdatenpunkten zu Grunde liegt. Entsprechend können diese also erst nach dem Erreichen dieser Anzahl berechnet werden. Die hierdurch entstehenden Lücken in den einzelnen Sequenzen werden später, im Zuge der Vorbereitung der Daten für die Klassifikation, behandelt.

Ein weiterer Umstand, unter welchem NULL-Werte bei der Berechnung von Bewegungsmerkmalen auftreten können, ist gegeben, wenn der `timediff`-Wert zwischen zwei aufeinanderfolgenden Datenpunkten `0.0` beträgt, was in der Regel eine Folge von Dopplungen ist. Da in solch einem Fall kein `velocity`-Wert berechnet werden kann, wird *Forward-Filling* eingesetzt. Das heißt der fehlende `velocity`-Wert wird einfach auf den letzten `velocity`-Wert gesetzt. Analog wird auch mit nicht-berechenbaren `heading`-Werten verfahren, welche ebenfalls eine Folge von Dopplungen sind. Denn aus zwei identischen Koordinaten kann keine Richtung bestimmt werden. Der große Vorteil dieser direkten Behandlung von nicht-berechenbaren `velocity`- und `heading`-Werten mittels Forward-Filling ist, dass die übrigen Merkmale auf Basis der aufgefüllten Werte dennoch berechnet werden können, wobei eine gewisse Fehlerbehaftung jedoch nicht auszuschließen ist.

7 Umsetzung des Map-Matchings

7.1 Einrichtung der Valhalla-Instanz

Um die Valhalla Map-Matching-API nutzen zu können, muss Valhalla zunächst installiert werden. Für die Installation existieren hierbei zwei Optionen. Die erste ist, den zugehörigen Quellcode herunterzuladen und über das Toolkit *CMake* einen *Build* von Valhalla zu erstellen. Die zweite und von den Valhalla-Entwicklern empfohlene besteht darin, eine Valhalla-Instanz über ein bereitgestelltes *Image* in einem *Docker-Container* laufen zu lassen [19]. Auch Hagen und Saki [8] empfehlen diese Option, da sie nicht nur weniger komplex, sondern auch effizienter im Hinblick auf Zeit und Ressourcen ist.

Entsprechend wurde für die Installation von Valhalla im Zuge dieser Arbeit die zweite Option genutzt. Verwendet wurde die Version 4.19.0 von *Docker-Desktop* für *Microsoft Windows* und das Docker-Image von *GIS-OPS*, welches laut Hagen und Saki [8] dem Original-Image von Valhalla vorzuziehen ist und bequem über *Git* [32] bezogen werden kann. Das Image wurde so konfiguriert, dass die Valhalla-Instanz beim ersten Start mit *OpenStreetMap*-Daten der Regionen Oberfranken und Thüringen aufgebaut wurde. Bezogen wurden diese Daten vom hierfür bereitgestellten Download-Server der *Geofabrik GmbH* [33], einem Partner des OpenStreetMap-Projekts. Der initiale Build hat hierbei auf einem System mit 32 Gigabyte RAM und einer CPU mit sechs Kernen (zwölf Threads) ca. 25-30 Minuten in Anspruch genommen. Einmal aufgebaut startet die Valhalla Instanz bei Bedarf jedoch in wenigen Sekunden. Anschließend ist die Map-Matching-API unter `http://localhost:8002/trace_route` erreichbar. Für mehr Informationen zur lokalen Installation und Konfiguration von Valhalla für das Map-Matching, sei an dieser Stelle auf die Anleitung von Pandey [34] verwiesen.

7.2 Vorhersage des Matching-Modus / Vorklassifikation

7.2.1 Vorstellung des Klassifikationsansatz

Wie bereits erwähnt, handelt es sich bei der Vorhersage des Matching-Modus um eine Vereinfachung des dieser Arbeit zugrundeliegenden Klassifikationsproblems. Die Vereinfachung ergibt sich insbesondere daraus, dass der Map-Matching-Service von Valhalla keine Unterscheidung von motorisierten Straßenfahrzeugen voraussetzt, da diese unter dem Modus *auto* zusammengefasst werden. Hierdurch ergibt sich für die Vorklassifikation die Möglichkeit,

einen einfacheren Klassifikationsansatz zu verwenden, welcher keine sequenzielle Betrachtung der Daten durch ein RNN voraussetzt. Stattdessen soll die Vorklassifikation auf Basis ausgewählter deskriptiver Statistiken erfolgen, die der Zusammenfassung der verschiedenen Werte der Bewegungsmerkmale einer Sequenz dienen. Dass ein solcher Ansatz vielversprechend ist, zeigt beispielhaft Abb. 21. Jeder Punkt in den dargestellten Graphen entspricht einer einzelnen Sequenz mit einer Länge von einer Minute und einem Aufnahmeintervall von einer Sekunde. Die Einzeichnung in die Graphen beruht allein auf der Lage und Streuung der Geschwindigkeitswerte. Das gewählte Streuungsmaß ist in beiden Fällen die Standardabweichung, welche links in Abhängigkeit vom 95%-Quantil und rechts abhängig vom arithmetischen Mittel dargestellt wird. In beiden Graphen ist eine deutliche Clusterbildung erkennbar, wobei die sichtbaren Cluster die vorherzusagenden Matching-Modi bereits gut – aber sicherlich noch nicht optimal – abbilden.

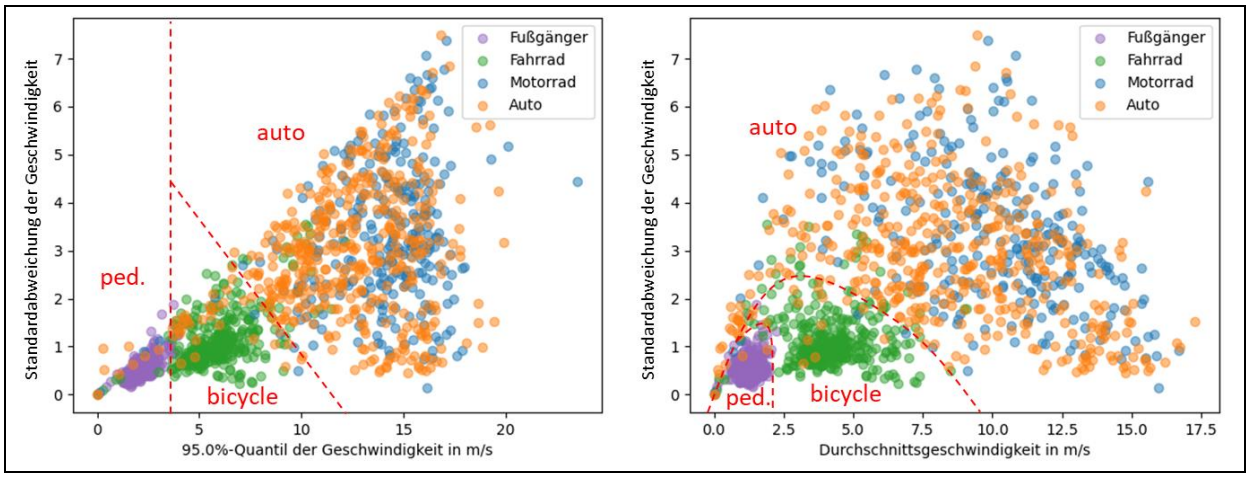

Abb. 21: Clusterbildung bei der Darstellung der Geschwindigkeitsstreuung in Abhängigkeit von der Durchschnittsgeschwindigkeit der einminütigen Sequenzen

7.2.2 Erzeugung der Trainings- und Testdaten

Der nicht-sequenzielle Klassifikationsansatz bei der Vorhersage des Matching-Modus macht eine weitere Vorverarbeitung der Datensätze für das Training und den Test der Vorklassifikatoren notwendig. Für diese Vorverarbeitung wurde eine Funktion umgesetzt, die aus einem beliebigen um Bewegungsmerkmale erweiterten Datensatz, wie in Kapitel 6 beschrieben, einen Datensatz für die Vorklassifikation generiert. Innerhalb dieser Funktion werden zunächst die verbleibenden NULL-Werte am Beginn der im Ursprungsdatensatz enthaltenen Sequenzen behandelt. Wie in Code 2 zu sehen, werden hierfür erst die NULL-Werte der Spalten `acceleration`, `deceleration` und `angular_speed` auf den Wert

`0.0` gesetzt. Darauffolgend werden die fehlenden `velocity`- und `heading`-Werte mittels *Backward-Filling* mit den ersten berechenbaren Werten der zugehörigen Spalten aufgefüllt. Schließlich wird über alle Sequenzen iteriert, um hieraus die Datenpunkte für die Vorklassifikation zu generieren.

```
dataset_df = dataset_df.fillna({"acceleration": 0.0,
  "deceleration": 0.0, "angular_speed":0.0}).fillna(
  method="bfill")
```

Code 2: Behandlung der verbleibenden NULL-Werte in den Datensätzen

Auch wenn die Vorklassifikation nicht sequenziell umgesetzt werden soll, so spielt es doch eine Rolle, dass die hierfür zugrundeliegenden Daten in der späteren Systemumgebung voraussichtlich nur sequenziell (durch die Verkehrsteilnehmer) bereitgestellt werden. Dieser Umstand muss bei der Generierung der Datensätze beachtet werden, denn ein Modell, welches lediglich auf Basis von Datenpunkten trainiert wurde, die die gegebenen mehrminütigen Sequenzen zusammenfassen, könnte in der Realität Schwierigkeiten bei der Klassifikation neuer und ggf. sehr kurzer Sequenzen haben. Um diesem Problem vorzubeugen, werden die Sequenzen im weiteren Ablauf in kleinere Sequenzpakete von jeweils zehn Sekunden (zehn bzw. fünf Datenpunkte je nach Aufnahmeintervall) zerlegt und anschließend schrittweise wieder zusammengefügt, wodurch aus einer Sequenz mehrere verschiedenlange Teilsequenzen entstehen. Die erste Teilsequenz besteht dabei lediglich aus den ersten zehn Datenpunkte der Sequenz, wohingegen die letzte wieder die gesamte Sequenz abbildet. Dieses Vorgehen bringt zwei Vorteile mit sich. Der erste und entscheidende liegt darin, dass die Modelle später auch auf Basis kurzer Sequenzen trainiert werden. Der zweite Vorteil liegt in der hierdurch deutlich erhöhten Menge an Datenpunkten im Zieldatensatz, da aus einer Sequenz nicht nur ein, sondern mehrere Datenpunkte erzeugt werden. Zu beachten ist dabei, dass Datenpunkte, welche auf Basis verschiedener Teilsequenzen einer Sequenz berechnet wurden, im Allgemeinen zwar ebenso verschieden, aber natürlich nicht unabhängig voneinander sind.

Ein einzelner Datenpunkt enthält die in Tab. 4 aufgeführten deskriptiven Merkmale, welche sich zum großen Teil mit Hilfe verschiedener NumPy-Funktionen direkt aus den Teilsequenzen berechnen lassen. Wie zu sehen ist, wird auf die Ermittlung von absoluten Minimal- und Maximalwerten der Merkmale verzichtet, da diese in vielen Fällen durch Ausreißer beeinflusst sind. Eine Annäherung erfolgt jedoch durch die Berechnung der 95%- bzw. 5%-Quantile.

Bezeichner	Datentyp	Erläuterung
`type`	Integer	Label/Klasse der Sequenz: 0 = Fußgänger, 1 = Fahrrad, 2 = Motorrad, 3 = Auto
`elapsed_ms`	Double	Dauer der zusammengefassten Sequenz in Millisekunden
`velo_mean`	Double	Arithmetisches Mittel der Geschwindigkeit
`velo_std`	Double	Standardabweichung der Geschwindigkeit
`velo_perc5`	Double	5%-Quantil der Geschwindigkeit
`velo_perc50`	Double	50%-Quantil/Median der Geschwindigkeit
`velo_perc95`	Double	95%-Quantil der Geschwindigkeit
`acc_std`	Double	Standardabweichung der Beschleunigung
`acc_perc95`	Double	95%-Quantil der Beschleunigung
`dec_std`	Double	Standardabweichung der Verzögerung
`dec_perc95`	Double	95%-Quantil der Verzögerung
`ang_speed_std`	Double	Standardabweichung der Winkelgeschwindigkeit
`ang_speed_perc95`	Double	95%-Quantil der Winkelgeschwindigkeit

Tab. 4: Spalten der Datensätze für die Vorklassifikation

Für die Vorklassifikation werden lediglich zwei, nicht – wie bisher üblich – sechs, Datensätze erzeugt. Der Grund hierfür ist, dass die Länge der zugrundeliegenden Sequenzen durch die nicht-sequenzielle Klassifikation hier eine untergeordnete Rolle spielt. Entsprechend unterscheiden sich die beiden Datensätze lediglich im Aufnahmeintervall, wobei für beide betrachteten Aufnahmeintervalle (eine und zwei Sekunden) zur Generierung der Zieldatensätze die jeweiligen Datensätze mit einer Sequenzlänge von zwei Minuten genutzt werden.

7.2.3 Training, Optimierung und Auswahl der besten Vorklassifikatoren

Auf Basis der beiden erzeugten Datensätze können nun verschiedene Klassifikationsverfahren trainiert, optimiert und hinsichtlich ihrer Klassifikationsqualität miteinander verglichen werden. Wie bereits in Kapitel 5 erwähnt, sollen insbesondere die Verfahren Support Vector Machine und Random Forest gegeneinander abgewogen werden.

Der nachfolgende Code 3 zeigt die notwendige Vorbereitung der Datensätze, bevor diese für das Training Vorklassifikatoren eingesetzt werden können. Entscheidend ist hierbei vor allem

die Ersetzung des `type`-Wertes `3` durch den Wert `2`, wodurch die Klassen *Auto* und *Motorrad* auf eine Klasse, die dem Matching-Modus *auto* entspricht, verallgemeinert werden. Anschließend werden die `type`-Werte, welche die Labels repräsentieren, und die übrigen Merkmale, die die Eingabedaten für die Klassifikation darstellen, auf separate Arrays aufgeteilt. Ausgaben sind hierbei durch den Präfix `y` und Eingaben durch den Präfix `X` gekennzeichnet. Die Vorbereitung der Testdaten für die spätere Evaluierung erfolgt analog.

```
trainset_df.loc[trainset_df["type"] == 3, "type"] = 2
y_train = trainset_df["type"].to_numpy().astype(int)
X_train = trainset_df.drop(["type"], axis=1).to_numpy()
```

Code 3: Datenvorbereitung für das Training der Vorklassifikatoren

Anschließend wird je eine Instanz der scikit-learn-Klassen `SVC` (*Support Vector Classifier*) und `RandomForestClassifier` erzeugt, die das jeweilige Basismodell für die nachfolgende Hyperparameteroptimierung bildet. Im Falle des `RandomForestClassifier` wird hierbei der Parameter `class_weight` auf den Wert `'balanced'` gesetzt, wodurch die Klassen während des Trainings umgekehrt proportional zu ihrer Häufigkeit in den Trainingsdaten gewichtet werden. Hierdurch wird dem unausgeglichene Klassenverhältnis in den Trainingsdaten entgegengewirkt. Außerdem wird der Parameter `criterion` auf den Wert `'entropy'` gesetzt, wodurch die Entropie als Reinheitsmaß beim Aufbau der Decision Trees festgelegt wird. Weitere Parameter werden zunächst nicht gesetzt, allerdings wird in Vorbereitung auf die Hyperparameteroptimierung ein Parametergitter definiert, welches in Code 4 zu sehen ist. Dieses legt die zu optimierenden Hyperparameter für den Random Forest fest, wobei explizit angegeben wird, welche Werte im Zuge der Optimierung jeweils getestet werden sollen. Insgesamt lassen sich die definierten Werte zu 648 Hyperparameterkombinationen kombinieren.

```
{'max_depth': [5, 10, 20, None],
 'max_features': ['sqrt', 'log2'],
 'min_samples_leaf': [2, 4, 8],
 'min_samples_split': [4, 8, 16],
 'n_estimators': [500, 750, 1000],
 'max_samples': [1.0, 0.5, 0.1]}
```

Code 4: Parametergitter für die Optimierung der Random-Forest-Klassifikatoren

Anders als beim `RandomForestClassifier`, werden bei der Instanziierung der Klasse `SVC` keine Parameter gesetzt. Allerdings wird die `SVC`-Instanz in ein Pipeline-Objekt verpackt und so mit einem anführenden `StandardScaler` zusammengefasst, der dafür sorgt, dass die Merkmale der Eingabedaten vor dem Training bzw. vor der Klassifikation zunächst standardisiert werden. Das heißt, die Eingabemerkmale werden so skaliert, dass sie einen Mittelwert von null und eine Standardabweichung von eins besitzen [11, S. 71f.]. Da SVMs empfindlich auf unterschiedliche Wertebereiche der Eingabemerkmale reagieren, können durch eine vorangehende Standardisierung oft deutlich bessere Klassifikationsergebnisse erzielt werden [11, S. 156]. Anschließend wird auch für den Support-Vector-Klassifikator ein Parametergitter zur Hyperparameteroptimierung, das in Code 5 zu sehen ist, festgelegt. Dieses enthält lediglich 75 mögliche Hyperparameterkombinationen. Zu bedenken ist jedoch, dass für eine Betrachtung einer einzelnen Kombination – innerhalb einer `SVC`-Instanz gekapselt – gleich mehrere SVMs trainiert werden müssen. Der Grund hierfür ist, dass sich einzelne SVMs lediglich zur binären Klassifikation eignen. Es existieren jedoch verschiedene Strategien, um auf Basis mehrerer binärer Klassifikatoren, eine mehrklassige Klassifikation umzusetzen. Die Klasse SVC nutzt die *One-versus-One-Strategie*, bei welcher für jedes Klassenpaar jeweils eine SVM zur Unterscheidung der betreffenden Klassen trainiert wird [11, S. 103f.]. Für eine dreiklassige Klassifikation werden intern also drei SVMs benötigt. Bei vier Klassen erhöht sich diese Zahl auf sechs.

```
{'svc__C': [10, 100, 1000, 2500, 5000],
 'svc__gamma': [1, 0.1, 0.01, 0.001, 0.0001],
 'svc__kernel': ['rbf', 'linear', 'sigmoid']}
```

Code 5: Parametergitter für die Optimierung der Support-Vector-Klassifikatoren

Nach der Definition der Basismodelle und der zugehörigen Parametergitter werden die optimalen Hyperparameter für das jeweilige Klassifikationsverfahren gesucht. Hierfür wird, wie in Code 6 zu sehen ist, die scikit-learn-Klasse `GridSearchCV` zur Gittersuche genutzt, welcher bei der Instanziierung ein Basismodell `clf` und das zughörige Parametergitter `param_grid` übergeben werden. Beim Aufruf der `fit`-Methode wird jede mögliche Hyperparameterkombination über eine *Kreuzvalidierung* evaluiert. Hierbei wird der Trainingsdatensatz zunächst in mehrere sogenannte *Folds* aufgeteilt. Danach wird das Modell unter Nutzung der aktuellen Hyperparameterkombination einmal pro Fold trainiert, wobei der betrachtete Fold zur Validierung und die übrigen Daten für das Training genutzt werden [11, 76ff.]. Für jeden Fold wird dabei die Klassifikationsqualität auf Basis des F1-Scores bestimmt. Die Bewertung der

Hyperparameterkombination ergibt sich anschließend, indem die F1-Scores aller Folds gemittelt werden [25].

```
grid = GridSearchCV(clf, param_grid, refit=True, cv=val_split,
  scoring='f1_macro')
grid.fit(X_train, y_train)


best_clf = grid.best_estimator_
```

Code 6: Gittersuche mittels `GridSearchCV`

Das Setzen von `refit` auf `True` bei der Instanziierung von `GridSearchCV` sorgt dafür, dass das beste Modell nach der Suche erneut unter der Nutzung der gesamten Trainingsdaten trainiert wird. Über den Parameter cv kann die Strategie zur Kreuzvalidierung festgelegt werden [25]. Hier wurde das Objekt `val_split` der Klasse `PredifinedSplit` übergeben, welches eine explizite Aufteilung der Trainingsdatenpunkte in fünf Folds festlegt. Der Grund hierfür ist, dass aufeinanderfolgende Datenpunkte in den Datensätzen für die Vorklassifikation in der Regel nicht voneinander unabhängig sind (siehe Unterabschnitt 7.2.2). Würde die Aufteilung also – dem Standard entsprechend – zufällig geschehen, würden sehr wahrscheinlich Abhängigkeiten zwischen Trainings- und Validierungsdaten bestehen, die die Bewertung der Modelle bei der Validierung verzerren. Durch die Festlegung der Folds wird dafür gesorgt, dass voneinander abhängige Datenpunkte demselben Fold zugeordnet werden, wodurch eine Abhängigkeit zwischen den Folds und damit auch zwischen Trainings- und Validierungsdaten vermieden wird.

Nachdem alle Kombinationen kreuzvalidiert wurden, kann auf das Modell mit den besten gefundenen Hyperparametern über das Klassenattribut `best_estimator_` zugegriffen werden [25]. Den nachfolgenden Tabellen Tab. 5 und Tab. 6 können die besten gefundenen Hyperparameter und die zugehörigen F1-Scores für die Support-Vector- bzw. die Random-Forest-Klassifikatoren entnommen werden.

Intervall	C	γ	$K(x, y)$	**F1-Score (Validierung)**
1s	10	0.01	RBF	0,9357
2s	2500	0.01	RBF	0,9391

Tab. 5: Ergebnisse der Gittersuche für die dreiklassige SVM

Intervall	n_{Trees}	s_{max}	d_{max}	$f_{max}(x)$	s_{min}^{leaf}	s_{min}^{split}	**F1-Score (Validierung)**
1s	1000	50%	20	$\sqrt{x}$	2	4	0,9369
2s	500	100%	10	$\sqrt{x}$	2	4	0,9457

Tab. 6: Ergebnisse der Gittersuche für den dreiklassigen Random Forest

Wie zu erkennen ist, erreichen alle optimierten Modelle einen F1-Score von über 0,9350. Es fällt auf, dass die Erhöhung des Aufnahmeintervalls auf zwei Sekunden scheinbar einen positiven Effekt auf beide Klassifikationsverfahren besitzt. Besonders stark profitiert hiervon jedoch der Random Forest, der auf dem entsprechenden Datensatz mit einem F1-Score von 0,9457 die beste Validierungsbewertung unter allen optimierten Modellen erreichen konnte und entsprechend für die Vorhersage des Matching-Modus auf Basis von Sequenzen mit einem Aufnahmeintervall von zwei Sekunden eingesetzt werden sollte. Auch für das Aufnahmeintervall von einer Sekunde erzielt der Random Forest eine etwas bessere Leistung als der konkurrierende Support-Vector-Klassifikator und ist diesem somit vorzuziehen.

7.2.4 Evaluierung auf den Testdaten

Da eine Hyperparameteroptimierung und Modellauswahl grundsätzlich auf Basis der Validierungsbewertung erfolgt, wurden im letzten Unterabschnitt bereits die beiden Random-Forest-Klassifikatoren als endgültige Modelle für die Vorhersage des Matching-Modus festgelegt. Durch die Abhängigkeit zwischen Modellauswahl und Validierungsbewertung eignen sich die zugehörigen F1-Scores jedoch nicht zur Abschätzung der Verallgemeinerungsfähigkeit der Modelle. Deswegen werden die Modelle im Folgenden auf Basis der zugehörigen Testdatensätze evaluiert, wobei besondere Aufmerksamkeit auf die Sensitivität der Modelle gelegt wird.

Für ein Aufnahmeintervall von einer Sekunde ergibt sich dabei die in Abb. 22 dargestellte Konfusionsmatrix, aus welcher mit Hilfe von Formel (2.12) die Sensitivität für die einzelnen Klassen abgeleitet werden kann. Für die Klasse *Fußgänger* (Modus *pedestrian*) erreicht das Modell eine hervorragende Sensitivität von etwa 0,98, das heißt, 98% der Testdatenpunkte dieser Klasse werden korrekt vorhergesagt. Etwas schlechter, aber dennoch gut, schneidet das Modell mit einer Sensitivität von 0,94 bei der Klassifikation von motorisierten Fahrzeugen (Modus *auto*) ab. Die größten Schwierigkeiten bestehen bei der Klasse *Fahrrad* (Modus *bycicle*), für welche das Modell eine relativ geringe Sensitivität von 0,89 aufweist. Der häufigste Fehler ist dabei eine Verwechslung von Fahrrädern und motorisierten Fahrzeugen. So werden ca. 11%

der Fahrräder fälschlicherweise als motorisiertes Fahrzeug klassifiziert. Umgekehrt beträgt die Rate der Fehlklassifikationen 4,8%. Ca. 1,6% der motorisierten Fahrzeuge werden der Klasse *Fußgänger* zugeordnet.

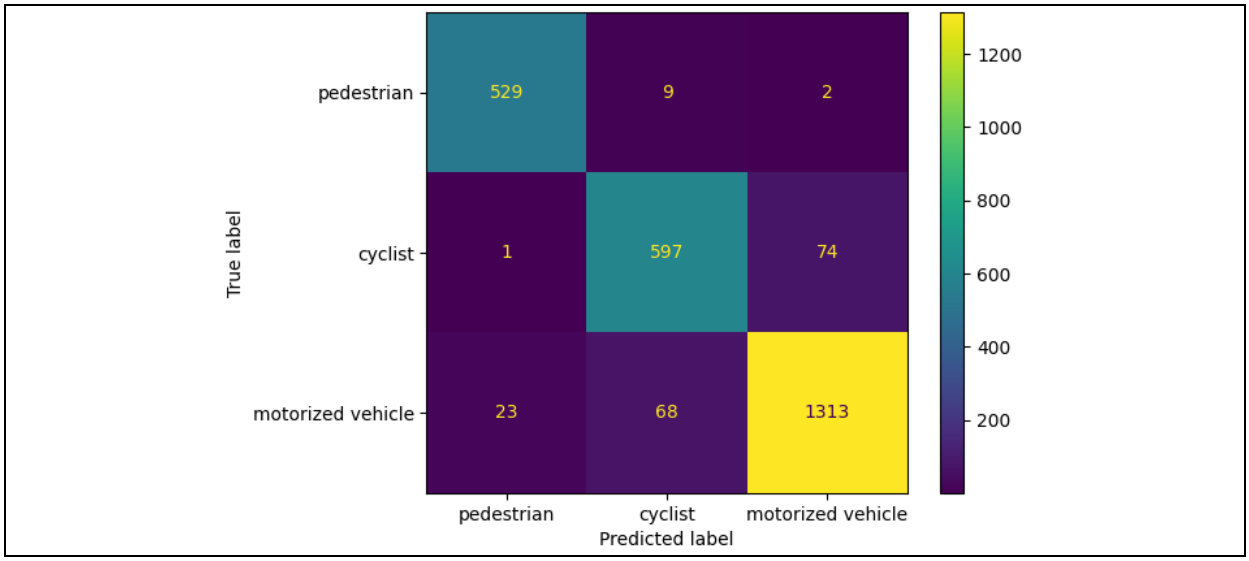

Abb. 22: Konfusionsmatrix für die Vorhersage des Matching-Modus bei einem Aufnahmeintervall von einer Sekunde

Insgesamt erreicht der Random-Forest-Klassifikator für ein Aufnahmeintervall von einer Sekunde auf den Testdaten einen guten F1-Score von 0,9318.

Abb. 23 zeigt die Konfusionsmatrix für ein Aufnahmeintervall von zwei Sekunden. Das zugehörige Modell erreicht für die Klasse Fußgänger eine nahezu perfekte Sensitivität von 0,99. Für motorisierte Fahrzeuge ergibt sich aus der Matrix eine etwas geringere Sensitivität als beim vorher betrachteten Modell. Konkret beträgt diese 0,93. Ebenfalls etwas verschlechtert hat sich die Sensitivität für die Klasse Fahrrad, welche nun lediglich 0,88 beträgt. Die übrigen 12% der Datenpunkte dieser Klasse werden vor allem als motorisierte Fahrzeuge klassifiziert. Umgekehrt werden 5% der Datenpunkte falsch zugeordnet. Dass hier wie auch schon im vorherigen Modell einige Datenpunkte motorisierter Fahrzeuge der Klasse Fußgänger zugeordnet werden (1,8%), lässt sich auf Stau- und Haltesituationen zurückführen.

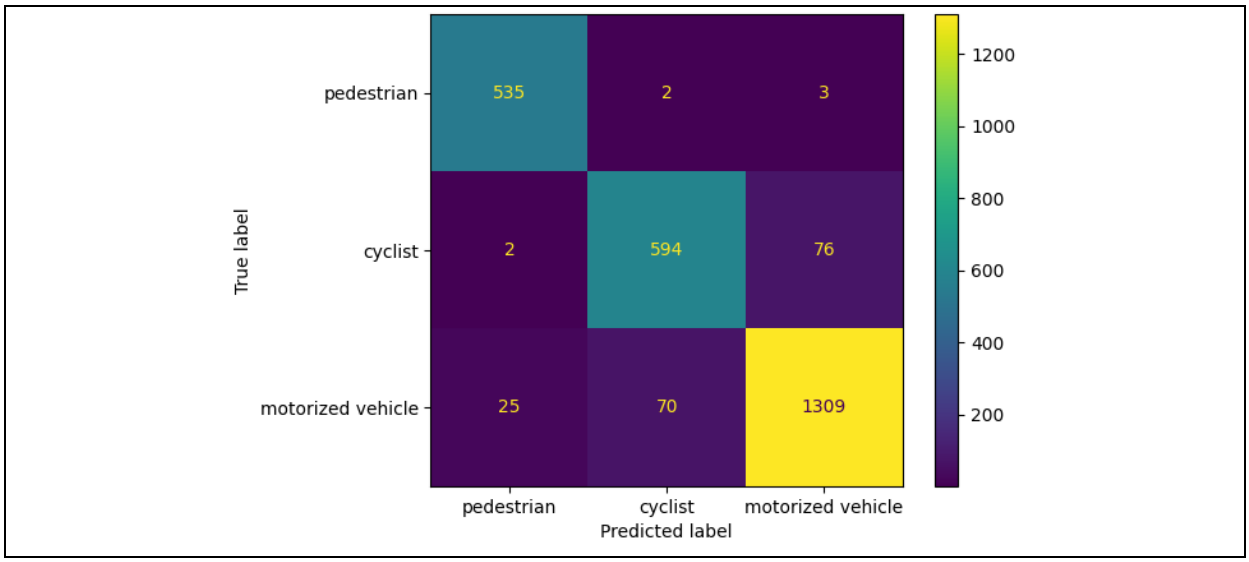

Abb. 23: Konfusionsmatrix für die Vorhersage des Matching-Modus bei einem Aufnahmeintervall von zwei Sekunden

Insgesamt erreicht der Random-Forest-Klassifikator für ein Aufnahmeintervall von zwei Sekunden auf den Testdaten einen guten F1-Score von 0,9322. Damit erzielt er im Zuge der Evaluierung eine minimal bessere Gesamtbewertung als das zuvor betrachtete Modell für ein Aufnahmeintervall von einer Sekunde.

7.3 Map-Matching der Positionssequenzen

Für das Map-Matching der Positionssequenzen wurde die Klasse `MapMatcher` implementiert, welche eine API zum Map-Matching-Endpunkt `trace_route` von Valhalla bereitstellt. Abb. 24 zeigt das zugehörige Klassendiagramm. Bei der Instanziierung der Klasse muss dieser ein String übergeben werden, welcher dem Pfad zum Speicherort des zu nutzenden Modells für die Vorhersage des Matching-Modus entspricht. Anschließend kann über die Methode `match` eine beliebige Positionssequenz, die in Form eins Dataframes mit den in Kapitel 6 beschriebenen Spalten übergeben wird, auf das digitale Straßennetz von OpenStreetMap abgebildet werden, wobei in diesem Zuge auch eine Neuberechnung aller Bewegungsmerkmale erfolgt.

Innerhalb der `match`-Methode wird dabei zunächst über die private Methode `__predict_costing_mode` und unter Nutzung des spezifizierten Klassifikators der Matching-Modus für die übergebene Sequenz bestimmt. Hierfür werden die nötigen deskriptiven Merkmale für die Sequenz berechnet und anschließend an die `predict`-Methode des Klassifikators übergeben. Der durch die Methode zurückgelieferte Integer-Wert wird auf den zugehörigen Matching-Modus abgebildet und anschließend gemeinsam mit der Sequenz an die

Methode `__create_request` übergeben. Diese erstellt hieraus den JSON-Payload für die Anfrage an den Valhalla-Endpunkt. Dieser setzt sich aus einer Liste an Koordinatenpunkten mit zugehörigen Zeitstempeln, dem Matching-Modus und einer Reihe an weiteren dynamisch gesetzten Hyperparametern zusammen. Die Übergabe der Zeitstempel an Valhalla ist hierbei optional, allerdings können durch diese die von Valhalla berechneten Übergangswahrscheinlichkeiten beeinflusst werden, wodurch ein besseres Matching-Resultat erzielt werden kann [8]. Ähnlich verhält es sich auch mit den dynamisch gesetzten Hyperparametern. Hierzu gehören bspw. die Parameter `search_radius` und `gps_accuracy`, die in Abhängigkeit von den `accuracy`-Werten einer Sequenz gesetzt werden und einen direkten Einfluss auf die Auswahl der möglichen Kandidaten beim Matching eines Koordinatenpunktes nehmen. Einer Empfehlung innerhalb der Valhalla-Dokumentation [19] folgend wurde außerdem der Parameter `turn_penalty_factor` in Abhängigkeit von der vorhergesagten Fahrzeugklasse gesetzt, wodurch vertikalem Rauschen in den Daten entgegengewirkt werden kann. Weitere gesetzte Parameter können dem im Anhang A1 zu findenden Beispiel für den Request-Payload entnommen werden.

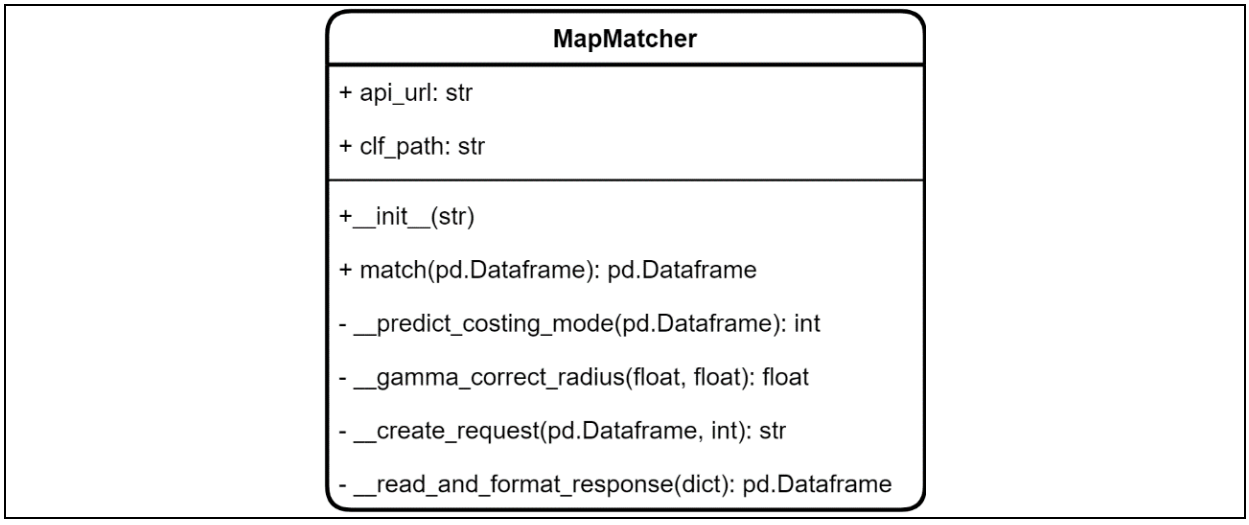

Abb. 24: Klassendiagramm der `MapMatcher`-Klasse

Nach der Erstellung des Payloads erfolgt mit Hilfe der requests-Bibliothek eine POST-Request an den `trace_route`-Endpunkt des Valhalla-Services, welcher die übergebene Sequenz auf das digitale Straßennetz von OpenStreetMap abbildet und anschließend in Form des Response-Payloads – ebenfalls im JSON-Format – zurückliefert. Die angepasste Sequenz wird mit Hilfe der Methode `__read_and_format_response` aus dem `tracepoints`-Attribut des Payloads (siehe Anhang A2) geparst und danach in den Dataframe eingelesen, wodurch die

ursprünglich gemessenen Koordinatenpunkte überschrieben werden. Anschließend werden die Bewegungsmerkmale der Sequenz auf Basis der neuen Koordinatenpunkte erneut berechnet und der Dataframe zurückgeliefert.

Die `MapMatcher`-Klasse wird eingesetzt, um aus allen sechs erzeugten Datensätzen einen weiteren Datensatz auf Basis der gematchten Sequenzen zu erzeugen. Somit existieren also insgesamt zwölf Datensätze zur Untersuchung der verschiedenen Fehlerbereinigungsstrategien im Zuge der nachfolgenden Verkehrsteilnehmerklassifikation.

7.4 Evaluierung des Map-Matchings

Eine objektive Evaluierung des Map-Matchings ist schwierig, da die exakten Strecken, welche die Verkehrsteilnehmer bei der Aufzeichnung der Positionssequenzen zurückgelegt haben, nicht mehr nachvollzogen werden können. Vergleicht man jedoch die gematchten Sequenzen mit den Rohsequenzen, indem man diese wie in Abb. 25 visualisiert, so fällt auf, dass das Map-Matching im Allgemeinen zum erwarteten Ergebnis führt. Während weite Teile der Rohsequenzen klar abseits von den offensichtlich zurückgelegten Verkehrswegen liegen, scheinen die gematchten Sequenzen nahezu perfekt auf diese abgebildet worden zu sein.

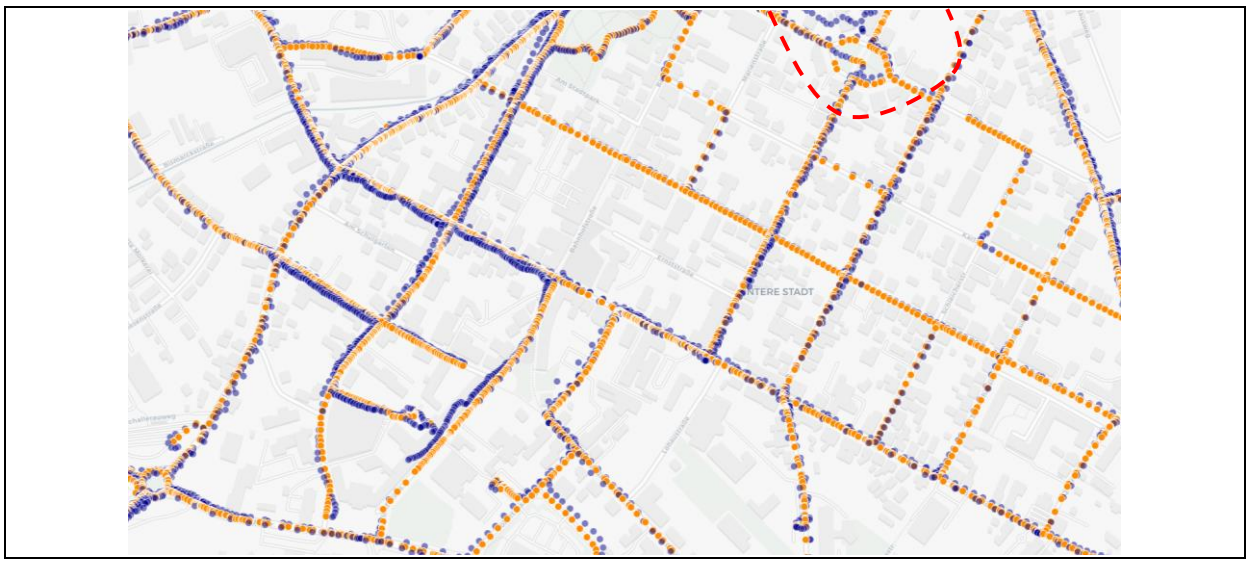

Abb. 25: Auszug aus der Visualisierung der Sequenzen des 2s_4min Trainingsdatensatzes vor (blau) und nach dem Map-Matching (orange)

Nichtsdestotrotz fällt bei genauer Betrachtung auf, dass das Map-Matching bei einigen Sequenzen durchaus auch zu fehlerhaften Resultaten führt. Hierfür konnten insbesondere zwei Ursachen identifiziert werden. Die erste ist die Vorhersage eines falschen Matching-Modus durch

den Vorklassifikator, die zwar nicht zwingend zu einem falschen Map-Matching führt, ein solches aber definitiv begünstigt. Am häufigsten treten Fehler auf Grund eines falschen Matching-Modus bei Fahrradfahrern auf. Werden diese fälschlicherweise für ein motorisiertes Straßenfahrzeug gehalten, kann es bspw. dazu kommen, dass einzelne Koordinaten oder komplette Sequenzen, die eigentlich auf einem Fahrradweg aufgenommen wurden auf eine nahe Straße abgebildet werden. In der Visualisierung ist dies bspw. im markierten Gebiet innerhalb des Kreisverkehrs oben rechts zu erkennen. Hierbei sind einzelne Koordinaten der Sequenz betroffen, was zu großen räumlichen Lücken in der gematchten Sequenz und somit in Folge zu einer Ableitung falscher Bewegungsdaten führt.

Die zweite beobachtbare Fehlerursache liegt in der Bewegung von Verkehrsteilnehmern abseits verzeichneter Verkehrswege. Dies führt in der Regel zu einer Abbildung der Sequenz auf den nächstgelegenen verzeichneten Verkehrsweg – vorausgesetzt ein solcher ist im Suchradius vorhanden – und somit zu einer Verzerrung der tatsächlich zurückgelegten Strecke. Ein Beispiel hierfür ist ebenfalls im markierten Gebiet der Visualisierung oberhalb des Kreisverkehrs erkennbar.

Die Fehlerbehaftung der mit Map-Matching vorverarbeiteten Sequenzen, welche sich in Folge dieser beiden Fehlerursachen ergibt, ist nur schwer zu messen. Ebenso ist nicht objektiv abschätzbar, ob das Map-Matching in der Gesamtheit eher realistischere oder unrealistischere Bewegungsdaten begünstigt. Subjektiv ist jedoch der Eindruck entstanden, dass das Map-Matching insbesondere bei den stärker verrauschten Sequenzen einen positiven Effekt auf den Realismus der Sequenzen und den daraus abgeleiteten Bewegungsdaten besitzt. Beim Großteil der Sequenzen, welcher nur leicht verrauscht ist, sind nur geringe Unterschiede zwischen den Rohsequenzen und gematchten Sequenzen erkennbar. Die Ausnahme bilden Sequenzen, die eine oder beide der oben beschriebenen Fehlerursachen aufweisen und somit dem Risiko unterliegen, dass das Map-Matching einen negativen Effekt auf den Realismus der Sequenzen besitzt.

8 Umsetzung der Klassifikation

8.1 Naiver Ansatz: Erweiterung der Vorklassifikatoren

Der im Abschnitt 7.2 vorgestellte nicht-sequenzielle Klassifikationsansatz und die hierauf basierenden Vorklassifikatoren erzielten bei der dreiklassigen Vorhersage des Matching-Modus ein gutes Ergebnis. Eine naheliegende Möglichkeit das dieser Arbeit zugrundeliegende vierklassige Problem der Verkehrsteilnehmerklassifikation zu lösen, besteht somit in einer simplen Erweiterung dieses bereits umgesetzten Klassifikationsansatzes.

Hierfür sind lediglich kleine Anpassungen an der Umsetzung notwendig. Entscheidend ist vor allem, dass anders als in Code 3 gegeben, die Verallgemeinerung der `type`-Werte `2` und `3` auf den gemeinsamen Wert `2` entfällt, wodurch die im Anschluss erzeugten Ausgabearrays `y_train` und `y_test` also vier mögliche Labels für die in `X_train` und `X_test` gegebenen Eingabedaten enthalten. Diese Labels entsprechen somit nicht mehr den Matching-Modi von Valhalla, sondern den möglichen Verkehrsteilnehmertypen. Anpassungen an den Basismodellen, den Parametergittern oder der Optimierungsstrategie müssen nicht vorgenommen werden, wodurch das Training und die Hyperparameteroptimierung exakt so, wie in Unterabschnitt 7.2.3 beschrieben, umgesetzt werden kann.

Den nachfolgenden Tabellen Tab. 7 und Tab. 8 können die besten gefundenen Hyperparamter und die zugehörigen F1-Scores für die auf vier Klassen erweiterten Support-Vector- bzw. Random-Forest-Klassifikatoren entnommen werden.

Intervall	C	γ	$K(x, y)$	**F1-Score (Validierung)**
1s	1000	0.01	RBF	0,7723
2s	5000	0.01	RBF	0,7685

Tab. 7: Ergebnisse der Gittersuche für die vierklassige SVM

Intervall	n_{Trees}	s_{max}	d_{max}	$f_{max}(x)$	s_{min}^{leaf}	s_{min}^{split}	**F1-Score (Validierung)**
1s	1000	50%	10	$\sqrt{x}$	2	8	0,7711
2s	750	100%	10	$\sqrt{x}$	2	4	0,7777

Tab. 8: Ergebnisse der Gittersuche für den vierklassigen Random Forest

Leicht zu erkennen ist, dass alle optimierten Modelle auf den Validierungsdaten einen F1-Score von über 0,7710 erreichen. Hier besitzt eine Erhöhung des Aufnahmeintervalls auf zwei Sekunden verglichen mit der Validierung der Vorklassifikation einen deutlich weniger offensichtlich einzuordnenden Effekt auf die beiden betrachteten Klassifikationsverfahren. Dennoch scheint zumindest der Random Forest davon zu profitieren. Dieser erreicht für ein Aufnahmeintervall von zwei Sekunden einen F1-Score von 0,7777 und damit auch die beste Validierungsbewertung aller hier optimierten Modelle.

8.2 Sequenzieller Ansatz: Rekurrente neuronale Netze

8.2.1 Klassifikationsansatz und Basisarchitektur

Anders als die bisher umgesetzten Klassifikationsansätze erfordert die Klassifikation mit Hilfe rekurrenter neuronaler Netze keine Zusammenfassung der Sequenzen auf einen einzelnen Datenpunkt. Die in Kapitel 6 beschriebenen Datensätze und ihre gematchten Pendants können somit ohne größere Vorbereitungen direkt zur Umsetzung der sequenziellen Klassifikation genutzt werden. Entsprechend des bereits erläuterten Konzeptes soll die Klassifikation dabei vor allem auf Basis der von den Positionssequenzen abgeleiteten Bewegungsmerkmale `velocity`, `acceleration`, `deceleration` und `angular_speed` erfolgen. Zusätzlich wird außerdem das Merkmal `timediff` in die Eingabe aufgenommen. So wird den rekurrenten Netzen die Möglichkeit gegeben, selbstständig zu lernen, wie diese mit Dopplungen und variierenden Aufnahmeintervallen umgehen können.

Auf Basis dieser konzeptuellen Überlegungen und den Erkenntnissen, welche durch die Arbeit von Simoncini et al. [15] gewonnen werden konnten, wurde die in Abb. 26 dargestellte Basisarchitektur für die umzusetzenden RNNs entworfen. Zu beachten ist, dass sich die Darstellung zur Vereinfachung lediglich auf einen einzelnen Zeitschritt einer Sequenz beschränkt; also nicht für eine gesamte Sequenz in der Zeit aufgerollt ist.

Der Darstellung entsprechend setzen sich die RNNs aus je einer Schicht für die Ein- und Ausgabe und insgesamt drei Gruppen an versteckten Schichten zusammen. Die Eingabeschicht besteht hierbei in jedem Fall aus genau fünf vollständig verbundenen Neuronen ohne Aktivierungsfunktion, je eines für jedes oben aufgeführte Eingabemerkmal. Die Ausgabeschicht besteht aus genau vier vollständig verbundenen Neuronen, die jeweils eine der vorherzusagenden Klassen repräsentieren. Hierbei kommt für die gesamte Schicht die Softmax-

Aktivierungsfunktion zum Einsatz, wodurch sichergestellt wird, dass alle Ausgaben des Netzes zwischen null und eins liegen und in Summe eins ergeben. Somit kann der Wert eines einzelnen Ausgabeneurons als Wahrscheinlichkeit dafür interpretiert werden, dass eine Sequenz, auf Basis aller Zeitschritte bis t, der durch das jeweilige Neuron repräsentierten Klasse zugehörig ist.

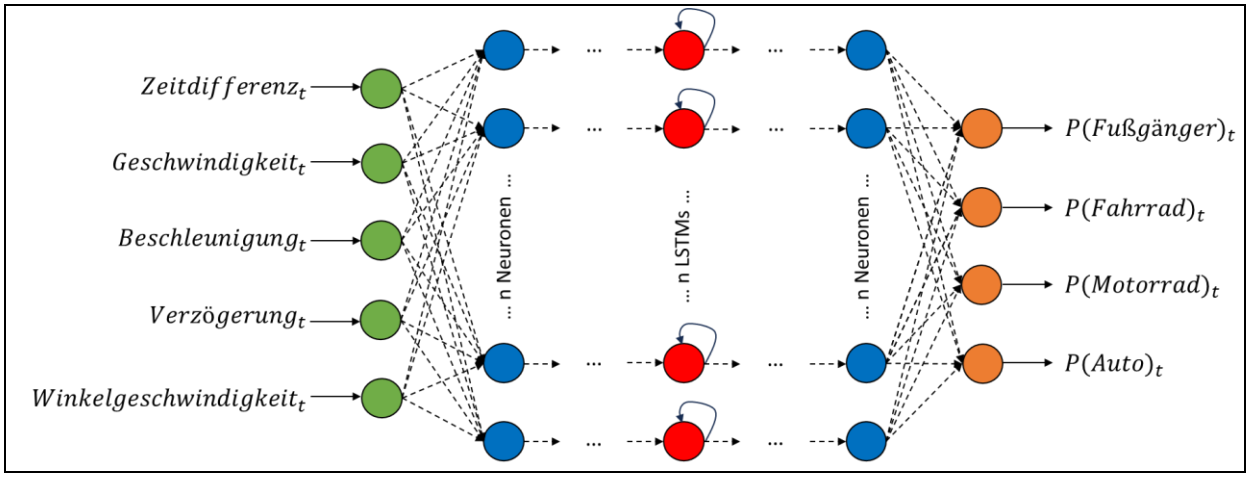

Abb. 26: Basisarchitektur der RNNs für die Verkehrsteilnehmerklassifikation

In Bezug auf die versteckten Schichten lässt die Basisarchitektur viel Raum zur Anpassung der Netze, bspw. durch eine Hyperparameteroptimierung. Nichtsdestotrotz ist eine grobe Struktur der Netze durch die in der Darstellung sichtbare Einteilung der versteckten Schichten in drei Gruppen gegeben. Die erste Gruppe folgt direkt auf die Eingabeschicht der Netze. Sie setzt sich aus einer oder mehreren *vorbereitenden Schichten*, bestehend aus vollständig verbundenen Neuronen, zusammen. Hierauf folgen eine oder mehrere *rekurrente Gedächtnisschichten*, bestehend aus LSTM-Zellen, welche die Netze rekurrent machen und ihnen somit die Fähigkeit verleihen, zeitlich codierte Informationen in den Sequenzen abzubilden. Einfache rekurrente Neuronen ohne interne Zustände sind aufgrund der hohen Sequenzlänge nicht geeignet. Zwischen den Gedächtnisschichten und der Ausgabeschicht befinden sich die *nachbearbeitenden Schichten*, welche sich erneut aus vollständig verbundenen Neuronen zusammensetzen. Die Anzahl der Schichten in den einzelnen Gruppen, die Anzahl der Neuronen pro Schicht und die genutzte Aktivierungsfunktion sind Hyperparameter der Basisarchitektur, welche zur Minderung der Komplexität bei der Hyperparameteroptimierung jedoch bewusst teilweise eingeschränkt wurden. Wie auch der Darstellung zu entnehmen ist, besitzen bspw. alle versteckten Schichten immer dieselbe Anzahl an Neuronen bzw. Zellen.

Die Basisarchitektur wurde durch Implementierung der in Code 7 vorgestellten Funktion `build_rnn_model` mit Hilfe der Keras-API umgesetzt. Die Umsetzung basiert hierbei auf der `models.Sequential`-Klasse, welcher basierend auf den erwähnten Hyperparametern schrittweise alle benötigten Schichten über die Methode `add` hinzugefügt werden. Für die vollständig verbundenen Schichten kommt hierbei, mit Ausnahme der Eingabeschicht, die `layers.Dense`-Klasse zum Einsatz. Für die Gedächtnisschicht wird die Klasse `layers.LSTM` genutzt. Dieser wird keine Aktivierungsfunktion übergeben, da LSTM-Zellen intern bereits Sigmoid-Funktionen nutzen und somit eine garantiert nicht-lineare Ausgabe erzeugen [18]. Nach Hinzufügen der Ausgabeschicht liefert die Funktion das erstellte Modell zurück, welches nun bereit für das Training ist.

```
def build_rnn_model(l_pre, l_post, l_lstm, n, phi):
  model = keras.models.Sequential()
  model.add(keras.layers.InputLayer(input_shape=[None, 5]))

  for layer in range(l_pre):            # pre-memory-layers
    model.add(keras.layers.TimeDistributed(
      keras.layers.Dense(n, activation= phi)))

  for layer in range(l_lstm):           # memory-layers
    model.add(keras.layers.LSTM(n, return_sequences=True))

  for layer in range(l_post):           # post-memory-layers
    model.add(keras.layers.TimeDistributed(
      keras.layers.Dense(n, activation= phi)))

  model.add(keras.layers.TimeDistributed(    # output-layer
    keras.layers.Dense(4, activation="softmax")))
  return model
```

Code 7: Funktion zur dynamischen Erzeugung der RNNs entsprechend der Basisarchitektur in Abhängigkeit eines Hyperparametersatzes

Durch den Einsatz der Wrapper-Klasse `layers.TimeDistributed` beim Hinzufügen der vollständig verbundenen Schichten und das Setzen des Parameters `return_sequences` auf `True` bei der Instanziierung der `layers.LSTM`-Klasse handelt es sich beim erzeugten RNN

um ein Sequence-to-Sequence-Modell. Das heißt um ein Modell, welches nicht eine Ausgabe pro Sequenz, sondern eine Ausgabe für jeden Zeitschritt innerhalb der Sequenzen erzeugt [11, S. 514]. Ein Sequence-to-Vector-Modell wäre für das gegebene Klassifikationsproblem zwar hinreichend, allerdings lohnt es sich laut Géron [11, S. 513f.] oftmals, ein STV-RNN als STS-RNN zu implementieren, da hierdurch der beim Training berechnete Verlust einen Term für die Ausgabe eines jeden einzelnen Zeitschrittes enthält. Hierdurch gibt es deutlich mehr Fehlergradienten, welche durch das RNN propagiert werden können. Diese Propagierung erfolgt dabei nicht nur ausgehend von der letzten Ausgabe des in der Zeit aufgerollten Netzes, sondern auch ausgehend von den Ausgaben aller einzelnen Zeitschritte, was zu einer Beschleunigung und Stabilisierung des Trainings führt.

8.2.2 Vorbereitung der Datensätze

Damit die Datensätze für das Training und den Test der RNNs eingesetzt werden können, müssen diese in Vorbereitung darauf zunächst in ein durch Keras vorgeschriebenes Format umgewandelt werden. Der nachfolgende Code 8 zeigt die nötigen Vorbereitungsschritte beispielhaft anhand der Trainingsdaten. Für die Testdaten erfolgen diese jedoch analog. Wie zu sehen ist, werden zunächst die verbleibenden NULL-Werte am Beginn der Sequenzen behandelt. Hierfür werden erst die NULL-Werte der Merkmale `acceleration`, `deceleration` und `angular_speed` auf den Wert `0.0` gesetzt. Darauffolgend werden die fehlenden `velocity`- und `heading`-Werte mittels Backward-Filling durch den ersten gültigen Wert der jeweiligen Merkmalsspalte aufgefüllt. Direkt im Anschluss werden die Trainingsdaten auf die benötigten Merkmale reduziert und in Ausgabe- (`y`) sowie Eingabedaten (`X`) aufgeteilt. Keras setzt voraus, dass die Eingabedaten beim Training von RNNs als dreidimensionale Arrays der Form [*Anzahl der Sequenzen, Zeitschritte pro Sequenz, Anzahl der Eingabemerkmale*] übergeben werden. Die hierfür benötigte Umwandlung wird durch die Funktion `make_cubic` realisiert.

```
train_data = train_data.fillna({"acceleration": 0.0,
  "deceleration": 0.0, "angular_speed":0.0}).fillna(method="bfill")
  [["type", "timediff", "velocity", "acceleration", "deceleration",
  "angular_speed"]]

y = train_data.groupby("recording_nr")["type"].first().to_numpy()
X = make_cubic(train_data)
```

Code 8: Datenvorbereitung für das Training der RNNs

`X` und `y` werden anschließend, wie in Code 9 zu sehen, über die scikit-learn Funktion `train_test_split` stratifiziert in Trainings- und Validierungsdaten aufgeteilt. Letztere bilden die Grundlage für das Early-Stopping beim Training der RNNs und die Modellauswahl. Danach werden die gesamten Daten mit Hilfe einer Instanz der `StandardScaler`-Klasse standardisiert. Dabei ist zu beachten, dass die Standardisierungsparameter durch den Aufruf der Methode `fit_transform` allein auf Basis der Trainingsdaten in `X_train` bestimmt werden. Validierungs- und Testdaten sollen keinen Einfluss auf die Standardisierung nehmen, weshalb für `X_test` und `X_valid` an die Methode `transform` übergeben werden.

```
X_train, X_valid, y_train, y_valid = train_test_split(X, y,
  test_size=0.2, stratify=y)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train.reshape(-1, X_train.shape[-1]
  )).reshape(X_train.shape)
X_valid = scaler.transform(X_valid.reshape(-1, X_valid.shape[-1]
  )).reshape(X_valid.shape)
X_test = scaler.transform(X_test.reshape(-1, X_test.shape[-1]
  )).reshape(X_test.shape)
```

Code 9: Erzeugung der Validierungsdaten und Skalierung

Die Vorbereitung der Eingabedaten ist an dieser Stelle abgeschlossen. Die Umsetzung der RNNs als Sequence-to-Sequence-Modelle macht jedoch eine weitere Vorbereitung der Trainings- und Validierungsdaten notwendig. Denn während des Trainings und der Validierung benötigt das RNN nicht nur die finale Ausgabe für die Sequenzen, sondern je eine Ausgabe für jeden enthaltenen Zeitschritt. Entsprechend müssen die Ausgabedaten von einem eindimensionalen Array, welches je eine Ausgabe pro Sequenz enthält, auf ein zweidimensionales Array der Form [*Anzahl der Sequenzen*, *Anzahl der Zeitschritte*] erweitert werden. Die einfache Umsetzung dieser Erweiterung zeigt Code 10 am Beispiel der Trainingsdaten.

```
Y_train = np.empty((len(y_train), X_train.shape[1]))

for i in range(len(y_train)):
  Y_train[i, :] = y_train[i]
```

Code 10: Erweiterung der Labels für das Sequence-to-Sequence-Training

Eine Erweiterung der Testdaten ist nicht notwendig, da die Evaluierung der Modelle – ebenso wie deren Einsatz in der Systemumgebung – anders als das Training nach dem Sequence-to-Vector-Prinzip erfolgt.

8.2.3 Hyperparameteroptimierung und Ergebnisse

Für die Hyperparameteroptimierung der RNNs wird erneut eine Gittersuche eingesetzt. Anders als bei den Vorklassifikatoren kann hier jedoch nicht die Klasse `GridSearchCV` der scikit-learn-Bibliothek genutzt werden. Zwar bietet Keras Wrapper-Klassen an, die es erlauben, Keras-Modelle als scikit-learn-Prädiktoren zu behandeln, allerdings wird dabei nicht die Möglichkeit des Sequence-to-Sequence-Trainings unterstützt. Deshalb wurde eine einfache Gittersuche ohne Kreuzvalidierung implementiert, die zur Hyperparameteroptimierung der RNNs genutzt wird.

Basis für die Gittersuche bei allen zu optimierenden Modellen sind die in Code 11 dargestellten Parameterlisten, die im Prinzip dem Parametergitter, wie es bei `GridSearchCV` zum Einsatz kommt, entsprechen. Auf Basis dieser Listen wird die Liste `param_list` erstellt, welche alle 216 möglichen Hyperparameterkombinationen als Tupel enthält.

```
l_pre_list = [4, 2, 1]
l_post_list = [4, 2, 1]
l_lstm_list = [4, 2, 1]
n_list = [256, 128, 64, 32]
phi_list = ["tanh", "relu"]
```

Code 11: Parametergitter für die Optimierung der RNNs

Code 12 zeigt, wie die weitere Gittersuche für ein einzelnes Modell implementiert wurde. Für jede Hyperparameterkombination in `param_list` wird mit Hilfe der in Unterabschnitt 8.2.1 vorgestellten Funktion `build_rnn_model` ein Modell erzeugt. Dieses wird anschließend über die `compile`-Methode des Modells kompiliert. Dabei wird die zu verwendende Verlustfunktion und der Optimierer festgelegt. Zur Berechnung des Verlustes wird hier eine Instanz der Klasse `losses.SparseCategoricalCrossEntropy` übergeben, welche die Kreuzentropie, eine verbreitete Funktion zur Bestimmung des Verlustes bei mehr als zwei vorherzusagenden Klassen, berechnet. *Sparse,* da im vorliegenden Fall die Labels zu jedem Zeitschritt nur einem einzigen Index für die Zielkategorie entsprechen und sich die einzelnen

Zielkategorien gegenseitig ausschließen [11, S. 304]. Zur Optimierung wird die im Vergleich zum herkömmlichen Gradientenabstieg verbesserte *Adam-Optimierung* (*Adaptive Moment Estimation*) durch eine Übergabe einer Instanz der zugehörigen Klasse `optimziers.Adam` eingesetzt. Diese führt unter Umständen zu einer deutlichen Beschleunigung des Trainingsprozesses [11, S. 354ff.]. Die Lernrate wurde hierbei auf dem Standardwert belassen, welcher 1^{-3} beträgt [27].

```
early_stopping_cb = keras.callbacks.EarlyStopping(
  monitor="val_loss", patience=10, restore_best_weights=True)


min_loss = float("inf")
best_params = ()
best_model = None


for (l_pre, l_post, l_lstm, n, phi) in param_list:
  model = build_rnn_model(l_pre, l_post, l_lstm, n, phi)
  model.compile(loss=keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3))
  model.fit(X_train, Y_train, epochs=1000,
    validation_data=(X_valid, Y_valid),
    callbacks=[early_stopping_cb])


  loss = model.evaluate(X_valid, Y_valid)


  if loss < min_loss:
    min_loss = loss
    best_params = (l_pre, l_post, l_lstm, n, phi)
    best_model = model
```

Code 12: Implementierung der Gittersuche für die Optimierung der RNNs

Der Trainingsprozess wird schließlich über einen Aufruf der `fit`-Methode des Modells angestoßen. Hierbei werden die Eingabedaten für das Training `X_train` und die zugehörigen Labels `Y_train` übergeben. Außerdem wird die Anzahl der Epochen, über welche das RNN maximal trainiert werden soll, festgelegt. Der angegebene Wert von 1000 ist dabei bewusst sehr hoch gewählt, da im vorliegenden Fall nicht die spezifizierte Anzahl der Epochen, sondern das Early-Stopping genutzt wird, um zu bestimmen, wann das Training abgebrochen werden sollte. Hierfür werden zusätzlich die Validierungsdaten `X_valid` und -labels `Y_valid` sowie eine

Instanz der Klasse `callbacks.EarlyStopping` übergeben. Letztere sorgt dafür, dass das Training des RNNs abgebrochen wird, sobald der Verlust auf den Validierungsdaten für zehn aufeinanderfolgende Epochen nicht weiter abgenommen hat – das Training also vermutlich in ein Overfitting übergegangen ist. Durch das Setzen des Parameters `restore_best_weights` auf `True` wird hierbei sichergestellt, dass die Modellparameter auf den Stand zurückgesetzt werden, an welchem das Overfitting noch nicht messbar begonnen hat [27].

Nach dem Abbruch des Trainings wird der Verlust des Modells auf den Validierungsdaten über die `evaluate`-Methode des Modells bestimmt. Anschließend wird der Verlust mit dem des bisher besten gefundenen Modells verglichen, woraufhin dieses gegebenenfalls durch das aktuell betrachtete Modell ausgetauscht wird. Nach einer Iteration über alle Hyperparameterkombinationen kann auf das beste gefundene Modell und die zugehörigen Hyperparameter über die Variablen `best_model` respektive `best_params` zugegriffen werden.

Die nachfolgende Tab. 9 zeigt die Ergebnisse der Hyperparameteroptimierung und somit die besten gefundenen RNN-Modelle für alle zu untersuchenden Rohdatensätze (Suffix *_raw*) und deren mit Map-Matching vorverarbeiteten Pendants (Suffix *_mm*).

Datensatz	l_{pre}	l_{lstm}	l_{post}	$n_{Neurons}$	$\varphi(z)$	**Loss (Validierung)**	**Rang**
1s_1min_raw	1	1	2	128	$\tanh(z)$	0,4562	9
1s_1min_mm	4	1	1	64	$\tanh(z)$	0,4568	10
1s_2min_raw	4	1	1	64	$\tanh(z)$	0,4411	6
1s_2min_mm	2	2	2	128	$\tanh(z)$	0,4644	11
1s_4min_raw	4	2	1	128	$\tanh(z)$	0,4136	3
1s_4min_mm	4	1	2	64	$\tanh(z)$	0,4149	4
2s_1min_raw	4	1	2	64	$\tanh(z)$	0,4534	8
2s_1min_mm	4	1	1	64	$\tanh(z)$	0,4652	12
2s_2min_raw	4	1	2	64	$\tanh(z)$	0,4282	5
2s_2min_mm	4	1	1	64	$\tanh(z)$	0,4436	7
2s_4min_raw	2	1	2	128	$\tanh(z)$	0,4037	2
2s_4min_mm	1	1	4	128	$\tanh(z)$	0,3967	1

Tab. 9: Ergebnisse der Gittersuche für die RNNs

In Bezug auf die gefundenen Hyperparamter sind der Tabelle hierbei einige Auffälligkeiten zu entnehmen. Bspw. ist zu erkennen, dass die meisten RNNs eine höhere Anzahl vorbereitender Schichten (l_{pre}) verglichen mit den nachbereitenden Schichten (l_{post}) besitzen. Erstere scheinen im Allgemeinen also relevanter für die Leistung der RNNs zu sein.

Die Anzahl der LSTM-Schichten (l_{lstm}) beträgt bis auf zwei Ausnahmen immer eins. Eine Erhöhung dieser Anzahl führt also scheinbar zu keiner weiteren Verbesserung der Gedächtnisleistung der RNNs. Anders verhält es sich jedoch mit der Anzahl der Neuronen pro Schicht ($\mathrm{n_{Neurons}}$). Hier ist erkennbar, dass längere Sequenzen im Zuge der Optimierung zu tendenziell breiteren RNNs geführt haben. Dabei waren in allen Fällen jedoch höchstens 128 Neuronen ausreichend. Möglicherweise hat eine weitere Verbreiterung der Netze trotz Early-Stopping zu einem Overfitting der Trainingsdaten geführt. Auf der anderen Seite scheinen RNNs mit einer Breite, welche 64 Neuronen unterschreitet, nicht mehr in der Lage zu sein, die für die Klassifikation zugrundeliegenden Muster in den Daten ausreichend abzubilden.

Als eindeutige Empfehlung bei der Wahl der Aktivierungsfunktion hat sich in diesem Fall der Tangens-Hyperbolicus erwiesen, der bei allen betrachteten Modellen zu besseren Ergebnissen als die ReLU-Funktion geführt hat.

Betrachtet man die konkreten Verlustwerte, welcher auf Basis der verschiedenen Datensätze im Rahmen der Validierung erreicht wurden, fällt auf, dass längere Sequenzen in Abhängigkeit vom Aufnahmeintervall in allen Fällen besser klassifiziert werden konnten als kürzere Sequenzen. Insbesondere scheint eine Sequenzlänge von einer Minute zu vergleichsweise schlechten Ergebnissen zu führen. Betrachtet man die übrigen Datensätze mit einer Sequenzlänge von zwei und vier Minuten, ist auffällig, dass die Erhöhung des Aufnahmeintervalls auf zwei Sekunden zu einer sehr deutlichen Verringerung des Validierungsverlustes führt. Der Effekt des Map-Matchings ist anhand dieser Verlustwerte jedoch nur schwer einzuschätzen, da die Bewertungen der korrespondierenden Datensätze oft nah beieinander liegen. Deshalb erfolgt hierzu eine nähere Betrachtung im Zuge der Evaluierung.

9 Evaluierung und Diskussion

Im letzten Kapitel wurden zahlreiche Modelle umgesetzt, welche sich prinzipiell dazu eignen, das dieser Arbeit zugrundeliegende Problem der Verkehrsteilnehmerklassifikation zu lösen. Im Rahmen dieses Kapitels sollen die vielversprechendsten Modelle auf Basis der bisher zurückgehaltenen Testdatensätze beurteilt und miteinander verglichen werden, wobei wie auch schon bei der Evaluierung der Vorklassifikatoren ein besonderes Augenmerk auf die Sensitivität der Modelle gelegt wird.

Als Vertreter des in Abschnitt 8.1 umgesetzten *naiven Klassifikationsansatzes* und als Ausgangspunkt des Leistungsvergleichs wird zunächst der Random Forest evaluiert, welcher auf Basis eines Aufnahmeintervalls von zwei Sekunden optimiert wurde. Abb. 27 zeigt die zugehörige Konfusionsmatrix, die sich aus einer Vorhersage von insgesamt 2616 Testdatenpunkten ergibt. Der Klassifikator erreicht für die Klasse *Fußgänger* eine nahezu perfekte Sensitivität von ca. 0,99. Für die Klasse *Fahrrad* beträgt die Sensitivität jedoch bereits nur noch 0,88. Von den übrigen 12% der Datenpunkte werden knapp 10% und damit der überwiegende Teil fälschlicherweise der Klasse *Auto* zugeordnet. Wie erwartet treten die größten Schwierigkeiten bei der Unterscheidung der Klassen *Motorrad* und *Auto* auf. Erstere erkennt der Klassifikator mit einer Sensitivität von 0,61, Zweitere mit einer Sensitivität von 0,66. Die falsch klassifizierten Datenpunkte werden dabei, wie zu erwarten, vor allem der jeweils anderen Klasse zugeordnet. Trotz der hohen Verwechslungsraten ist der Klassifikator jedoch offensichtlich zu einer gewissen Unterscheidung von motorisierten Straßenfahrzeugen fähig.

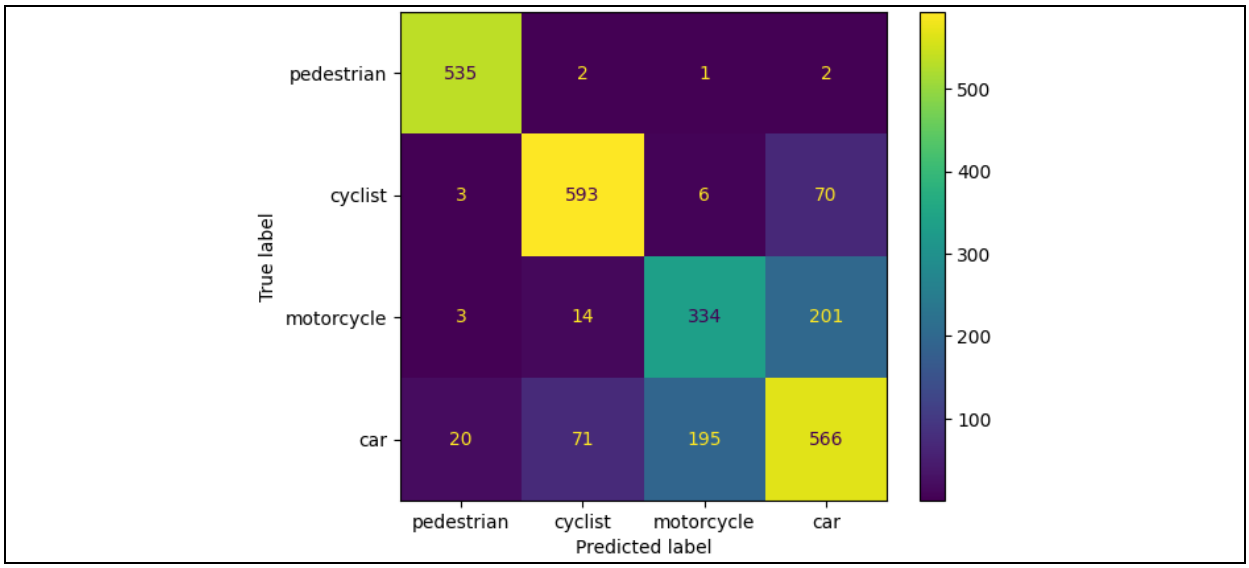

Abb. 27: Konfusionsmatrix des naiven Klassifikationsansatzes für ein Aufnahmeintervall von zwei Sekunden

Insgesamt erreicht der Random-Forest-Klassifikator für ein Aufnahmeintervall von zwei Sekunden als bester Vertreter des naiven Klassifikationsansatzes eine Genauigkeit von 0,7752 und einen F1-Score von 0,7831.

Für die nachfolgende Evaluierung des *sequenziellen Klassifikationsansatzes* erfolgt eine Betrachtung aller RNN-Modelle, welche auf Basis der Datensätze mit einem Aufnahmeintervall von zwei Sekunden umgesetzt wurden. Somit kann nicht nur die Klassifikation selbst, sondern auch die Auswirkungen, welche mit einer Veränderung der Sequenzlänge oder einer Vorverarbeitung mittels Map-Matching einhergehen, genau evaluiert werden. Die Modelle, denen ein Aufnahmeintervall von einer Sekunde zugrunde liegt, werden hier nicht betrachtet, da dies den Rahmen dieses Kapitels sprengen würde und die Validierungsergebnisse, die in Unterabschnitt 8.2.3 vorgestellt wurden, eine weitere Betrachtung zur Beurteilung obsolet machen. Anscheinend wirkt sich eine Erhöhung des Aufnahmeintervalls im Allgemeinen positiv auf die Klassifikationsqualität aus. Die vermutete Ursache hierfür liegt in einer Reduktion des relativen Rauschens (insbesondere horizontal) in den Ausgangsdaten. Da diese Maßnahme jedoch auch mit einem potenziell starkem Informationsverlust bei der Ermittlung von Beschleunigung und Winkelgeschwindigkeit einhergeht, ist es nicht zu empfehlen das Aufnahmeintervall über zwei Sekunden hinaus zu erhöhen.

Abb. 28 stellt die Konfusionsmatrizen für die beiden RNN-Modelle, welche auf den Datensätzen mit einer Sequenzlänge von einer Minute (30 Datenpunkte) trainiert wurden, einander gegenüber. Die Matrix für das Modell, welches auf den Rohsequenzen (*raw*) trainiert wurde, ist hierbei links zu sehen. Die rechte Seite zeigt die analoge Matrix für das Modell, für welches die Sequenzen zuvor mittels Map-Matching (*mm*) vorverarbeitet wurden. In beiden Fällen basieren die abgebildeten Matrizen dabei auf einer Vorhersage von insgesamt 456 Sequenzen aus den zugehörigen Testdatensätzen.

Für die Klasse *Fußgänger* erreichen beide Modelle mit jeweils nur einer Fehlklassifikation eine ebenso hervorragende Sensitivität von 0,99 wie der vorher betrachtete Random-Forest-Klassifikator. Für alle übrigen Klassen zeigt der sequenzielle Ansatz schon hier eine deutliche Verbesserung. Für die Klasse *Fahrrad* wird links eine Sensitivität von 0,92 und rechts eine etwas schlechtere Sensitivität von 0,90 erreicht. Verwechslungen passieren in beiden Fällen zum Teil mit Fußgängern, aber nach wie vor primär mit der Klasse *Auto*. In Bezug auf die Klassen *Motorrad* und *Auto* zeigen die Matrizen ein stark voneinander abweichendes Bild. Während die Sensitivitäten dieser beiden Klassen auf der linken Seite mit 0,65 und 0,71 auf einem moderaten Niveau recht nah beieinander liegen, so unterliegt auf der rechten Seite die Klasse *Motorrad*

mit einer Sensitivität von nur 0,6 ganz klar der Klasse *Auto*, die im Gegenzug jedoch eine gute Sensitivität von 0,8 erreicht. Aus beiden Matrizen lässt sich hierbei ablesen, dass die falsch klassifizierten Sequenzen beider Klassen, wie auch schon zuvor, vor allem der jeweils anderen Klasse zugeordnet werden.

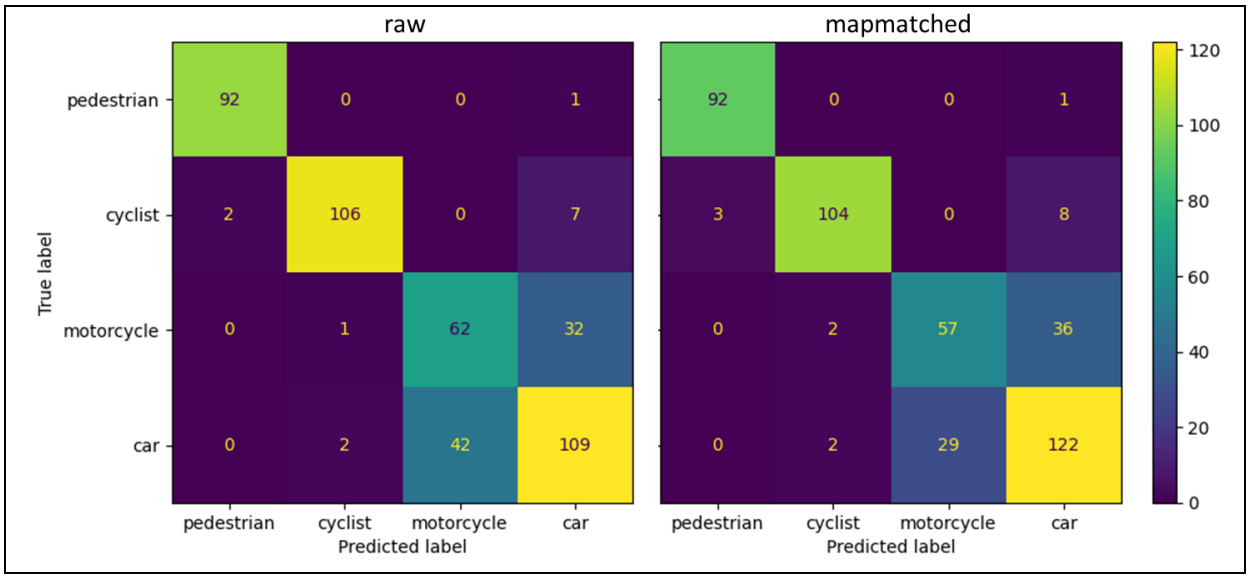

Abb. 28: Konfusionsmatrizen der RNNs für eine Sequenzlänge von einer Minute

Insgesamt erreicht das auf den Rohsequenzen basierende Modell bei einer Sequenzlänge von einer Minute eine Genauigkeit von 0,8092 und einen F1-Score von 0,8188. Erfolgt eine Vorverarbeitung mit Map-Matching erreicht das Modell mit einer Genauigkeit von 0,8224 und einen F1-Score von 0,8259 trotz der geringeren Ausgewogenheit eine etwas bessere Gesamtbewertung. Damit weisen beide Modelle eine deutliche bessere Klassifikationsqualität auf als der Random-Forest-Klassifikator. Der sequenzielle Klassifikationsansatz schlägt somit den naiven Ansatz bereits bei einer Sequenzlänge von einer Minute. Dies verdeutlicht die Stärke von RNNs bei der Klassifikation von sequenziellen Daten.

Als Nächstes erfolgt die Betrachtung der in Abb. 29 in analoger Weise dargestellten Konfusionsmatrizen für die Modelle, welche auf den Datensätze mit einer Sequenzlänge von zwei Minuten (60 Datenpunkte) basieren. Diese stützen sich auf einer Vorhersage von jeweils 218 Testsequenzen. Hier erreichen beide Modelle erstmals eine perfekte Sensitivität von 1,0 bei der Klassifikation von Fußgängern. Für die Klasse *Fahrrad* berechnet sich aus der linken Matrix eine sehr gute Sensitivität von 0,95, welche jedoch nochmal deutlich durch den Sensitivitätswert von 0,98, der sich rechts für diese Klasse ergibt, geschlagen wird. Für die Unterscheidung der Klassen *Motorrad* und *Auto* verzeichnet insbesondere das Modell auf Basis der Rohdaten

eine deutliche Verbesserung. Dieses erreicht für Motorräder eine gute Sensitivität von 0,72 und für Autos eine sehr gute Sensitivität von 0,85. Die Vorverarbeitung mit Map-Matching führt auf der anderen Seite zwar zu einer höheren Sensitivität von 0,83 für Motorräder, dafür jedoch zur bisher schlechtesten verzeichneten Sensitivität für die Klasse *Auto*, welche bei nur 0,59 liegt. Die falsch klassifizierten Sequenzen dieser beiden Klassen werden auch hier in beiden Fällen vor allem der jeweils anderen Klasse zugeordnet. Eine nicht irrelevante Menge der Sequenzen der Klasse *Auto* werden jedoch auch als Fahrräder klassifiziert.

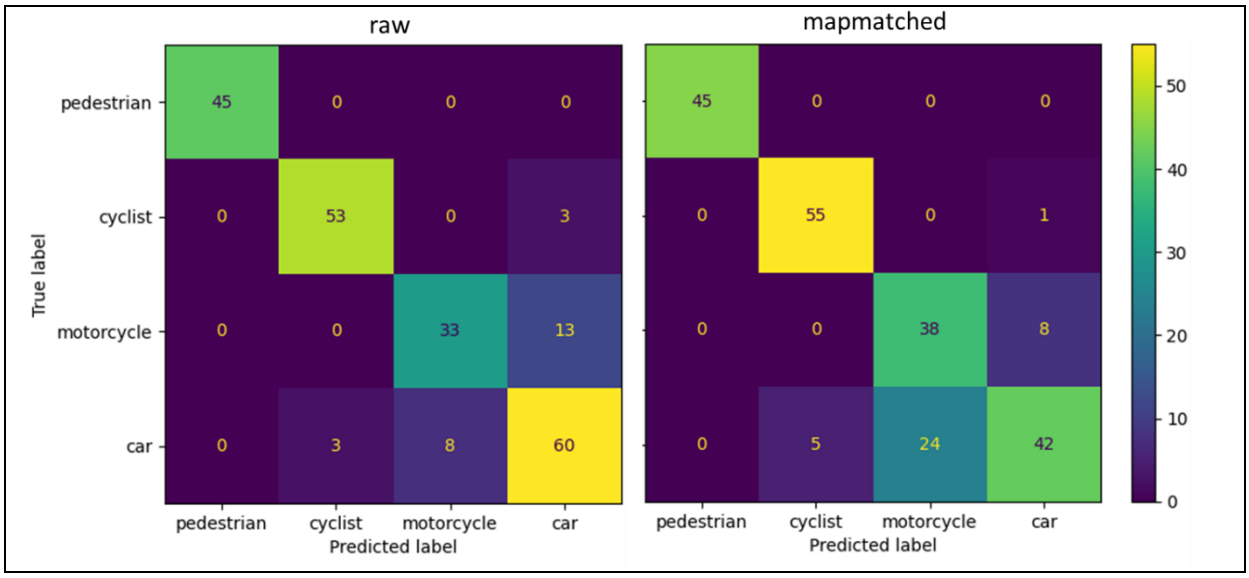

Abb. 29: Konfusionsmatrizen der RNNs für eine Sequenzlänge von zwei Minuten

Insgesamt erreicht das auf den Rohsequenzen basierende Modell bei einer Sequenzlänge von zwei Minuten eine Genauigkeit von 0,8761 und einen F1-Score von 0,8803. Durch die Verdopplung der Sequenzlänge konnte sich die Klassifikationsqualität also nochmal deutlich steigern. Erfolgt eine Vorverarbeitung mit Map-Matching wird – vermutlich auf Grund der schlechten Sensitivität für die Klasse Auto – jedoch nur eine Genauigkeit von 0,8257 und ein F1-Score von 0,8351 erreicht. Damit kann in Bezug auf die Gesamtbewertung zwar trotzdem ein Leistungszuwachs verzeichnet werden, dieser fällt aber deutlich geringer aus als zu erwarten wäre. Eine mögliche Ursache hierfür könnten Zufallseffekte beim Training des RNNs sein, was sich jedoch nur schwer überprüfen lässt.

Abb. 30 zeigt abschließend die Sensitivitätsmatrizen der beiden Modelle, die unter Nutzung der Datensätze mit einer Sequenzlänge von vier Minuten (120 Datenpunkte) trainiert wurden und auf einer Vorhersage von 99 Testsequenzen basieren. Es ist anzumerken, dass die schwindende Größe des Testdatensatzes Zufallseffekte fördert und damit die Aussagekraft der Evaluierung

negativ beeinflusst. Nichtsdestotrotz sollten die Testsequenzen für eine grobe Einordnung der beiden Modelle ausreichend sein.

Die Modelle erzielen mit 0,95 die bisher schlechteste errechnete Sensitivität für die Klasse *Fußgänger*. Diese ist jedoch immer noch als sehr gut einzuordnen, insbesondere im Hinblick darauf, dass sie sich in beiden Fällen aus nur einer einzigen Fehlklassifikation ergibt. Links wird die falsch klassifizierte Sequenz hierbei der nächstnäheren Klasse *Fahrrad* zugeordnet. Auf der rechten Seite erfolgt trotz der hohen Sequenzlänge interessanterweise eine Zuordnung zur Klasse *Auto*, was sich ggf. auf in den Daten enthaltene Stausituationen zurückführen lässt. Für Fahrräder erreichen beide Modelle erstmalig eine perfekte Sensitivität von 1,0. In Bezug auf die Klassen *Motorrad* und *Auto* zeigen beide Modelle im Vergleich mit den anderen betrachteten Modellen eine sehr gute Leistung. Links wird für Motorräder eine gute Sensitivität von 0,71 und für Autos eine hervorragende Sensitivität von 0,90 erzielt. Rechts liegen die beiden Sensitivitätswerte mit 0,81 für die Klasse *Motorrad* und 0,79 für die Klasse *Auto* deutlich näher beieinander, wobei diese auf einem sehr guten Niveau einzuordnen sind.

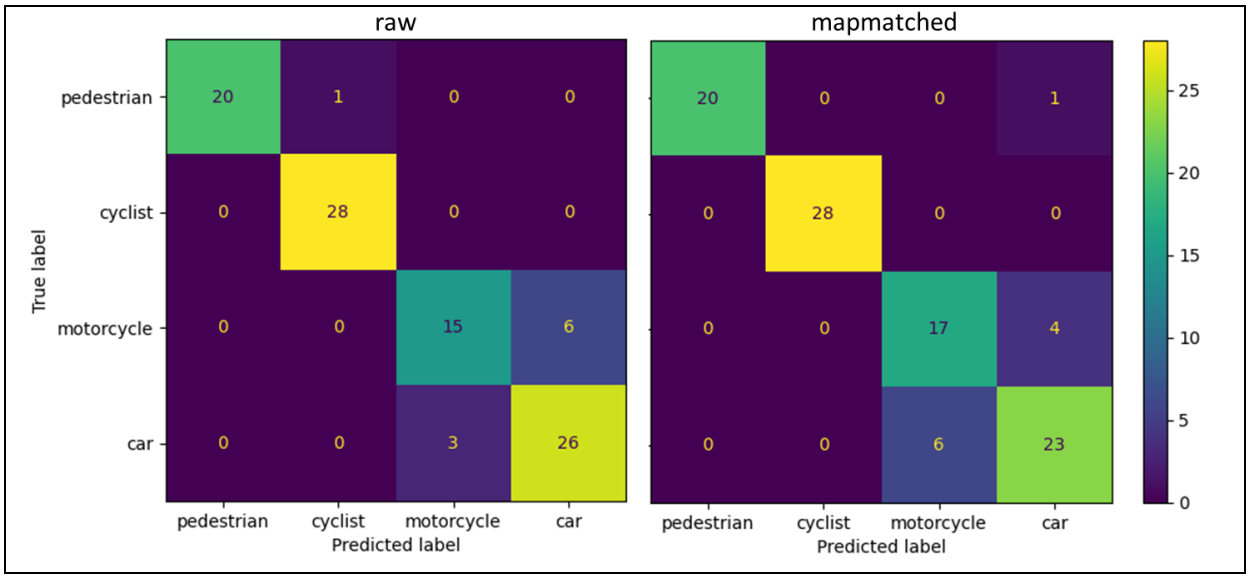

Abb. 30: Konfusionsmatrizen der RNNs für eine Sequenzlänge von vier Minuten

Die Gesamtbewertung für das auf den Rohsequenzen basierende Modell kann sich hierbei bei einer Sequenzlänge von vier Minuten erneut verbessern. Es erreicht eine Genauigkeit von 0,8989 und einen F1-Score von 0,8949. Anders als bei den zuvor betrachteten Modellen mit einer Sequenzlänge von zwei Minuten erzielt hier insbesondere auch das Modell, welches auf Basis der gematchten Sequenzen trainiert wurde, mit einer Genauigkeit und einem F1-Score von jeweils 0,8888 eine deutliche Steigerung in der Klassifikationsqualität. Damit unterliegt

dieses Modell in der Gesamtbewertung zwar knapp, allerdings ist es im Hinblick auf die Ausgewogenheit der Sensitivitäten für die Klassen *Motorrad* und *Auto* je nach Präferenz ggf. dennoch vorzuziehen.

Zusammenfassend lässt sich aus der Betrachtung der verschiedenen Modelle zur Umsetzung des sequenziellen Klassifikationsansatzes eindeutig ableiten, dass die Erhöhung der Sequenzlänge einen stark positiven Einfluss auf die Klassifikationsleistung RNNs besitzt. Dass die Unterschiede zwischen den besten Modellen mit einer Sequenzlänge von zwei und vier Minuten dabei nur recht gering sind, deutet darauf hin, dass eine Erhöhung der Sequenzlänge über vier Minuten keine weiteren erheblichen Verbesserungen verspricht. Insbesondere deswegen, da die Gedächtnisleistung von RNNs bzw. den darin verwendeten Gedächtniszellen eingeschränkt ist und im Falle der hier verwendeten LSTM-Zellen von einer Sequenzlänge von vier Minuten (120 Datenpunkten) gut ausgeschöpft werden sollte. Vier Minuten sind somit als Richtwert für die Sequenzlänge zu betrachten. Die Eigenschaft von RNNs, beliebig lange Sequenzen zu verarbeiten, setzt eine solche Sequenzlänge jedoch nicht zwingend voraus. Dies ist besonders nützlich, da in der Praxis somit die Möglichkeit der Klassifikation von anderen Verkehrsteilnehmern, unabhängig vom Umfang der zur Verfügung stehenden Daten, besteht. Zu beachten ist aber, dass kürzere Sequenzen hierbei mit einer Minderung der Klassifikationsqualität einhergehen, was sich besonders auf die Unterscheidung von motorisierten Fahrzeugen auswirken dürfte.

Die Vorverarbeitung der Sequenzen mittels Map-Matching führte (diskutabel) bei einem bis zwei der drei betrachteten Sequenzlängen zu einer sichtbaren Verbesserung der Klassifikationsqualität. Allerdings erscheint diese Verbesserung insbesondere im Hinblick auf den starken Leistungsabfall bei einer Sequenzlänge von zwei Minuten als sehr inkonsistent und möglicherweise von Zufallseffekten beeinflusst. Die Rohsequenzen erweisen sich trotz der festgestellten Fehlerbelastung als völlig hinreichend, wodurch der Mehraufwand, der mit dem Map-Matching der Sequenzen einhergeht, im Allgemeinen nicht zu rechtfertigen ist. Dies gilt zumindest dann, wenn von einer in der Gesamtheit guten Genauigkeit der zugrundeliegenden GNSS-Sequenzen – wie sie hier gegeben war – ausgegangen werden kann. Ist jedoch mit stärkerem Rauschen in den GNSS-Sequenzen zu rechnen, so soll an dieser Stelle erneut hervorgehoben werden, dass der subjektive Eindruck entstanden ist, dass das Map-Matching auf stark fehlerbelastete Sequenzen einen korrigierenden Einfluss besitzt, der sich positiv auf die Klassifikation der betroffenen Sequenzen auswirken sollte.

10 Zusammenfassung und Ausblick

Das Ziel dieser Bachelorarbeit war die Erforschung der Möglichkeit der Klassifikation von Verkehrsteilnehmern auf Basis von sequenziell bereitgestellten GNSS-Koordinaten durch den Einsatz von maschinellen Lernverfahren. Insbesondere sollte hierdurch die nachfolgende Forschungsfrage geklärt werden:

Welche Verfahren des maschinellen Lernens sind für die Klassifikation von Verkehrsteilnehmern auf Basis von sequenziellen Positionsdaten geeignet?

Besonders wichtig war dabei eine möglichst hohe Realitätsnähe, weshalb speziell für diese Arbeit eine Datenbasis bestehend aus gelabelten GNSS-Sequenzen mit einer Gesamtlänge von 32 Stunden geschaffen wurde. Diese GNSS-Sequenzen wurden durch eine kleine Anzahl freiwilliger Teilnehmer im realen Straßenverkehr aufgezeichnet und umfassten insgesamt vier Klassen: *Fußgänger*, *Fahrrad*, *Motorrad* und *Auto*. Eine genaue Betrachtung der Datenbasis erfolgte in Kapitel 4. Durch vorangegangene Forschungsarbeiten, auf welche im dritten Kapitel Bezug genommen wurde, war bereits bekannt, dass reale Positionsdaten fehlerbelastet sind. Deshalb wurden im Zuge der Betrachtung insbesondere auch die in der Datenbasis identifizierten Fehler aufgezeigt. Da davon auszugehen war, dass diese einen merkbaren Einfluss auf die Klassifikationsqualität besitzen, wurde die obige primäre Forschungsfrage durch die folgende ergänzt:

Wie können reale Positionsdaten, die Ungenauigkeiten und Rauschen aufweisen, so vorverarbeitet werden, dass sie sich gut für den Einsatz maschineller Lernverfahren eignen?

Ausgangspunkt für die Untersuchung beider Fragen war das im fünften Kapitel dieser Arbeit vorgestellte Gesamtkonzept. Wie in Kapitel 6 beschrieben, erfolgte als Ausgangspunkt für die praktische Umsetzung des Konzeptes, eine Generierung von verschiedenen Datensätzen aus der Datenbasis. Diese Datensätze bildeten die Grundlage für Training und Evaluierung der Klassifikatoren sowie für die Untersuchung der Auswirkungen verschiedener Sequenzlängen (eine, zwei oder vier Minuten) und Aufnahmeintervalle (eine oder zwei Sekunden) auf die Klassifikationsqualität. Direkt im Anschluss an die Datensatzgenerierung erfolgte dabei eine Erweiterung der Datensätze, indem aus aufeinanderfolgenden Positionsaufnahmen relevante Bewegungsmerkmale (Geschwindigkeit, Beschleunigung etc.) berechnet und den Sequenzen hinzugefügt wurden.

Kapitel 7 widmete sich der Umsetzung des Map-Matchings, welches eine vielversprechende Möglichkeit zur Behandlung von Verrauschungsfehlern in GNSS-Sequenzen darstellte. Die

Grundlage dafür bildete eine Instanz der Routing-Engine *Valhalla*, welche einen Map-Matching-Service über eine API zur Verfügung stellte. Bei der Übergabe der Sequenzen an diesen Service wurde die Angabe eines Matching-Modus vorausgesetzt, der sich daraus ergibt, ob die übergebene Sequenz von einem Fußgänger (Modus *pedestrian*), Fahrradfahrer (Modus *bicycle*) oder motorisierten Straßenfahrzeug (Modus *auto*) aufgenommen wurde. Um den Matching-Modus vorherzusagen, wurde deshalb eine Vorklassifikation auf Basis zusammenfassender deskriptiver Merkmale der Sequenzen umgesetzt. Als Klassifikatoren wurden dabei Random-Forests genutzt, welche sich in einem vorangegangenen Vergleich mit SVMs im Zuge der Validierung durchsetzen konnten.

Auf Basis der Vorhersage der Matching-Modi durch die Vorklassifikatoren erfolgte für jeden zuvor generierten Datensatz die Generierung eines weiteren Datensatzes durch ein Map-Matching aller enthaltenen Positionssequenzen und eine Neuberechnung der daraus abgeleiteten Bewegungsmerkmale. Die Resultate des Map-Matchings zeigten eine gute Anpassung der Positionssequenzen an das Straßennetz, die dem Anschein nach einen positiven Einfluss auf den Realismus stark verrauschter Sequenzen besaß. Beim Großteil der Sequenzen, waren allerdings nur geringe Unterschiede zwischen den Rohsequenzen und den angepassten Sequenzen erkennbar. In einigen Fällen war das Map-Matching fehlerhaft, insbesondere dann, wenn Matching-Modi falsch vorhergesagt wurden oder sich die Verkehrsteilnehmer bei der Aufzeichnung von Sequenzen abseits des verzeichneten Straßennetzes bewegten.

Kapitel 8 widmete sich der vollständigen Klassifikation der Sequenzen entsprechend der vier betrachteten Verkehrsteilnehmertypen. Hierfür wurden zwei Ansätze umgesetzt. Der *naive Klassifikationsansatz* bestand darin, die bereits umgesetzte nicht-sequenzielle Vorklassifikation um eine Klasse zu erweitern, um somit auch motorisierte Fahrzeuge zu unterscheiden. Nach den hierfür nötigen Anpassungen und dem Training der Modelle erwies sich dieser jedoch schon im Zuge der Validierung als wenig vielversprechend. Deutlich erfolgsversprechender war der sequenzielle Klassifikationsansatz, der auf RNNs basierte. Hierbei handelt es sich um neuronale Netze, die durch Rückkopplungen dazu in der Lage sind, Sequenzen zu verarbeiten und dabei zeitlich codierte Informationen zu beachten. Eine Zusammenfassung der Sequenzen auf deskriptive Merkmale wie bei der Vorklassifikation ist somit nicht nötig.

Zur Umsetzung des sequenziellen Klassifikationsansatzes wurde zunächst eine generalisierte RNN-Architektur ausgearbeitet, vorgestellt und implementiert. Auf Basis dieser Architektur erfolgte anschließend das Training und die Optimierung je eines Modells für jeden Datensatz und dessen zuvor mit Map-Matching vorverarbeitetes Pendant. Bereits bei der Validierung der

Modelle fiel hierbei auf, dass eine Erhöhung des Aufnahmeintervalls auf zwei Sekunden anscheinend zu einer deutlichen Steigerung der Klassifikationsqualität führt, da hierdurch vermutlich das relative Rauschen in den Sequenzen reduziert werden kann. Von einer weiteren Erhöhung ist jedoch abzuraten, da diese ggf. mit einem zu starken Informationsverlust bei der Berechnung der Bewegungsmerkmale einhergeht.

Die Evaluierung der Verkehrsteilnehmerklassifikation und eine Beurteilung der Auswirkungen, die mit verschiedenen Sequenzlängen und einer Vorverarbeitung mittels Map-Matching einhergehen, erfolgte schließlich in Kapitel 9. Hierbei konnte eindeutig gezeigt werden, dass der sequenzielle dem naiven Klassifikationsansatz deutlich überlegen ist. Des Weiteren konnte aus der Evaluation abgeleitet werden, dass eine Erhöhung der Sequenzlänge einen stark positiven Einfluss auf die Klassifikationsleistung der rekurrenten neuronalen Netze besitzt. Gut funktioniert hierbei bereits eine Sequenzlänge von zwei Minuten (60 Datenpunkte). Als Richtwert sollte allerdings eine Sequenzlänge von vier Minuten (120 Datenpunkte) betrachtet werden, denn diese schöpft die Gedächtnisleistung der RNNs besser aus. Da RNNs dazu in der Lage sind, beliebige lang Sequenzen zu verarbeiten, können in der Praxis unter Hinnahme einer ggf. abnehmenden Klassifikationsqualität in Bezug auf motorisierte Fahrzeuge dennoch auch deutlich kürzere Sequenzen verarbeitet werden.

Durch das Map-Matching der Sequenzen konnte im Allgemeinen keine Steigerung der Klassifikationsqualität festgestellt werden. Somit ist der Mehraufwand bei der Vorverarbeitung, der durch ein Map-Matching aller Sequenzen entsteht, in der Praxis nicht zu rechtfertigen. Da jedoch davon ausgegangen werden kann, dass das Map-Matching bei stärker verrauschten Sequenzen dennoch einen korrigierenden Effekt aufweist, sollte auf Basis der Erkenntnisse dieser Arbeit eine Hybridlösung untersucht werden, bei welcher Sequenzen oder Teile von Sequenzen nur dann mit Map-Matching vorverarbeitet werden, wenn ihre geschätzte GNSS-Genauigkeit einen gewissen Schwellenwert überschreitet.

Das insgesamt beste gefundene RNN-Modell konnte im Rahmen der Evaluierung einen F1-Score von 0,8949 erreichen. Dabei erwies sich vor allem die Klassifikation von vulnerablen Verkehrsteilnehmern wie Fußgängern oder Fahrrädern mit einer Sensitivität von 0,95 und 1,00 als äußerst zuverlässig. Aber auch die schwierige Unterscheidung zwischen motorisierten Fahrzeugen, konnte durch das Modell erstaunlich gut bewältigt werden. Für Motorräder ergab sich hierbei eine Sensitivität von 0,71 und für Autos eine Sensitivität von 0,90. Verwechslungen erfolgten vor allem innerhalb der motorisierten Fahrzeugklassen.

Damit erweisen sich RNNs als vielversprechende Möglichkeit zur Klassifikation von Verkehrsteilnehmern auf Basis von sequenziellen Positionsdaten – auch im Rahmen der erweiterten Umfeldwahrnehmung autonomer Fahrzeuge zur Unterstützung bestehender Sensorsysteme. Allerdings sollte der in diese Arbeit realisierte Klassifikationsansatz für einen praxisnahen Einsatz noch deutlich erweitert werden. Aspekte, die hierbei nahezulegen sind, umfassen unter anderem eine Erweiterung der betrachteten Verkehrsteilnehmertypen, aber auch des allgemeinen Umfangs der Datenbasis, welche hier in vielerlei Hinsicht eingeschränkt war. In der Praxis muss insbesondere beachtet werden, dass ein Verkehrsteilnehmer die Art seiner Fortbewegung (und damit seine Klasse) zu jedem Zeitpunkt verändern kann. Dieser Umstand ist hier noch nicht abgebildet worden. Außerdem sollten zusätzliche Kategorisierungen erforscht werden, um darauf basierend weitere Schlussfolgerungen, bspw. in Bezug auf die Relevanz anderer Verkehrsteilnehmer, zu ziehen.

Abschließend sei erwähnt, dass auch eine allgemeine Umsetzbarkeitsprüfung dieses Ansatzes in der weiteren Forschung nicht an letzter Stelle stehen sollte. Infrastrukturell sollten im Hinblick auf die weitläufige Verbreitung von GNSS-Empfängern und den schrittweisen Ausbau des 5G-Netzes zwar bald schon die Möglichkeit zur Realisierung dieses Ansatzes gegeben sein, allerdings ist unklar, inwiefern ggf. andere – bspw. ethische, datenschutzbezogene oder rechtliche – Aspekte dieser Realisierung im Wege stehen könnten.

Quellenverzeichnis

[1] Gartner Inc. „Gartner Says 5G Networks Have a Paramount Role in Autonomous Vehicle Connectivity: CSPs Need to Ensure Participation in Safety Design of Autonomous Vehicles.” https://www.gartner.com/en/newsroom/press-releases/2018-06-21-gartner-says-5g-networks-have-a-paramount-role-in-autonomous-vehicle-connectivity (Zugriff am: 20. Juni 2023).

[2] Innovations-Zentrum Region Kronach e.V. „5GKC: 5G basiertes Testfeld für das automatisierte Fahren.” https://5gkc.net/ (Zugriff am: 20. Juni 2023).

[3] H. Winner, S. Hakuli, F. Lotz und C. Singer, Hg. *Handbuch Fahrerassistenzsysteme: Grundlagen, Komponenten und Systeme für aktive Sicherheit und Komfort*, 3. Aufl. Wiesbaden: Springer Vieweg, 2015. [Online]. Verfügbar unter: https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=1997888

[4] E. D. Kaplan und C. J. Hegarty, *Understanding GPS/GNSS: Principles and applications*. Boston, London: Artech House, 2017.

[5] Wikipedia. „Globales Navigationssatellitensystem.” https://de.wikipedia.org/w/index.php?title=Globales_Navigationssatellitensystem&oldid=233312836 (Zugriff am: 22. Juni 2023).

[6] MagicMaps. „Wie funktioniert Satellitennavigation?” https://www.magicmaps.de/gnss-wissen/wie-funktioniert-gps (Zugriff am: 22. Juni 2023).

[7] N. Zhu, J. Marais, D. Betaille und M. Berbineau, „GNSS Position Integrity in Urban Environments: A Review of Literature,“ *IEEE Trans. Intell. Transport. Syst.*, Jg. 19, Nr. 9, S. 2762–2778, 2018, doi: 10.1109/TITS.2017.2766768.

[8] S. Saki und T. Hagen, „A Practical Guide to an Open-Source Map-Matching Approach for Big GPS Data,“ *SN Computer Science*, Jg. 3, Nr. 5, 2022, Art. Nr. 415, doi: 10.1007/s42979-022-01340-5.

[9] P. Chao, Y. Xu, W. Hua und X. Zhou, „A Survey on Map-Matching Algorithms,“ Okt. 2019. [Online]. Verfügbar unter: https://arxiv.org/pdf/1910.13065

[10] Wikipedia. „Arthur Samuel (computer scientist).” https://en.wikipedia.org/w/index.php?title=Arthur_Samuel_(computer_scientist)&oldid=1149301740 (Zugriff am: 26. Juni 2023).

[11] A. Géron, *Praxiseinstieg Machine Learning mit Scikit-Learn, Keras und TensorFlow: Konzepte, Tools und Techniken für intelligente Systeme*, 2. Aufl. Heidelberg: O'Reilly, 2020.

[12] M. Grandini, E. Bagli und G. Visani, „Metrics for Multi-Class Classification: an Overview," Aug. 2020. [Online]. Verfügbar unter: https://arxiv.org/pdf/2008.05756

[13] A. Mammone, M. Turchi und N. Cristianini, „Support vector machines," *WIREs Computational Statistics*, Jg. 1, Nr. 3, S. 283–289, 2009. doi: 10.1002/wics.49. [Online]. Verfügbar unter: https://wires.onlinelibrary.wiley.com/doi/pdfdirect/10.1002/wics.49?download=true

[14] J. Ali, R. Khan, N. Ahmad und I. Maqsood, „Random Forests and Decision Trees," *IJCSI International Journal of Computer Science Issues*, Jg. 9, Nr. 3, 2012, Art. Nr. 5. [Online]. Verfügbar unter: https://www.uetpeshawar.edu.pk/TRP-G/Dr.Nasir-Ahmad-TRP/Journals/2012/Random%20Forests%20and%20Decision%20Trees.pdf

[15] M. Simoncini, L. Taccari, F. Sambo, L. Bravi, S. Salti und A. Lori, „Vehicle Classification from Low-Frequency GPS Data with Recurrent Neural Networks," *Transportation Research Part C: Emerging Technologies*, Jg. 91, S. 176–191, 2018, doi: 10.1016/j.trc.2018.03.024.

[16] M. Nielsen, „Neural Networks and Deep Learning," 2019. [Online]. Verfügbar unter: https://static.latexstudio.net/article/2018/0912/neuralnetworksanddeeplearning.pdf

[17] Q. Bingfeng. „Understanding gradient descent." https://www.programmersought.com/article/3507388679/ (Zugriff am: 28. Juli 2023).

[18] S. Hochreiter und J. Schmidhuber, „Long Short-Term Memory," *Neural computation*, Jg. 9, Nr. 8, S. 1735–1780, 1997, doi: 10.1162/neco.1997.9.8.1735.

[19] Valhalla. „Map Matching API." https://valhalla.github.io/valhalla/api/map-matching/api-reference (Zugriff am: 4. September 2023).

[20] Python Foundation. „General Python FAQ." https://docs.python.org/3/faq/general.html (Zugriff am: 24. Juni 2023).

[21] Python Foundation. „Python 3.10.12 documentation." https://docs.python.org/3.10/ (Zugriff am: 4. September 2023).

[22] pandas. „pandas documentation." https://pandas.pydata.org/docs/ (Zugriff am: 24. Juni 2023).

[23] NumPy. „NumPy documentation." https://numpy.org/ (Zugriff am: 24. Juni 2023).

[24] J. Hao und T. K. Ho, „Machine Learning Made Easy: A Review of Scikit-learn Package in Python Programming Language," *Journal of Educational and Behavioral Statistics*, Jg. 44, Nr. 3, S. 348–361, 2019, doi: 10.3102/1076998619832248.

[25] scikit-learn. „Machine Learning in Python: scikit-learn 1.3.0 documentation." https://scikit-learn.org/stable/ (Zugriff am: 25. Juni 2023).

[26] TensorFlow. „TensorFlow.” https://www.tensorflow.org/ (Zugriff am: 4. September 2023).

[27] Keras. „API reference.” https://keras.io/api/ (Zugriff am: 29. Juni 2023).

[28] R. F. Torlak, „Detektion der Bewegung von Verkehrsteilnehmern aus Positionsdaten,“ Bachelorarbeit, Hochschule für angewandte Wissenschaften Coburg, 2022.

[29] M. Sohl, „Klassifizierung der Bewegungsmuster von Mobilfunkteilnehmern zur erweiterten Umfeldwahrnehmung autonomer Fahrzeuge,“ Bachelorarbeit, Hochschule für angewandte Wissenschaften Coburg, 2022.

[30] D. Fischer, „Verwendung von Positionsdaten zur automatisierten Klassifizierung von Verkehrsteilnehmern mittels maschinellen Lernverfahren,“ Masterarbeit, Hochschule für angewandte Wissenschaften Coburg, 2023.

[31] Z. Sun und X. Ban, „Vehicle classification using GPS data,“ *Transportation Research Part C: Emerging Technologies*, Jg. 37, S. 102–117, 2013. doi: 10.1016/j.trc.2013.09.015. [Online]. Verfügbar unter: https://www.sciencedirect.com/science/article/pii/S0968090X13002040

[32] GIS-OPS. „Valhalla Docker image by GIS-OPS.” https://github.com/gis-ops/docker-valhalla (Zugriff am: 3. August 2023).

[33] Geofabrik GmbH. „OpenStreetMap Data Extracts.” https://download.geofabrik.de/ (Zugriff am: 3. August 2023).

[34] S. Pandey. „Efficient and fast map matching with Valhalla.” https://ikespand.github.io/posts/meili/ (Zugriff am: 3. August 2023).

Anhang A 1. Beispiel: `trace_route`-Request-Payload

```
{
  "shape": [
    {
      "lat": 50.362309,
      "lon": 11.1765724,
      "time": 1685712673
    },
    ...
  ],
  "trace_options": {
    "turn_penalty_factor": 250,
    "search_radius": 23.376,
    "gps_accuracy": [
      7,
      ...
    ],
    "interpolation_distance": 5
  },
  "costing_options": {
    "auto": {
      "maneuver_penalty": 2
    },
    "bicycle": {
      "cycling_speed": 17.2
    },
    "pedestrian": {
      "walking_speed": 17.2
    }
  },
  "shape_match":"map_snap",
  "costing":"bicycle",
  "format":"osrm"
}
```

Anhang A 2. Beispiel: `trace_route`-Response-Payload


```
{
    "matchings": [
        ...
    ],
    "tracepoints": [
        {
            "matchings_index": 0,
            "waypoint_index": 0,
            "alternatives_count": 0,
            "distance": 0.380,
            "name": "",
            "location": [
                11.117737,
                50.395143
            ]
        },

        ...
        {
            "matchings_index": 1,
            "waypoint_index": 1,
            "alternatives_count": 0,
            "distance": 6.801,
            "name": "",
            "location": [
                11.119408,
                50.393756
            ]
        }
    ],
    "code": "Ok"
}
```