HOCHSCHULE
MITTWEIDA
University of Applied Sciences

MASTERARBEIT

Herr
Lucas Johns, B.Sc.

Entwicklung eines Vorhersagemodells für Zugverspätungen durch maschinelles Lernen und anschließende Integration in eine Blockchainanwendung

Mittweida, Mai 2023

Fakultät **Angewandte Computer- und Biowissenschaften**

MASTERARBEIT

Entwicklung eines Vorhersagemodells für Zugverspätungen durch maschinelles Lernen und anschließende Integration in eine Blockchainanwendung

Autor:
Lucas Johns

Studiengang:
Blockchain & Distributed Ledger Technologies (DLT)

Seminargruppe:
BC20w1-M

Erstprüfer:
Prof. Dr.-Ing. Andreas Ittner

Zweitprüfer:
Erik Neumann, M.Sc.

Einreichung:
Mittweida, 21.05.2023

Verteidigung/Bewertung:
Mittweida, 2023

Faculty of **Applied Computer Sciences and Biosciences**

MASTER THESIS

Development of a prediction model for train delays using machine learning and subsequent integration into a blockchain application

Author:
Lucas Johns

Course of Study:
Blockchain & Distributed Ledger Technologies (DLT)

Seminar Group:
BC20w1-M

First Examiner:
Prof. Dr.-Ing. Andreas Ittner

Second Examiner:
Erik Neumann, M.Sc.

Submission:
Mittweida, 21.05.2023

Defense/Evaluation:
Mittweida, 2023

Bibliografische Beschreibung:

Johns, Lucas:
Entwicklung eines Vorhersagemodells für Zugverspätungen durch maschinelles Lernen und anschließende Integration in eine Blockchainanwendung. – 2023. – 44 S.
Mittweida, Hochschule Mittweida – University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften, Masterarbeit, 2023.

Referat:

This master thesis shows an approach for the prediction of train delays by using Supervised Learning. Therefore, models with different algorithms are tested and compared with each other. Furthermore, it shows how the developed prediction model can be integrated into a blockchain application.

Inhaltsverzeichnis

Inhaltsverzeichnis I

Ergänzende Verzeichnisse III

1 Überblick 1
- 1.1 Einführung . . . 1
- 1.2 Motivation . . . 2
- 1.3 Kapitelübersicht . . . 2

2 Grundlagen 4
- 2.1 Maschinelles Lernen . . . 4
  - 2.1.1 Supervised Learning . . . 4
  - 2.1.2 Overfitting . . . 4
  - 2.1.3 Einordnung in bestehende Prozessmodelle . . . 5
- 2.2 Blockchain als Anwendungsplattform . . . 5
  - 2.2.1 Blockchain . . . 5
  - 2.2.2 Smart Contracts . . . 6
  - 2.2.3 Orakel . . . 6

3 Problemstellung und Anforderungen 7
- 3.1 Beschreibung des Problems . . . 7
- 3.2 Risikoermittelung für Versicherungen . . . 9
- 3.3 Metriken zur Bewertung der Modelle . . . 10
  - 3.3.1 Konfusionsmatrix . . . 10
  - 3.3.2 Receiver Operating Characteristic (ROC) . . . 11
  - 3.3.3 Precision, Recall und F1-Score . . . 12

4 Umsetzung 13
- 4.1 Datenaggregation . . . 13
  - 4.1.1 Generierung der Zugverbindungen . . . 13
  - 4.1.2 Verspätungshistorie . . . 16
  - 4.1.3 Wetterdaten . . . 17
  - 4.1.4 Feiertage . . . 19
  - 4.1.5 Weitere Merkmale . . . 20
- 4.2 Testumgebung, Datenimport und Bereinigung . . . 21
- 4.3 Aufteilung in Trainings- und Evaluierungsdaten . . . 23
- 4.4 Evaluierung der Merkmale . . . 25
- 4.5 Untersuchung verschiedener Algorithmen . . . 27
  - 4.5.1 Lazy Predict . . . 27
  - 4.5.2 Decision Tree . . . 27
  - 4.5.3 Random Forest . . . 30

4.5.4 CatBoost . . . 31
4.5.5 Support Vector Machine . . . 33
4.5.6 Bernoulli Naive Bayes . . . 34
4.6 Ergebnisanalyse und Vergleich mit Baseline-Modell . . . 35
4.7 Deployment des Modells . . . 37
4.8 Integration in einen Smart Contract . . . 39

5 Fazit und Ausblick **43**
5.1 Fazit . . . 43
5.2 Ausblick . . . 43

Anhang **45**

A Quelltexte **45**
A.1 journey_helper.py . . . 45
A.2 journey_builder.py . . . 46
A.3 dataset_builder.py . . . 49
A.4 delay_history.py . . . 53
A.5 weather_helper.py . . . 54
A.6 chainlink-job.toml . . . 56
A.7 Prediction.sol . . . 56

Literaturverzeichnis **59**

Eidesstattliche Erklärung **64**

Ergänzende Verzeichnisse

Abbildungsverzeichnis

4.1 Feature Importance einschließlich historischer Daten, Bildquelle: eigene Darstellung . . . 25
4.2 Feature Importance **ohne** historische Daten, Bildquelle: eigene Darstellung . . 26
4.3 Ausschnitt des konstruierten Entscheidungsbaums, Bildquelle: eigene Darstellung . . . 29
4.4 Konfusionsmatrix des Entscheidungsbaum-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 30
4.5 ROC-Chart des Entscheidungsbaum-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 30
4.6 Konfusionsmatrix des Random-Forest-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 31
4.7 ROC-Chart des Random-Forest-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 31
4.8 Konfusionsmatrix des CatBoost-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 32
4.9 ROC Chart des CatBoost-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 32
4.10 Konfusionsmatrix des SVM-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 33
4.11 ROC Chart des SVM-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 33
4.12 Konfusionsmatrix des Bernoulli Naive Bayes-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 35
4.13 ROC Chart des Bernoulli Naive Bayes-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 35
4.14 Konfusionsmatrix des Baseline-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 36
4.15 ROC Chart des Baseline-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung . . . 36
4.16 Swagger-Interface des Prediction-Service, Bildquelle: eigene Darstellung . . . 39
4.17 Weboberfläche des Chainlink-Nodes, Bildquelle: eigene Darstellung . . . 40
4.18 Komponentenübersicht des Basic Request Models mittels Chainlink (LINK Token nicht dargestellt), Bildquelle: eigene Darstellung . . . 41
4.19 Remix IDE mit Prediction Contract und abgerufener Verspätungswahrscheinlichkeit von 13 % (unten links), Bildquelle: eigene Darstellung . . . 42

Tabellenverzeichnis

3.1 Aufbau einer Konfusionsmatrix . . . 11

4.1 Vorkommen der Bahnhofskategorien mit zugeordneter Gewichtungen . . . 14
4.2 Abfahrtsstunden mit zugeordneter Gewichtungen . . . 15
4.3 Erläuterung der aggregierten Wettermerkmale aus dem ERA5-Land Datensatz 18
4.4 Auflistung aller Merkmale des finalen Datensatzes mit Beschreibung . . . 22
4.5 Ergebnisse der Lazy Predict-Bibliothek, absteigend nach AUC sortiert, über beide Klassen gemittelter F1-Score . . . 28
4.6 Vergleich des F1-Scores der Modelle gegenüber dem Baseline Model, absteigend sortiert . . . 36
4.7 Vergleich des AUC-Scores der Modelle gegenüber dem Baseline Model, absteigend sortiert . . . 36

Quelltextverzeichnis

4.1 Auszug der Ausgabe von journey_helper.py . . . 15
4.2 Skriptausgabe der ersten Version zur Aggregation der Verspätungshistorie . . . 16
4.3 Feature-Aggregation von Feiertagen und langen Wochenenden . . . 19
4.4 Rückgabe der `generate set`-Funktion, welche zum Generieren der Basisfeatures asynchron aufgerufen wird . . . 20
4.5 Bereinigung des Datensatzes und Erstellen des Labels . . . 21
4.6 Implementierung der Aufteilung nach Zugverbindung . . . 24
4.7 Implementierung der Aufteilung nach Reisedatum . . . 24
4.8 Berechnung des Klassengewichts für verspätete Verbindungen im Trainingsdatensatz . . . 25
4.9 Parameter des Entscheidungsbaum-Klassifikators aus der scikit-learn-Bibliothek 29
4.10 Parameter des Random Forest-Klassifikators aus der scikit-learn-Bibliothek . . . 30
4.11 Parameter des CatBoost-Klassifikators . . . 32
4.12 Parameter des SVM-Klassifikators . . . 33
4.13 Request-Funktion des Prediction Contracts (ChainlinkClient-Bibliothek) . . . 40
4.14 Callback-Funktion für die Verspätungswahrscheinlichkeit des Prediction Contracts . . . 41

A.1 Generieren von 1000 Verbindungen mit gewichtetem Sampling . . . 45
A.2 Abfragen von Fahrplandaten für 300000 Verbindungen . . . 46
A.3 Aggregation der Basisfeatures . . . 49
A.4 Aggregation der Verspätungshistorie . . . 53
A.5 Aggregation der Wetterfeatures . . . 54
A.6 Chainlink-Jobspezifikation des Direct Request . . . 56
A.7 Solidity-Quelltext des Prediction-Contracts . . . 56

Abkürzungsverzeichnis

ABI	Application Binary Interface
API	Application Portable Interface
CRISP-DM	Cross Industry Standard Process for Data Mining
DevOps	Development and Operations
EVM	Ethereum Virtual Machine
ICE	Intercity Express
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MLOps	Machine Learning Operations
NetCDF	Network Common Data Format
RBF	Radial Basis Function
RPC	Remote Procedure Call
SVM	Support Vector Machine

1 Überblick

1.1 Einführung

Die Zuverlässigkeit und Pünktlichkeit im öffentlichen Verkehr ist ein wesentlicher Aspekt für die Mobilität von Millionen Menschen. Jeden Tag sind Pendler, Geschäftsreisende, Studenten und Touristen auf den öffentlichen Verkehr angewiesen. Im Jahr 2022 waren im deutschen Fernverkehr nur etwa 75,2% der Züge pünktlich [1]. Das führt einerseits zu Unzufriedenheit bei den Fahrgästen und ist andererseits mit wirtschaftlichen Verlusten verbunden.

Die vorliegende Arbeit beschäftigt sich mit einem Ansatz zur Prognose von Zugverspätungen. Der Rahmen dieses Vorhabens ist durch ein Arbeitspaket im Forschungsprojekt „Community Supported Insurance“ gegeben, in welchem das Blockchain Competence Center Mittweida, ein Institut der Hochschule Mittweida, maßgeblich involviert ist [2]. Ein Bestandteil des Projektes ist die Konzeption einer „Versicherung gegen Zugverspätungen“ auf Basis sogenannter Smart Contracts, das heißt dezentral ausgeführter Programme. Ziel ist es dabei, neue Möglichkeiten hinsichtlich Transparenz, Sicherheit und Beteiligungsmöglichkeiten für den Versicherungssektor zu erforschen. Ein in diesem Rahmen entstehender Prototyp benötigt für die Preiskalkulation der Versicherungsprämien eine Vorhersage, mit welcher Wahrscheinlichkeit eine Zugverbindung eine bestimmte Verspätung aufweisen wird. Dazu ist eine Komponente erforderlich, die eine entsprechende Verspätungsprognose durchführen kann. Im Rahmen dieser Masterarbeit wird die Entwicklung und finale Implementierung dieser Prognosekomponente vorgenommen. Im Kontext der aktuellen Forschung und der wachsenden Bedeutung maschinellen Lernens für praktische Anwendungen bietet die Entwicklung eines solchen Prognosemodells eine interessante Chance, die Möglichkeiten und Grenzen der Technologie für dieses spezielle Problem zu untersuchen. Dazu wird der Einsatz verschiedener Algorithmen des maschinellen Lernens evaluiert, indem Verfahren wie Entscheidungsbäume und Gradienten-basierte Ansätze getestet und gegenübergestellt werden. Die Arbeit umfasst außerdem den Prozess der Datenaggregation, welcher im Hinblick auf das finale Ergebnis eine entscheidende Rolle spielt. Hierfür werden mehrere Datensätze mit Zugverbindungen innerhalb Deutschlands erstellt und diese mit Merkmalen (Features), wie zum Beispiel Wetterinformationen erweitert, um deren Einfluss auf das Verspätungsrisiko zu ermitteln. Untersucht werden soll, wie groß der Mehrwert durch die Integration zusätzlicher Merkmale und der Anwendung des maschinellen Lernens ist und welches Verfahren die besten Ergebnisse liefert. Es werden Erkenntnisse hinsichtlich der Vorhersage, entdeckter Zusammenhänge sowie für weitere zu untersuchende Ansätze gezeigt. Das entwickelte Modell wird über eine geeignete Schnittstelle bereitgestellt. Abschließend wird eine Möglichkeit demonstriert, innerhalb eines Smart Contracts auf das Modell zuzugreifen. Diese Demonstration beinhaltet eine Referenzimplementierung des Modells, den Smart Contract, sowie die notwendige Infrastruktur, um diesen Zugriff zu realisieren.

1.2 Motivation

Die Motivation dieser Arbeit erschließt sich zum einen natürlich durch das beschriebene Forschungsprojekt, in welchem eine solche Prognosemöglichkeit benötigt wird. Insbesondere die Kombination der hoch relevanten Themenfelder KI und Blockchain, deren Anstieg in den letzten Jahren zu verzeichnen war, machen dieses Vorhaben interessant [3]. Ein Smart Contract soll schließlich die Prognose des Modells verarbeiten können, sodass hier diese beiden Technologien aufeinandertreffen. Unabhängig davon ist eine Prognosemöglichkeit für Bahnverspätungen aus einer Reihe weiterer Gründe interessant, welche im Folgenden aufgeführt sind. Verspätungen im Bahnverkehr betreffen eine breite Masse und sorgen immer wieder für Schlagzeilen [4]. Das Thema betrifft die öffentliche Mobilität und hat dadurch eine gesellschaftliche Relevanz. Weiterhin können Verspätungen nicht nur zu Unannehmlichkeiten für Reisende führen, sondern zu gleich negative wirtschaftliche Auswirkungen haben. Die Einsatzbereiche eines solchen Modells sind vielfältig und reichen von Bahnreisenden, die es beispielsweise zur Reiseplanung nutzen können, bis hin zu den Betreibern, welche die Verspätungsprognose zur besseren Planung interner Betriebsabläufe verwenden könnten. Außerdem sind bisher keine Untersuchungen in diese Richtung mit Bezug auf vollständige Zugverbindungen bekannt. Es existieren bereits Forschungsarbeiten, welche sich mit der Entwicklung von Vorhersagemodellen beschäftigen und in welchen auch maschinelles Lernen zum Einsatz kommt [5][6]. Diese sind jedoch auf einzelne Züge beschränkt. Aus Sicht eines Bahnreisenden werden aber in den meisten Fällen nur bestimmte Streckenanteile einer Zuglinie genutzt, vor allem im Regionalverkehr. Eine Verspätung, die ein Zug im letzten Streckenabschnitt seiner Route aufweist, kann daher völlig unerheblich für den Reisenden sein. Dafür können mehrere Züge in einer Verbindung vorkommen und Faktoren, wie z.B. Umstiegszeiten, spielen eine Rolle. Diese Arbeit soll daher als erster Schritt dienen, Verspätungen für eben solche vollständigen Zugverbindungen zu prognostizieren.

1.3 Kapitelübersicht

Im Anschluss an diese Einführung werden in Kapitel 2 zunächst die nötigen Grundlagen zum Verständnis dieser Arbeit erklärt. Dabei geht es einerseits um die Einordnung des Vorhabens in den Bereich des maschinellen Lernens. Weiterhin werden die relevanten Aspekte spezieller Bereiche, insbesondere des Supervised Learning und der Smart Contract-Technologie erklärt. In Kapitel 3 werden allgemeine Anforderungen an das Vorhaben erklärt und die konkrete Aufgabenstellung präzisiert. Dies beinhaltet Aspekte der geplanten Datenaggregation und die Beschreibung des Klassifikationsziels. Insbesondere die Methodik zur Auswertung und Verwendung unterschiedlicher Metriken zum Vergleich der Modelle wird in diesem Kapitel beschrieben. Das Kapitel 4 stellt den Hauptteil der Arbeit dar. Es beschreibt zu Beginn das Verfahren der Datenaggregation und der Generierung der Zugverbindungen. Es wird gezeigt, wie diese Verbindungen mit verschiedenen Merkmalen erweitert werden, um einen geeigneten Datensatz zu erstellen. Anschließend werden die Merkmale hinsichtlich ihrer Aussagekraft

bewertet. In Kapitel 4.5 werden dann verschiedene Modelle auf Basis dieser Merkmale erstellt und evaluiert. Abschließend werden die Resultate verglichen und für das beste Modell die Bereitstellung und Integration in den Smart Contract gezeigt. In Kapitel 5 werden die wichtigsten Erkenntnisse und Ergebnisse festgehalten sowie ein Fazit gezogen und ein Ausblick auf Ansätze gegeben, welche in dieser Arbeit nicht abgedeckt wurden. Die wichtigsten Teile des Quelltextes sind im Anhang dieser Arbeit zu finden und es wird an den entsprechenden Stellen darauf referenziert. Der vollständige Quelltext aller entwickelten Komponenten ist aufgrund des Umfangs separat beigelegt und nicht im Anhang abgebildet.

2 Grundlagen

2.1 Maschinelles Lernen

2.1.1 Supervised Learning

Maschinelles Lernen bezeichnet im Allgemeinen Verfahren zur Generierung von Erkenntnissen aus Erfahrung [7]. Hierbei wird in verschiedene Bereiche unterschieden. Das sogenannte Supervised Learning, welches im Rahmen dieser Arbeit angewendet wird, ist eines der Teilgebiete des maschinellen Lernens. Dabei lernt ein Algorithmus aus einer Menge von Beispielen, die aus einer Eingabe und einer dazugehörigen Ausgabe bestehen [8]. Die Ausgabe ist bereits bekannt und dient dem Algorithmus als Trainingsgrundlage. Ziel des Supervised Learning ist es, eine Funktion zu finden, die eine korrekte Zuordnung von Eingabe zu Ausgabe beschreibt. Dabei kann die Ausgabe kategorisch sein, aber auch kontinuierlich. Ein typisches Beispiel für Supervised Learning ist das Klassifizieren von E-Mails als Spam oder Nicht-Spam. Der Algorithmus lernt anhand von Trainingsdaten mit bereits richtig klassifizierten E-Mails, welche Spam sind und welche nicht. Sobald der Algorithmus aus genügend Daten gelernt hat, kann er automatisch neue E-Mails klassifizieren. Zu den Algorithmen des Supervised Learning gehören beispielsweise Entscheidungsbäume [7]. Weitere Gebiete des maschinellen Lernens sind das Unsupervised Learning und das Reinforcement Learning. Dabei kommen Methoden zum selbständigen Finden von Mustern in unstrukturierten Daten zum Einsatz, beispielsweise das sogenannte „clustering“. Generell wird beim Supervised Learnig der Datensatz, aus welchem gelernt werden soll, in zwei Teile aufgeteilt. Ein Teil wird für das Training verwendet und der Andere dient der Evaluation des Modells. Während in den Trainingsdaten Input und Output vorkommen, wird in den Testdaten der Output durch das Modell vorhergesagt. Anschließend werden die Ergebnisse des Modells mit den echten Outputs der Testdaten verglichen. Durch diesen Vergleich können nun Aussagen über die Leistungsfähigkeit des Modells getroffen werden, indem bestimmte Kennzahlen berechnet werden. Diese werden im folgenden als Metriken bezeichnet. Abhängig davon, ob die Ausgabe des Modells eine Kategorie oder ein numerischer Wert ist, unterscheidet man zwischen einer Klassifikation oder einer Regression [7]. In dieser Arbeit wird durch die Einteilung von anstehenden Zugverbindungen in „pünktlich“ oder „verspätet“ ein Klassifikationsproblem erzeugt. Eine Regression wäre beispielsweise die Vorhersage der Verspätung in Minuten.

2.1.2 Overfitting

Abhängig davon, welcher Algorithmus angewendet wird, kann es vorkommen, dass ein Modell in zu hohem Maße auf die Trainingsdaten angepasst ist. Das bedeutet, dass das Modell zwar sehr gut auf den Trainingsdaten funktioniert, aber bei unbekannten Daten schlechte

Vorhersagen trifft. Es werden Muster aus den Trainingsdaten gelernt, die spezifisch für die Trainingsdaten sind und daher nicht auf neue Fälle übertragen werden können. Zur Vermeidung von Overfitting, werden Verfahren zur Limitierung angewandt wie zum Beispiel das sogenannte Early Stopping [6]. Dadurch kann die Allgemeingültigkeit der erzeugten Modelle gesteigert werden.

2.1.3 Einordnung in bestehende Prozessmodelle

Für die Umsetzung und den Betrieb von Modellen im Bereich des maschinellen Lernens existieren verschiedene Prozessmodelle. Vor allem durch die Verbreitung in der Wirtschaft haben sich bestimmte Verfahren in der Praxis etabliert. Auf zwei bekannte Vertreter wird im Folgenden näher eingegangen, um das Vorhaben dieser Arbeit einzuordnen. Eines der bekanntesten Prozessmodelle ist das CRISP-DM-Modell (Cross-Industry Standard Process for Data Mining) [9]. Es beschreibt sechs Phasen: Verstehen des Problems, Verstehen der Daten, Datenvorbereitung, Modellierung, Evaluation und Bereitstellung. Das Modell ist eher allgemein gehalten, dadurch aber sehr leicht anwendbar und mittlerweile zur gängigen Praxis geworden. Die vorliegende Arbeit orientiert sich an CRISP-DM. In Kapitel 3 ist das vorliegende Problem sowie die Struktur der Daten beschrieben, während in Kapitel 4 schließlich die Umsetzung gemäß der weiteren vier Phasen dargestellt ist.

Eine weitere Vorgehensweise, welche insbesondere auf den kontinuierlichen Betrieb zugeschnitten ist, ist das MLOps-Modell (Machine Learning Operations). Es definiert nicht nur den Prozessablauf sondern auch den Betrieb, welcher auf Automatisierungsmethoden aus der Softwareentwicklung (DevOps) basiert [10]. Durch dieses Zusammenspiel soll eine effiziente und verlässliche Bereitstellung von Modellen gewährleistet werden. Gleichzeitig können Abläufe bei mehrköpfigen Entwicklerteams in Unternehmen besser geregelt werden. Im Rahmen dieser Arbeit werden nur einige Komponenten aus dem MLOps-Bereich verwendet. So wird für die Datenvorbereitung beispielsweise eine sogenannte Pipeline erstellt, welche es ermöglicht, einmalig definierte Verarbeitungsschritte schnell auf neue Daten anzuwenden.

2.2 Blockchain als Anwendungsplattform

2.2.1 Blockchain

Als Blockchain wird eine verteilte Datenstruktur von in Blöcken zusammengefassten Einträgen bezeichnet. Die Blöcke sind geordnet und jeder Block ist kryptografisch mit seinem direkten Vorgänger verknüpft, sodass eine Kette entsteht [11]. Diese wird von mehreren, unabhängigen Netzwerkteilnehmern, den sogenannten Full Nodes gespeichert, welche einem Konsensmechanismus folgen. Dadurch wird eine Art verteilte Buchführung (Distributed Ledger) möglich. Diese verschiedenen Komponenten ergeben im Zusammenspiel das, was heute in der Regel unter dem Begriff „Blockchain“ zu verstehen ist. Neben dem Speichern von

Transaktionen, um beispielsweise dezentrale Währungen zu erschaffen, wie es bei Bitcoin der Fall ist, kann eine Blockchain auch als Plattform für weitere Anwendungsfälle dienen. Die größte öffentliche Blockchain, die im Vergleich zu Bitcoin auch komplexere Operationen erlaubt, ist die Ethereum-Blockchain und die dahinter liegende Smart Contract-Technologie.

2.2.2 Smart Contracts

Als Smart Contract wird eine Sammlung von Anweisungen bezeichnet, die unter bestimmten Bedingungen auszuführen sind. Diese Anweisungen sind meist in Form von Skript-ähnlichen Programmen zusammengefasst. In der Ethereum-Blockchain, welche den Großteil der aktuellen Smart Contract-Infrastruktur ausmacht [12], werden diese Programme in der Programmiersprache Solidity geschrieben und zu Bytecode übersetzt. Dieser wird dann von einer Laufzeitumgebung, der sogenannten Ethereum Virtual Machine (EVM) ausgeführt. Jeder Teilnehmer im Ethereum-Netzwerk führt dann Teile dieses Programms aus, sobald eine Transaktion einen Funktionsaufruf (Call) an diesem Smart Contract beinhaltet. Die Programme werden dadurch nicht von einer zentralen Instanz kontrolliert, sondern „auf der Blockchain“ ausgeführt. Wenn dies auf einer tatsächlich dezentralen und öffentlichen Blockchain, wie dem Ethereum Mainnet geschieht, kann von einer dezentralen Anwendung gesprochen werden. Dies ist zu unterscheiden, da es beispielsweise auch private Blockchains gibt, bei welchen die Dezentralität im Einzelnen zu prüfen wäre.

2.2.3 Orakel

Aufgrund der beschriebenen verteilten Ausführung, können Smart Contracts nicht auf Dinge außerhalb des eigenen Speichers zugreifen. Dadurch ist etwa der Aufruf einer API nicht möglich. Zur Realisierung derartiger Anfragen, werden sogenannte Orakel verwendet [13]. Orakel dienen als Schnittstelle und können Informationen entweder in den Smart Contract schreiben (Inbound) oder aus dem Smart Contract empfangen (Outbound) [14]. Ein Inbound-Orakel lässt sich im einfachsten Fall als einzelner Service beschreiben. Dieser wartet darauf, dass ein Smart Contract an einer Stelle im Programmablauf angekommen ist, an welchem externe Daten, wie beispielsweise Preisinformationen, benötigt werden. Bei EVM-basierten Smart Contracts wird an dieser Stelle ein RPC-Event ausgelöst, welches auch die entsprechende Datenanforderung enthält. Sobald der Orakel-Service dieses Event abfängt, ruft er einen Endpunkt mit der Anforderung auf und schreibt die Antwort über einen Funktionsaufruf zurück in den Smart Contract. Das Problem, dass eine zentrale Entität dabei die Kontrolle über den Smart Contract hat und somit oft die Vorteile der dezentralen Ausführung verloren gehen, wird allgemein als „Orakelproblem“ bezeichnet [13]. Es existieren verschiedene Ansätze, Orakel weniger zentral zu gestalten [15][16], wobei Aussagen von Anbietern etwaiger Lösungen im Einzelnen zu prüfen sind, da Begriffe wie „dezentral“ auch aus Marketinggründen verwendet werden könnten.

3 Problemstellung und Anforderungen

3.1 Beschreibung des Problems

Das Problem besteht darin, für einen Zeithorizont von bis zu 14 Tagen in der Zukunft, eine möglichst korrekte Wahrscheinlichkeit eines definierten Verspätungsereignisses für beliebige Zugverbindungen in Deutschland zu ermitteln. Als Verspätungsereignis wird eine Verspätung am Zielort, also dem letzten Bahnhof der Verbindung, von mehr als 60 Minuten festgelegt. Um eine derartige Prognose durchzuführen, sollen verschiedenen Algorithmen des maschinellen Lernens getestet werden. Speziell soll dabei der Fokus auf Verfahren des Supervised Learning liegen. Die Algorithmen werden mit einem Datensatz trainiert, der aus historischen Zugverbindungen in Deutschland besteht. Daraus ergibt sich die Aufgabe, zunächst einen solchen Datensatz zu erstellen und diesen mit aussagekräftigen Merkmalen zu erweitern. Ein wichtiger Aspekt des Problems ist dabei, dass die Prognose nicht nur für einzelne Züge durchgeführt werden soll, sondern für vollständige Zugreisen, welche oft aus mehreren Teilstrecken bestehen. Der Datensatz muss daher so aufgebaut sein, dass jeder Eintrag eine komplette Reise definiert. Dies führt zu einer deutlich komplexeren Ausgangssituation hinsichtlich der Daten, da viele auf einen speziellen Zug bezogene Informationen, wie Zugnummer, Bahnbetreiber oder Wagenanzahl, nicht homogen für eine Verbindung erhoben werden können. So könnte beispielsweise nur ein Bahnbetreiber an einer Verbindung beteiligt sein oder - bei drei Umstiegen - auch bis zu vier Verschiedene. Es soll daher experimentell geprüft werden, wie und mit welchen Merkmalen der Datensatz sinnvoll erweitert werden kann. Anschließend ist zu evaluieren, wie stark diese einen Einfluss auf die Verspätung haben. Die Prognose ist aufgrund der Datenlage auf reine Zugverbindungen aus dem Regional- und Fernverkehr beschränkt. Andere Transportmittel und insbesondere Schienenersatzverkehr sind davon ausgenommen.

Das Prognosemodell soll später in Echtzeit Ergebnisse liefern, um die Grundlage für eine Versicherungspolice liefern zu können. Daher ist es notwendig, dass die letztlich verwendeten Merkmale schnell und zu jeder beliebigen Zeit aggregiert werden können. Für eine finale Implementierung ist daher neben der Aggregation für die Trainings- und Testdatensätze auch eine Komponente zur Feature Aggregation in Echtzeit zu entwickeln. Diese muss für jede beliebige Zugverbindung alle für das Modell notwendigen Merkmale aggregieren und setzt daher einige Anforderungen zum Beispiel hinsichtlich der Verfügbarkeit der verwendeten Datenquellen voraus. Werden Niederschlagsinformationen im Modell verwendet, so muss auch eine Schnittstelle, die jederzeit eine entsprechende Prognose für eben diesen Niederschlagswert liefert, zur Verfügung stehen. Anfragen für Verbindungen, welche nicht prognostizierbar sind, weil beispielsweise ein Abschnitt mit Schienenersatzverkehr enthalten ist, sind entsprechend zu identifizieren. Es soll eine Softwarekomponente im Rahmen dieser Arbeit entstehen, die diese Aufgaben übernimmt und eine API zur Prognose von Verspätungsrisiken bereitstellt. Weiterhin soll ein Smart Contract geschrieben werden, der in der Lage

ist, für eine noch festzulegende Definition einer Zugverbindung ein Risiko abzufragen und dieses zu verarbeiten. Der Smart Contract muss darüber hinaus keine Funktion erfüllen und soll lediglich als Grundlage weiterer, für das Forschungsprojekt relevanter Schritte dienen. Diese umfassen beispielsweise die Preisberechnung und gehören nicht mehr zum Umfang der vorliegenden Arbeit.

Zur Erstellung möglichst guter Prognosen, soll im Datensatz auch die bisherige Performance der enthaltenen Verbindungen hinsichtlich eventueller Verspätungen enthalten sein. Vorhandene Modelle für einzelne Züge basieren meist auf einer einfachen Auswertung vergangener Fahrten. Diese Auswertungen liefern bereits wichtige Informationen und werden zum Beispiel für Statistiken zur generellen Pünktlichkeit verwendet [17][18]. Hierbei ist zu beachten, dass bei derartigen Angaben nur Direktverbindungen ausgewertet werden. So werden beispielsweise für den ICE 501 von Berlin Gesundbrunnen nach München Hauptbahnhof die Verspätungen an allen Zwischenhalten und am Ziel München Hauptbahnhof erfasst. So kann rückblickend relativ einfach eine Statistik über die Pünktlichkeit des Zuges erstellt werden. Für Aussagen über die Pünktlichkeit einer kompletten Verbindung, die aus mehreren Teilstrecken besteht, müssen zusätzliche Daten ermittelt werden. So ist es zum Beispiel entscheidend ob ein eventueller Umstieg zeitlich möglich war und wie der Bahnreisende bei einem Ausfall eines Zuges alternativ sein Ziel erreichen konnte. Es muss also die tatsächliche und potenziell vom Fahrplan abweichende Verbindung ermittelt werden. Nur so kann letztlich die persönliche Verspätung aus der Perspektive des Bahnreisenden ermittelt werden. Für diese Aufgabe wird eine Schnittstelle von zugfinder.net verwendet [18]. Diese Schnittstelle ermittelt für eine historische Zugverbindung die tatsächliche Verbindung, die der Bahnreisende gefahren ist. Diese soll dann mit der ursprünglichen Verbindung gemäß Fahrplan verglichen werden, um die Verspätung zu ermitteln. Datenabrufe, die im Rahmen dieser Arbeit in Verbindung mit zugfinder.net beschrieben werden, verwenden immer diese eine Schnittstelle. Der Endpunkt ist nicht im Quellcode enthalten, da dieser nicht öffentlich verfügbar ist. Er wurde durch den Betreiber von zugfinder.net für das Forschungsprojekt zur Verfügung gestellt. Generell ist die Datenlage hinsichtlich historischer Zugverspätungen als sehr spärlich zu bewerten. Während in einigen EU-Ländern Schnittstellen seitens der Bahnbetreiber dafür geschaffen wurden, ist dies in Deutschland bei relevanten Bahnunternehmen nicht der Fall. In der Regel sind Soll-Daten, wie Fahrplatinformationen relativ gut verfügbar. So hatte beispielsweise die Deutsche Bahn 2016 eine Fahrplan-API für den Fernverkehr im Rahmen einer Open Data-Initiative veröffentlicht [19]. Zum Zeitpunkt der Umsetzung dieser Arbeit war diese allerdings schon wieder offline. Echtzeitinformationen sind schwieriger zu ermitteln und von Seite der Bahnunternehmen meist nur über die mobilen Apps verfügbar. Am kritischsten hinsichtlich der Aggregation sind aber historische Daten, die von den meisten Bahnunternehmen nicht zur Verfügung gestellt werden. Einige Projekte in diesem Bereich haben die historischen Daten in Eigenarbeit aus Echtzeitdaten gesammelt [20]. Im Rahmen dieser Arbeit wird keine solche Erhebung vorgenommen und für diesen Zweck auf die Schnittstelle von zugfinder.net zurückgegriffen.

Eine weitere wesentliche Eigenschaft des vorliegenden Problems ist die Erzeugung eines binären Klassifikationsproblems. Es ist nicht das Ziel, eine exakte Prognose der Verspätung in Minuten zu ermitteln. Ziel ist eine Prognose darüber, mit welcher Wahrscheinlichkeit eine Verspätung von über 60 Minuten auftritt. Theoretisch ist also nicht nur eine einfache diskrete Klassifikation in „pünktlich“ oder „verspätet“ von Interesse, sondern auch die Stärke der Klassenzugehörigkeit, um schließlich eine Wahrscheinlichkeit zu erhalten. Tatsächlich ändert dies am Verfahren selbst nichts, da in den unterschiedlichen Algorithmen letztlich immer ein numerischer Wert über die Klassenzugehörigkeit entscheidet. Statt der bloßen Zuordnung wird an dieser Stelle einfach der Wert für die Bildung der Wahrscheinlichkeit verwendet. Dies wird auch als probabilistische Klassifikation bezeichnet [8]. Das Verspätungsereignis von 60 Minuten ließe sich theoretisch jederzeit anpassen. Die gezeigten Verfahren und Modelle werden für diese Anforderung optimiert, wobei der gesamte Prozess einschließlich der Datenbasis immer die genauen Verspätungen in Minuten umfasst. Somit ist auch eine allgemeine Gültigkeit der Arbeit, unabhängig von dieser speziellen Anforderung, gewährleistet.

3.2 Risikoermittelung für Versicherungen

Um eine Versicherung ökonomisch sinnvoll zu betreiben, ist die genaue Berechnung eines Risikos, also der Eintrittswahrscheinlichkeit eines Schadens von essentieller Bedeutung [21]. In diesem Fall geht es insbesondere um sogenannte parametrische Versicherungen. Bei dieser Art Versicherungen ist eine Entschädigung nicht von der tatsächlichen Schadenshöhe abhängig, sondern von dem Eintritt vorher festgelegter Parameter [22]. Typischerweise sind Naturkatastrophen solche Ereignisse, bei denen die tatsächliche Schadenshöhe schwer zu prognostizieren ist. Der entscheidende Parameter, der über eine Auszahlung entscheidet könnte zum Beispiel die Überschreitung einer bestimmten Windgeschwindigkeit sein. Eine Zugverspätung verhält sich insofern analog dazu, als dass die individuelle Schadenshöhe schwer abzuschätzen ist und das Schadensereignis anhand bestimmter Parameter fest definiert werden kann. In diesem Fall wäre die Verspätung, also die Überschreitung einer festgelegten Differenz aus geplanter und tatsächlicher Ankunftszeit am Zielbahnhof der Auslöser, auch oft bezeichnet als Trigger. Nur wenn das Risiko für den Eintritt einer solchen Überschreitung bekannt ist, kann entschieden werden, ob eine Zugfahrt versichert werden kann und zu welchem Preis. Ein zu niedrig berechnetes Risiko würde auf Seite der Versicherung zu hohe Kosten durch Auszahlungen verursachen. Ein zu hoch berechnetes Risiko hingegen würde die Versicherung weniger attraktiv erscheinen lassen und dadurch zu Verlusten durch weniger abgeschlossene Policen führen. Daraus erschließt sich das Interesse einer möglichst präzisen Risikoberechnung und auch das grundlegende Ziel der in dieser Arbeit zu entwickelnden Modelle und Softwarekomponenten.

Oft ist für eine solche Risikoberechnung bereits ein einfaches statistisches Modell ausreichend, welches auf Erfahrung vergangener Ereignisse basiert, um die Wahrscheinlichkeit für einen erneuten Schadenseintritt zu ermitteln. Im Rahmen dieses Projekts könnte ein solches Modell darin bestehen, für eine angeforderte Zugverbindung, vergangene Fahrten der

gleichen Verbindung zu betrachten und den Anteil der verspäteten Verbindungen zu ermitteln. Um zu untersuchen, ob durch die Anwendung der unterschiedlichen Algorithmen des maschinellen Lernens eine tatsächliche Verbesserung erzielt werden kann, soll ein solches Modell als Referenz dienen. Entsprechend ist für alle getesteten Verfahren eine zahlenmäßige Leistungsbewertung anzugeben und diese dann mit dem Referenzmodell zu vergleichen. Die dazu verwendeten Metriken werden im folgenden Kapitel 3.3 erläutert.

3.3 Metriken zur Bewertung der Modelle

Um die unterschiedlichen Modelle hinsichtlich ihrer Leistung miteinander zu vergleichen, werden verschiedene Metriken herangezogen. Diese sollen die Leistung des Modells in der Praxis so gut wie möglich widerspiegeln. Die Auswahl und korrekte Verwendung dieser Vergleichswerte ist daher von großer Bedeutung. So wäre beispielsweise der oft verwendete Accuracy Score in diesem Fall schlecht geeignet. Der Accuracy Score ist eine einfache Metrik, die wie folgt definiert wird:

$$Acc = \frac{(TP+TN)}{(P+N)}$$

Acc - Accuracy, TP - True Positives, TN - True Negatives, P - Positives, N - Negatives

Beschrieben wird die Anzahl der korrekt klassifizierten Beispiele im Verhältnis zur Gesamtzahl aller Dateneinträge [23]. Die Güte einer Klassifikation lässt sich aber für den vorliegenden Sachverhalt schlecht durch die Accuracy bewerten. Das liegt daran, dass die Anzahl der falsch klassifizierten Dateneinträge nicht berücksichtigt wird. Wenn also bei einer binären Klassifikation beide Klassen nicht gleich häufig vorkommen, liefert der Accuracy Score keine Aussagekraft über die Güte des Modells. Deutlich wird das an einem Beispiel mit einem Datensatz aus 1000 Einträgen, die in zwei Klassen aufzuteilen sind. Die Klasse *A* enthält 900 Dateneinträge und die Klasse *B* nur 100. Ein Modell, welches einfach alle Dateneinträge als *A* klassifiziert, würde trotzdem einen Accuracy Score von 90% erzielen. Bei unterschiedlichen Kosten einer Fehlklassifikation ist die Accuracy ebenso wenig geeignet. Es haben sich eine ganze Reihe weiterer Metriken etabliert, welche hier besser funktionieren. Die ausgewählten Metriken, welche zum Vergleich der verschiedenen Modelle herangezogen werden, sind im Folgenden beschrieben.

3.3.1 Konfusionsmatrix

In einer Konfusionsmatrix werden die von einem Modell vorhergesagten Klassen mit den tatsächlichen Klassen gegenübergestellt. Für die Bewertung von binären Klassifikationsmodellen entsteht so eine Tabelle mit zwei Zeilen und zwei Spalten. In Tabelle 3.1 ist der Aufbau einer Konfusionsmatrix für so ein binäres Klassifikationsproblem gezeigt. Jede Zelle enthält die Anzahl der in diese Gruppe fallenden Klassifikationen. Eine fehlerhafte Vorhersage „negativ“, würde entsprechend in die obere rechte Zelle fallen, da es sich um eine falsch negative

Klassifikation handelt. So kann jeder Dateneintrag der Testdaten nach der Klassifikation einer der Zellen zugeordnet werden, sodass die Summe aller Zellen letztlich der Größe der Testdatenmenge entspricht. Ein ideales Modell würde bei der Evaluierung eine Konfusionsmatrix ergeben, die keine einzige falsch negative oder falsch positive Klassifikation enthält.

Der Gefahr bei der Nutzung des Accuracy Scores, eine Verfälschung bei einer stark unterrepräsentierten Klasse zu erhalten, kann durch das Aufstellen einer Konfusionsmatrix begegnet werden [24]. Ein Indikator für eine minderwertige Modellqualität ist dann eine verhältnismäßig größere Anzahl falsch positiver oder falsch negativer Klassifikationen, bei einem gleichzeitig hohen Accuracy Score.

		Vorhergesagte Klasse	
		Positiv	**Negativ**
Tatsächliche Klasse	**Positiv**	True Positive	False Negative
	Negativ	False Positive	True Negative

Tabelle 3.1: Aufbau einer Konfusionsmatrix

3.3.2 Receiver Operating Characteristic (ROC)

Eine ROC-Kurve ist ein Diagramm, welches das Verhältnis der True Positive Rate (TPR) zur False Positive Rate (FPR) darstellt [24]. Die True Positive Rate, auch Sensitivität, gibt die Fähigkeit eines binären Klassifikators an, positive Fälle korrekt zu identifizieren. Sie ist wie folgt definiert:

$$TPR = \frac{TP}{(TP+FN)}$$

TPR - True Positive Rate, TP - True Positives, FN - False Negatives

Die False Positive Rate, auch Spezifität, gibt hingegen die Fähigkeit an, negative Fälle fälschlicherweise als positiv einzustufen. Sie ist wie folgt definiert:

$$FPR = \frac{FP}{(FP+TN)}$$

FPR - False Positive Rate, FP - False Positives, TN - True Negatives

Ein ROC-Chart zeigt auf der x-Achse die FPR und auf der y-Achse die TPR. Daher lässt sich sagen, je näher die Kurve an der oberen linken Ecke des Diagramms liegt, desto besser ist die Leistung des Klassifikators. Ein Klassifikator, welcher einfach zufällig die Klassen zuordnet, würde in einer diagonalen Gerade resultieren. Bei einer ROC-Kurve spielt die Verteilung der Klassen im Datensatz keine Rolle. Daher eignet diese sich gut, um für das vorliegende Problem als Metrik zu dienen. Denn auch die Anzahl der Zugreisen mit mehr als 60 Minuten Verspätung kommt entsprechend seltener vor, als pünktliche Fahrten. Zur besseren Vergleichbarkeit kann die Fläche unterhalb der ROC-Kurve berechnet werden. Diese wird als AUC („area under

the curve“) bezeichnet und fasst das Ergebnis in einer einzigen Zahl zusammen [24]. Der ausschließliche Vergleich des AUC-Score bei ungleichem Klassenverhältnis ist hingegen nicht unumstritten. [25] Für jeden getesteten Algorithmus in Kapitel 4.5 wird deshalb nicht nur der AUC-Score angegeben, sondern auch die zugehörige ROC-Kurve dargestellt.

3.3.3 Precision, Recall und F1-Score

Precision und Recall sind zwei weiteren Metriken, die ebenfalls aus den Werten der Konfusionsmatrix berechnet werden können. Die Precision gibt an, wie viele der positiv vorhergesagten Fälle auch tatsächlich positiv sind. Der Recall beschreibt, wie viele der tatsächlich positiven Fälle auch als positiv vorhergesagt wurden. Sie sind wie folgt definiert:

$$precision = \frac{TP}{(TP+FP)} \quad recall = \frac{TP}{P}$$

Beide Metriken hängen vom Klassenverhältnis im vorliegenden Datensatz ab. Oft ist einer der beiden Werte besonders interessant für das gegebene Problem. In diesem Fall ließe sich argumentieren, dass der Recall von größerem Interesse ist. Eine Positiv-Klassifizierung entspricht der Vorhersage, dass ein Zug mit mehr als 60 Minuten Verspätung ankommt. Für die Anwendung des Modells in der Praxis ist wichtig, dass die meisten verspäteten Zugfahrten auch als solche erkannt werden. Eine pünktliche Fahrt fälschlicherweise als verspätet zu klassifizieren, ist aber dennoch unerwünscht. In der vorliegenden Problematik kann daher nicht exklusiv nur einer der beiden Werte betrachtet werden. Daher erscheint es sinnvoll, eine Verknüpfung beider Werte in die Bewertung mit einzubeziehen. Dies ermöglicht der sogenannte F1-Score. Der F1-Score stellt das harmonische Mittel von Precision und Recall dar. Er kombiniert diese beiden Werte und kann daher auch verwendet werden, wenn die Klassen im Datensatz nicht gleich verteilt sind [26]. Er ist somit generell weniger vom vorliegenden Problem abhängig. Der F1-Score ist wie folgt definiert:

$$F1\text{-}score = 2 \cdot \frac{(precision \cdot recall)}{(precision + recall)}$$

Dem zufolge wird vor allem der F1-Score für die zu testenden Modelle herangezogen. Dieser ermöglicht eine zahlenmäßige Vergleichbarkeit und ist bedenkenlos auch bei ungleichem Klassenverhältnis anwendbar. Zusätzlich weisen die Bestandteile Precision und Recall eine sehr gute Interpretierbarkeit auf, welche bei ROC-basierte Metriken wie dem AUC-Score nicht gegeben ist. Der Recall wird zusätzlich mit angegeben, sodass auch die Zusammensetzung des F1-Scores nachvollziehbar ist.

4 Umsetzung

4.1 Datenaggregation

Für die Erstellung des Vorhersagemodells werden Daten aus verschiedenen Quellen benötigt. Die Aggregation, Formatierung und Speicherung dieser Daten werden im folgenden Kapitel beschrieben. Das Ziel ist hierbei, einen ausreichend großen Datensatz aus einzelnen Zugverbindungen zu generieren. Das bedeutet, dass eine Zugverbindung letztlich durch eine „Zeile“ in einer Tabelle vollständig beschrieben werden muss. In diesem Prozess wurden zwei Datenerhebungen durchgeführt. Auf Grundlage der Erkenntnisse der ersten Erhebung wurde ein weiterer Datensatz aggregiert, um die Vorhersage weiter zu optimieren.

4.1.1 Generierung der Zugverbindungen

Aufgrund der Definition einer Zugverbindung, die mindestens durch zwei Bahnhöfe in Deutschland und einer Abfahrtszeit bestimmt wird, ist es nicht realistisch, eine Datenbasis aller theoretisch möglichen Verbindungen aufzustellen. Die öffentlich verfügbaren Stationsdaten umfassen über 5000 Stationen, sodass allein 25 Mio. verschiedene Verbindungen möglich sind, wenn von nur einer möglichen Route pro Verbindung ausgegangen wird [27]. Die angestrebte Datenmenge wurde nach einigen Tests mit kleineren Mengen auf 100.000 Verbindungen festgelegt. Jede Datenreihe erhält als Index einen Abfahrts- und Zielbahnhof sowie einen Abfahrtszeitpunkt. Mit diesen Daten kann dann eine komplette Zugverbindung über die öffentlichen Fahrplanschnittstellen abgefragt werden. Daher erfolgt die Erhebung in zwei Schritten. Zunächst werden die beabsichtigten Verbindungen aus jeweils diesen drei Indexwerten erzeugt. Im zweiten Schritt wird dann jeweils die fahrplanmäßige Zugverbindung abgefragt.

Für den ersten Schritt muss die Auswahl der Verbindungen möglichst realitätsnah geschehen. Zunächst wurde in einem ersten Versuch immer ein Bahnhof mit der Zeichenkette „hbf“ im Stationsnamen und einer, der dieses Merkmal nicht aufweist, gewählt. Dies erwies sich jedoch als ungeeignet, da beispielsweise reine Fernverkehrsstrecken (Hauptbahnhof zu Hauptbahnhof) hier immer nur in Verbindung mit einem Regionalzug auftraten. Ein besserer Ansatz war es, eine Gewichtung bei der Auswahl der Bahnhöfe vorzunehmen, sodass immer aus dem kompletten Pool ausgewählt wird, aber eben verstärkt Bahnhöfe, mit hohem Passagieraufkommen. Zunächst schien es am naheliegendsten, dafür aus den Stationsdaten die Werte aus dem Feld `weights` heranzuziehen, in welchem die Verkehrsbetriebe eine Gewichtung nach Nutzung der Stationen eintragen können. Bei einem Abgleich der Daten mit verfügbaren Statistiken zeigten sich allerdings starke Abweichungen [28][29]. Möglicherweise sind die dort übermittelten Gewichtungen veraltet oder werden von unterschiedlichen Verkehrsbetrieben erhoben, sodass keine Vergleichbarkeit gegeben ist. Ein anderer Ansatz scheint hier vielversprechender, bei dem für das Ranking die Bahnhofskategorie herangezogen wird.

Diese Kategorie ist ein von der Deutschen Bahn vergebener Wert von 1 bis 7, wobei 1 für sehr große Bahnhöfe mit umfassender Infrastruktur steht und 7 für die kleinsten Bahnhöfe mit entsprechend weniger Reisenden, Gleisen und geplanten Halten [30]. Teilt man die Bahnhöfe nun in Teilmengen entsprechend ihrer Kategorie auf, so kann für die spätere Auswahl jeder Kategorie eine eigene Gewichtung zugeordnet werden. Als Datengrundlage dienen die Zahlen der vergangenen Jahre bezüglich des Passagieraufkommens, insbesondere durch die Drucksache 19/19475 des deutschen Bundestages [29]. Mithilfe dieses Expertenwissens werden den Kategorien geeignete Gewichtungen zugeordnet und manuell so angepasst, dass typische Fernverkehrsbahnhöfe häufig auftreten. Stichprobenartig wurde geprüft, ob viel befahrene Fernverkehrsstrecken, wie beispielsweise Berlin Hbf (tief) nach Hamburg Hbf, enthalten sind. Die finalen Verbindungen wurden abseits der Stichproben nicht zusätzlich durch Umfragen oder weitere Statistiken validiert. Die relativ großen Unterschiede in den Gewichtungen kommen dadurch zustande, dass es beispielsweise von Kategorie 1 nur 21 Bahnhöfe in Deutschland gibt. Daher werden die Gewichtungen entsprechend stark angehoben, um ausreichend Verbindungen mit diesen Bahnhöfen zu generieren. In Tabelle 4.1 sind die Anzahl der Bahnhöfe einer Kategorie und die zugeordneten Gewichtungen aufgeführt.

Kategorie	Anzahl	Gewichtung
1	21	700
2	86	200
3	266	20
4	630	10
5	972	10
6	2499	5
7	894	5

Tabelle 4.1: Vorkommen der Bahnhofskategorien mit zugeordneter Gewichtungen

Als nächstes müssen die Abfahrtszeiten festgelegt werden. In der ersten Datenerhebung sind die 100.000 Verbindungen unabhängig voneinander generiert worden, sodass auch die Abfahrtszeiten zufällig aus einem festgelegten Zeitraum gewählt wurden. Dieser reichte vom 01.01.2022 bis zum 30.04.2022. Die Begrenzung sollte sicherstellen, nicht mit dem Zeitraum des bundesweit geltenden 9-Euro-Tickets zu kollidieren. Dieses war ab Juni 2022 verfügbar und führte ab diesem Zeitpunkt zu einer ungewöhnlich hohen Auslastung des Regionalverkehrs [31]. Die Abfahrtszeiten wurden zufällig aus einem Zeitraum von 05:00 Uhr bis 21:00 Uhr ausgewählt. Dieses Vorgehen bringt allerdings einige Nachteile mit sich. Zum Einen ist die gleichmäßige Verteilung der Abfahrtszeiten nicht realistisch, da beispielsweise 12:00 Uhr deutlich weniger Fahrten angetreten werden als um 7:00 Uhr [32]. Die strikte Begrenzung der Uhrzeit ist außerdem nicht optimal, da Verbindungen über Nacht in dieser Erhebung komplett herausfallen. Wie in den Anforderungen in Kapitel 3.1 beschrieben, soll eine Prognose aber für beliebige Zeitpunkte möglich sein.

Für die zweite, deutlich größere Datenerhebung wurden einige Anpassungen vorgenommen. Die Auswahl der Uhrzeit wurde ebenfalls mit einer Gewichtung getroffen, analog zur Auswahl des Bahnhofs. Dabei wurde eine Statistik über die typische Tagesauslastung herangezogen

[32]. Ziel ist es, dass der Verlauf der Tagesauslastung mit den beiden Spitzen um 7:00 und 17:00 entsprechend berücksichtigt wird. Die Gewichte stellen wieder die Verhältnisse dar. Wenn eine Fahrt um 7:00 sechs Mal häufiger als eine Fahrt um 4:00 gewählt werden soll, erhält diese entsprechend das sechsfache Gewicht. Somit ergeben sich die Gewichtungen, welche in Tabelle 4.2 dargestellt sind.

Stunde	Gewichtung	Stunde	Gewichtung	Stunde	Gewichtung
0:00	0	8:00	6	16:00	5
1:00	0	9:00	6	17:00	6
2:00	0	10:00	5	18:00	4
3:00	0	11:00	4	19:00	2
4:00	1	12:00	4	20:00	1
5:00	4	13:00	4	21:00	1
6:00	6	14:00	4	22:00	1
7:00	6	15:00	4	23:00	0

Tabelle 4.2: Abfahrtsstunden mit zugeordneter Gewichtungen

Unter Verwendung jener Gewichtungen wurden zunächst nur 1000 Verbindungen ohne Datum erstellt. Für diese werden dann aber jeweils 365 mal von 12.12.2021 bis 11.12.2022 Verbindungen abgefragt. Der Zeitraum entspricht genau einem Fahrplanjahr, da am 12. Dezember in der Regel ein neuer Fahrplan in Kraft tritt [33]. So sollten sich auch nach Bereinigung von ungültigen Verbindungen und Duplikaten insgesamt genug Daten ergeben. Der Prozess für diese 1000 Verbindungen ist in einem Python-Skript implementiert. Für die Verarbeitung der Daten wird in diesem und auch in den folgenden Skripten die Python-Bibliothek `pandas` verwendet [34]. Das Skript `journey_helper.py` ist in Anhang A.1 zu finden. Es benötigt die öffentlich verfügbaren Stationsdaten, die in diesem Fall in der Datei `stations.json` erwartet werden. Mit dieser Datenbasis wird eine definierte Anzahl an Verbindungen, nach dem hier beschriebenen Verfahren generiert. In Quelltext 4.1 ist beispielhaft ein Auszug der Ausgabe dieses Skriptes dargestellt. Die entstandenen Daten werden in einer Datei `journeys.csv` gespeichert.

```
1 departure id,arrival id,departure name,departure category,arrival name,arrival category,
      departure time
2 8000092,8000105,Elmshorn,3,Frankfurt (Main) Hbf,1,13:58:00
3 8006203,8000128,Wannweil,5,Göttingen,2,17:35:00
4 8010168,8010406,Hettstedt,6,Berlin Zoologischer Garten,2,16:39:00
5 8000262,8002549,München Ost,1,Hamburg Hbf,1,06:56:00
6 8000128,8012679,Göttingen,2,Prösen,6,07:07:00
7 ...
```

Quelltext 4.1: Auszug der Ausgabe von journey_helper.py

Im nächsten Schritt müssen aus den theoretischen Verbindungen vollständige Zugreisen gemäß Fahrplan abgefragt werden. Dabei werden auch einige Verbindungen entfallen, zum Beispiel wenn keine gültige Reise von der Fahrplanabfrage zurückkommt. Für diese Abfragen werden die Fahrplanschnittstellen der Transit-APIs von Public Transport verwendet [35]. Der Prozess sieht so aus, dass für jeden Tag diese 1000 Verbindungen angefragt und bei einer

gültigen Antwort die Reiseabschnitte in einer CSV-Datei gespeichert werden. So entstehen 365 Dateien, also für jeden Tag eine. Da auf diese Weise theoretisch 365000 HTTP-Requests gesendet werden müssen, wurden die Abfragen parallelisiert. Dabei wurde darauf geachtet, nicht zu viele Anfragen zu stellen, um keine zu hohen Arbeitslasten zu erzeugen. So konnte die Laufzeit der Erfassung von mehreren Tagen auf wenige Stunden reduziert werden. Dieser Arbeitsschritt ist in dem Python-Skript `journey builder.py` implementiert und in Anhang A.2 zu finden. Die vollständigen Antworten der Fahrplanabfragen werden jeweils tageweise in Dateien im JSON-Format gespeichert. In Anbetracht der Datenmenge wurde in diesem Schritt später auf die nicht relationale Datenbank MongoDB gewechselt. Da aber bereits große Teile noch auf Basis der einzelnen Dateien realisiert wurden, ist dies im Folgenden nicht weiter relevant. Für eine erneute Durchführung des Prozesses sollte hier aber direkt der Einsatz einer Datenbank in Betracht gezogen werden, wenn die vollständigen Antworten gespeichert werden sollen.

4.1.2 Verspätungshistorie

Wie bereits beschrieben, ist für die Bewertung einer einzelnen Zugverbindung die unmittelbare Verspätungshistorie von essentieller Bedeutung. Diese enthält bereits wichtige Faktoren, welche einen Einfluss auf die Verbindung haben. Die Historie beschreibt die Verspätungen vergangener Fahrten der exakt gleichen Verbindung. Um diese Information nun mit in den Prozess einfließen zulassen, ist eine geeignete Repräsentation im Datensatz notwendig. Ein Teil der Verspätungshistorie wird daher definiert aus Werten (in Minuten) von vier vergangenen Verbindungen. Diese werden in einem Abstand von jeweils 24 Stunden erhoben. Wenn der aktuelle Zeitpunkt durch t definiert ist und die Verspätung einer Verbindung in $t + 7$ Tage prognostiziert werden soll, wird die gleiche Verbindung zu den Zeitpunkten $t - 24$ Stunden, $t - 48$ Stunden, $t - 72$ Stunden, $t - 96$ Stunden betrachtet und jeweils die finale Verspätung in Minuten am Ziel ermittelt. Analog dazu werden für einen weiteren Teil vier Werte mit jeweils einem Abstand von einer Woche erhoben, sodass sich die Verspätungshistorie durch Mengen aus Verspätungsminuten beschreiben lässt. Beispielhaft könnten die Mengen wie folgt aussehen: $\{0, 84, 55, 0\}$, $\{0, 0, 43, 34\}$. Diese lassen sich dann als Ganzzahlen verarbeiten und sind im Datensatz bezeichnet mit beispielsweise `delay hist 4d 2` für den zweiten Wert der Vier-Tage-Menge, welcher $t - 48$ Stunden zurückliegt. Es ergeben sich dementsprechend acht weitere Merkmale im Datensatz.

In einem ersten Versuch wurde zunächst ein Skript geschrieben, welches für jede Verbindung diese acht Werte ermittelt, indem jeweils die tatsächliche Route über zugfinder.net ermittelt und daraus die letztliche Verspätung entnommen wurde. Dies nahm einige Zeit in Anspruch und wurde daher mit Hilfe der Pythonbibliothek `multiprocessing` parallel durchgeführt. Dabei musste wieder berücksichtigt werden, die Menge der Abfragen zu begrenzen, um ungewollten Lastspitzen bei den verwendeten Endpunkten zu vermeiden.

```
starting chunk 4041/10033
0, 3, 0, 0, 68, 0, 0, 0
```

```
54, 58, 60, 28, 53, 61, 7, 4
7, 1, 10, 3, 81, 65, 62, 56
0, 3, 2, 0, 0, 118, 60, 0
0, 0, 63, 2, 0, 0, 0, 55
30, 22, 3, 5, 15, 2, 43, 24
36, 64, 163, 0, 42, 175, 0, 0
0, 84, 55, 0, 0, 0, 43, 34
1, 0, 3, 39, 0, 30, 0, 0
60, 540, 180, 9999, 60, 0, 9999, 9999
waiting for pool to finish...
chunk 4041/10033 written succesfully
starting chunk 4042/10033
20, 0, 4, 0, 0, 1, 2, 60
65, 63, 66, 63, 65, 64, 67, 21
60, 60, 0, 65, 2, 0, 97, 4
0, 0, 0, 0, 0, 12, 3, 12
```

Quelltext 4.2: Skriptausgabe der ersten Version zur Aggregation der Verspätungshistorie

Für die zweite Datenerhebung war dieser Schritt deutlich einfacher, da wie in Kapitel 4.1.1 beschrieben, jede Verbindung für jeden Tag abgerufen wurde. Somit liegen ab den Datensätzen von Mitte Januar alle vergangenen Fahrten bereits vor und müssen lediglich aggregiert werden. Hierfür wurde das Skript aus der früheren Version angepasst. Es ist in Anhang A.4 enthalten.

4.1.3 Wetterdaten

Forschungsarbeiten im Bereich der Verspätungsprognose bei Zügen durch maschinelles Lernen untersuchen häufig den Einfluss von Wetterdaten auf die Verspätung. Im Artikel „Train delay analysis and prediction based on big data fusion“ von Pu Wang und Qing-peng Zhang wurde vor allem bei Schneeregen, Hagel und schwerem Schneefall ein Anstieg von Zugverspätungen beschrieben [5]. Untersuchungen im asiatischen Raum konnten oft eine Korrelation des Wetters nachweisen [6]. Die Ausgangssituation ist allerdings nicht vergleichbar mit dem Zugverkehr in der DACH-Region. Weiterhin sind hier ebenfalls keine Untersuchungen im Bezug auf komplette Zugverbindungen bekannt. Daraus ergibt sich die Motivation, zu evaluieren, ob unter den gegeben Bedingungen ein Einfluss von Wetterinformationen auf die Verspätungswahrscheinlichkeit gezeigt werden kann. Die Verfügbarkeit ist zumindest bei historischen Daten gut und es existieren zahlreiche Tools zur Verarbeitung der Daten [36]. Zu beachten ist aber, dass bei Echtzeitvorhersagen, die mit dem fertigen Modell durchgeführt werden sollen, auf Wetterprognosen zurückgegriffen werden muss. Dies führt bei größeren Zeitfenstern zu einer geringeren Genauigkeit, im Vergleich zu den historischen Wetterdaten, die für alle Tage der Trainings- und Testdaten rückwirkend natürlich vorliegen. Dadurch wird im Modell mit einem Wissensvorteil trainiert, da zu diesem Zeitpunkt bereits bekannt ist, wie das Wetter tatsächlich war. Eine Möglichkeit dies zu umgehen, ist die Verwendung von Prognosedaten bereits im Trainingsprozess. Ein Dateneintrag mit einer Zugverbindung am 14. März würde entsprechend mit Wettermerkmalen auf Basis der 7-Tage Prognose vom 7.

März angereichert werden. Dies erfordert wiederum die entsprechende Verfügbarkeit auch im Hinblick auf spezielle Wetterinformationen, wie zum Beispiel Schneefall. Außerdem bringt es eine weitere Unbekannte in Form der Güte der Wetterprognose mit in den Prozess. Für das weitere Verfahren wurde entschieden, für den Trainingsprozess historische Daten zu verwenden.

Für die Untersuchung wird als Datenquelle der ERA5-Land-Datensatz des Copernicus Climate Data Store genutzt [37]. Dieser beinhaltet stündlich aggregierte Satellitendaten in einem Rasterformat. Satellitendaten haben den Vorteil, dass für nahezu jeden beliebigen Ort immer Werte verfügbar sind. Die Daten liegen mit einer räumlichen Auflösung von 9 km vor und werden für einen Zeitraum von 1950 bis heute bereitgestellt. Für die Untersuchung werden nur Daten aus dem Zeitraum Dezember 2021 bis Dezember 2022 benötigt. Die Daten sind immer mit einer Verzögerung von etwa drei Monaten verfügbar. Dies führte in der zweiten Datenerhebung zu einem Problem, da der Dezember 2022 noch nicht abrufbar war und dadurch alle Verbindungen im Dezember bei den Versuchen mit Wettermerkmalen nicht mit in die Trainingsdaten einbezogen werden konnten. Der Rest des Zeitraumes war hingegen vollständig abgedeckt. Insgesamt werden für die Untersuchungen des Wettereinflusses sechs Werte für jede Zugverbindung erfasst und in den Datensatz geschrieben. Die Auswahl von Niederschlag und insbesondere Schneefall ist aufgrund der Ergebnisse genannter Studien getroffen worden [5]. Die Temperatur wird ebenfalls mit einbezogen, da diese bekanntermaßen eine Rolle bei der Verspätungswahrscheinlichkeit spielen kann [6]. Die Merkmale sind mit ihren jeweiligen Einheiten und einer kurzen Beschreibung in Tabelle 4.3 aufgeführt.

Merkmal	Einheit	Beschreibung
era5_temperature	K	durchschnittliche Temperatur an der Erdoberfläche in Kelvin
era5_dewpoint_temperature	K	Temperatur des Taupunktes an der Erdoberfläche in Kelvin
era5_precipitation	m	Niederschlagsmenge (Regen und Schnee) an der Erdoberfläche in Meter Wasserhöhe, bei Verteilung innerhalb des Rasterquadrats
era5_snow_coverage	%	Prozentsatz der mit Schnee bedeckten Erdoberfläche
era5_snow_depth	m	durchschnittliche Schneehöhe an der Erdoberfläche in Metern
era5_snow_fall	WE	auf die Erdoberfläche gefallene Menge an Schnee in Meter Wasseräquivalent

Tabelle 4.3: Erläuterung der aggregierten Wettermerkmale aus dem ERA5-Land Datensatz

Nach dem Abruf liegen die Wetterdaten im NetCDF-Format vor. Für die Anreicherung der Zugverbindungen mit den Wetterdaten müssen jeweils für eine Zeit- und Koordinatenangabe die am nächsten liegenden Werte in den gerasterten ERA5-Daten ausgewählt werden. Es sind hier verschiedene Ansätze denkbar, für das Problem, die am nächsten gelegenen Datenpunkte zu finden. Eine Möglichkeit ist das sogenannte nearest-Neighbor-Lookup mit Hilfe eines k-d-Baums [38]. In diesem Fall werden die NetCDF-Daten mit der Python-Bibliothek

xarray eingelesen, da diese bereits eine Implementierung dafür mitbringt. Die Funktion xarray.DataArray.sel wird mit dem Parameter method='nearest' aufgerufen, um genau die erforderlichen Datenpunkte zu extrahieren [36]. Das Erweitern des Datensatzes mit den ausgewählten sechs Wettermerkmalen für jeweils Start- und Zielbahnhof, wurde in einem separaten Python-Skript weather helper.py implementiert. Dieses ist in Anhang A.5 zu finden.

4.1.4 Feiertage

Zur Untersuchung der Auswirkung von Feiertagen auf die Verspätungswahrscheinlichkeit, werden die Bundesländerübergreifenden Feiertage aus dem Jahr 2022 in den Datensatz integriert. Die Daten werden in der Form eines Booleans gespeichert, welcher wahr ist, wenn der jeweilige Tag am Abfahrtsdatum ein Feiertag ist. Dazu wird die Python-Bibliothek holidays verwendet [39]. Diese liefert Sets aus Date-Objekten mit den gewünschten Feiertagen. Theoretisch sind auch Ferien damit abrufbar, diese werden aber aufgrund der Unterschiede zwischen den Bundesländern zunächst nicht berücksichtigt. Es wird daher geprüft, ob das Reisedatum in den Feiertagen enthalten ist. Ein weiteres möglicherweise interessantes Merkmal sind Brückentage, beziehungsweise verlängerte Wochenenden. Diese entstehen durch die Kombination mit einem Feiertag, wie beispielsweise an Ostern oder Pfingsten. Die Idee ist, dass diese unter Umständen zu einer erhöhten Auslastung führen könnten. Um dies zu untersuchen, muss ermittelt werden, ob das Reisedatum auf ein verlängertes Wochenende fällt oder auf unmittelbar einen Tag davor. Wenn also Dienstag Feiertag ist, könnte bereits am Freitag eine höhere Auslastung zu erwarten sein. In den Datensatz können die Informationen wieder als Boolean integriert werden. Für die Ermittelung dieser Information wird ebenfalls auf eine bestehende Python-Bibliothek zurückgegriffen. Mit der Funktion spot holiday -bridges aus der Bibliothek python-long-weekends können die langen Wochenenden mit Hilfe der bereits bekannten Feiertage herausgefiltert werden [40]. Somit können diese beiden Merkmale schnell und ohne externe Abfragen aggregiert werden. In Quelltext 4.3 ist der entsprechende Code gezeigt.

```
1  # load holidays and long weekends
2  holidaysDE = holidays.DE(years=[2021, 2022])
3  bridges, long weekends = spot holiday bridges(start='2021-01-01',
4    end='2022-12-31',
5    holidays=holidaysDE)
6
7  # checks if date is a holiday
8  def isHoliday(date):
9    return (date in holidaysDE)
10
11 # checks if date is one day before a long weekend
12 def isLongWeekend(date):
13   nextDay = date + timedelta(days=1)
14   if pd.Timestamp(date.date()) not in long weekends and
15     pd.Timestamp(nextDay.date()) in long weekends:
```

```
16      return True
17    else:
18      return False
```

Quelltext 4.3: Feature-Aggregation von Feiertagen und langen Wochenenden

4.1.5 Weitere Merkmale

Eine Reihe weiterer Merkmale wurde analog zu den Feiertagen bereits bei der Generierung des Datensatzes aus den entstandenen Zugverbindungen erhoben. In diesem Schritt wurde aus den einzelnen Tagesmengen, die mit Hilfe des Skripts `journey builder.py` generiert wurden, die Basis der Datenreihen für den Trainingsdatensatz gebildet. Dazu ist ein Skript entstanden, welches für alle Verbindungen über die Funktion `generate set(key, value)` eine Datenreihe zusammenstellt und diese in einen Pandas-Dataframe schreibt. Aus diesem kann anschließend eine CSV-Datei generiert werden. Im folgenden Quelltext 4.4 ist die Rückgabe dieser Funktion mit den entsprechenden Merkmalen gezeigt. Das vollständige Skript ist im Anhang A.3 zu finden.

```
1  journey data = {
2      'dep': dep,
3      'arr': arr,
4      'dep year': sched dep date.year,
5      'dep month': sched dep date.month,
6      'dep day': sched dep date.day,
7      'dep time': parse minutes of day(sched dep date.isoformat()),
8      'dep lat': value[0]['origin']['location']['latitude'],
9      'dep long': value[0]['origin']['location']['longitude'],
10     'arr lat': value[legNumber - 1]['destination']['location']['latitude'],
11     'arr long': value[legNumber - 1]['destination']['location']['longitude'],
12     'dist': get distance(value, legNumber),
13     'legs': legNumber,
14     'min transfer': get minimal transfer minutes(value),
15     'day of week': sched dep date.weekday(),
16     'is holiday': is holiday(sched dep date),
17     'long we': is long weekend(sched dep date),
18     'final delay': get final delay(retrieve arrival date(zf), sched arr date)
19 }
```

Quelltext 4.4: Rückgabe der `generate set`-Funktion, welche zum Generieren der Basisfeatures asynchron aufgerufen wird

Anzumerken ist, dass hier bereits die finale Verspätung, aus welcher später die Zielklasse (Label) abgeleitet wird, enthalten ist. Hingegen sind keine historischen und wetterbezogenen Daten sowie die Reisedauer enthalten. Diese wurden in einer späteren Iteration, wie in den vorangegangenen Kapiteln beschrieben, hinzugefügt. Erwartungsgemäß ist die Erhebung einiger Merkmale weniger komplex, wie es zum Beispiel bei den Koordinaten der Fall ist, welche direkt in den Stationsdaten mitgeliefert werden. Andere hingegen, wie beispielsweise die historischen Verspätungen, erfordern einen entsprechend größeren Aufwand. Bei den Koordinaten ist die Idee, regionale Tendenzen beispielsweise auf Bundeslandebene bezüglich

des Verspätungsrisikos mit zu erfassen. Die Anzahl der Reiseabschnitte ist ebenfalls enthalten und entspricht nach Bereinigung von Fußwegen, welche teilweise als einzelner Abschnitt in der Verbindung ausgewiesen werden, der Anzahl der Umstiege. Es wird auch die Umstiegszeit einer Reise in Minuten mit einbezogen. Da dieses Merkmal abhängig von der Anzahl der Umstiege mehrfach oder gar nicht enthalten sein kann, wird lediglich das Minimum ermittelt. Dies basiert auf der Annahme, dass eine kurze Umstiegszeit von beispielsweise 5 Minuten ein hohes Potenzial für einen verpassten Anschlusszug aufweist. Bei Reisen ohne Umstieg liegt dieses Merkmal nicht vor und das Risiko eines verpassten Anschlusszuges liegt bei null. Deshalb wird an dieser Stelle ein sehr großer Wert eingetragen. Dies schien der beste Weg zu sein, um die Relationen des numerischen Merkmals beizubehalten, erfordert aber die Annahme, dass sich eine höhere Umstiegszeit positiv auf das Verspätungsrisiko auswirkt. Alternativen, welche diese Annahme nicht erfordern, stellen das Entfernen des Merkmals oder die Nutzung eines Ersetzungsverfahren dar. Die Distanz wird als Luftlinie in Kilometer zwischen Abfahrts- und Ankunftsbahnhof über die Koordinaten berechnet. Dafür wird die Pythonbibliothek `geopy` verwendet [41]. In Tabelle 4.4 sind alle finalen Merkmale mit ihrer Bezeichnung im Datensatz und einer kurzen Beschreibung aufgeführt.

4.2 Testumgebung, Datenimport und Bereinigung

Für die weitere Untersuchung wird die webbasierte Entwicklungsumgebung Jupyter Notebook verwendet [42]. Diese interaktive Umgebung bietet einige Vorteile, insbesondere der direkten Einbindung von Diagrammen, welche mit Hilfe der Pythonbibliothek *matplotlib* erstellt wurden. Jupyter eignet sich sehr gut für experimentelle Arbeiten und der dokumentierten Veröffentlichung von Code und ist dementsprechend nicht für das Ausführen produktiver Anwendungen vorgesehen.

Für die weitere Vorbereitung müssen zunächst die Daten importiert werden. Da der fertige Datensatz im CSV-Format vorliegt, kann dieser direkt mit der pandas-Bibliothek in die Umgebung importiert werden [34]. Der Datensatz umfasst an dieser Stelle 315.605 Zugverbindungen, ca. 860 pro Tag. Das ist deutlich mehr als die in den Anforderungen genannten 100.000, was aber damit zusammenhängt, dass zum einen die Unterteilung in Trainings- und Evaluierungsdaten noch nicht durchgeführt wurde und zum anderen die Bereinigung ungültiger Datenreihen eine weitere Verkleinerung mit sich bringt. Im Rahmen der weiteren Datenvorbereitung werden zunächst alle Datenreihen mit fehlenden Einträgen gelöscht. Außerdem werden Einträge gelöscht, die zum Beispiel eine unrealistisch lange Reisezeit von über 12 Stunden aufweisen oder mehr als 6 Umstiege haben. Schließlich wird eine Spalte für das Label, also das Ziel der Vorhersage hinzugefügt, welches den Wert `true` im Falle einer Verspätung über 60 Minuten enthält. Im folgenden Quelltext 4.5 sind die Operationen gezeigt.

```
1  # clean up and fix datatypes
2  df = df.copy().dropna()
3
```

Merkmal	Inhalt	Datentyp
dep	Nummer des Abfahrtsbahnhofs	int64
arr	Nummer des Ankunftsbahnhofs	int64
dep_month	Monat des Abfahrtdatums	int64
dep_day	Tag des Abfahrtdatums	int64
dep_time	Abfahrtszeit in Minuten seit 0:00	int64
arr_lat	Breitengrad des Ankunftsbahnhofs	float64
arr_long	Längengrad des Ankunftsbahnhofs	float64
day_of_week	Wochentag	int64
delay_hist_4d_1	Verspätung vor einem Tag	float64
delay_hist_4d_2	Verspätung vor zwei Tagen	float64
delay_hist_4d_3	Verspätung vor drei Tagen	float64
delay_hist_4d_4	Verspätung vor vier Tagen	float64
delay_hist_4w_1	Verspätung vor einer Woche	float64
delay_hist_4w_2	Verspätung vor zwei Wochen	float64
delay_hist_4w_3	Verspätung vor drei Wochen	float64
delay_hist_4w_4	Verspätung vor vier Wochen	float64
dep_lat	Breitengrad des Abfahrtsbahnhofs	float64
dep_long	Längengrad des Abfahrtsbahnhofs	float64
dist	Distanz	int64
is_holiday	liegt Feiertag vor	bool
legs	Anzahl der Reiseabschnitte (Züge)	int64
long_we	liegt angrenzendes langes Wochenende vor	bool
min_transfer	kleinste Umstiegszeit der Reise	int64
duration	geplante Dauer der Reise	float64
era5_dep_temperature	Temperatur am Abfahrtsbahnhof	float64
era5_arr_temperature	Temperatur am Ankunftsbahnhof	float64
era5_dep_dewpoint	Taupunkt am Abfahrtsbahnhof	float64
era5_arr_dewpoint	Taupunkt am Ankunftsbahnhof	float64
era5_dep_precipitation	Niederschlag am Abfahrtsbahnhof	float64
era5_arr_precipitation	Niederschlag am Ankunftsbahnhof	float64
era5_dep_snow_cover	Schneebedeckung am Abfahrtsbahnhof	float64
era5_arr_snow_cover	Schneebedeckung am Ankunftsbahnhof	float64
era5_dep_snow_depth	Schneetiefe am Abfahrtsbahnhof	float64
era5_arr_snow_depth	Schneetiefe am Ankunftsbahnhof	float64
era5_dep_snow_fall	Schneefallmenge am Abfahrtsbahnhof	float64
era5_arr_snow_fall	Schneefallmenge am Ankunftsbahnhof	float64

Tabelle 4.4: Auflistung aller Merkmale des finalen Datensatzes mit Beschreibung

```
4  # delete journeys that are too long (>12h)
5  df = df[df.duration < 720]
6
7  # delete records with missing or negative min transfer
8  # while keeping 1-leg journeys
9  df = df[(df.min transfer >= 0)]
10 df = df[((df.min transfer == 9999) & (df.legs == 1)) | (df.min transfer <= 9999)]
11
12 # delete records with unrealistic leg amount
13 df = df[df.legs < 6]
14
15 # add 60 minutes target class (true if delay is more than 60 minutes)
16 df['delayed 60'] = df.apply(lambda row: row.final delay > 59 , axis=1)
```

Quelltext 4.5: Bereinigung des Datensatzes und Erstellen des Labels

Der Datensatz enthält nach der Bereinigung noch 221.018 Zugverbindungen. Neben dem vollständigen Entfernen können auch Verfahren zum Ausfüllen von fehlenden Werten in Betracht gezogen werden, wie zum Beispiel das sogenannte „imputing“ [43]. Wie in Kapitel 4.1.3 beschrieben, fallen allein durch die fehlenden Wetterdaten im Dezember, hier alle betreffenden Verbindungen weg. Dies lässt sich in der Datenvorbereitung auch nicht durch derartige Verfahren sinnvoll ausfüllen, da sonst alle Wettermerkmale im Dezember nur auf Grundlage des restlichen Jahres ausgefüllt werden.

4.3 Aufteilung in Trainings- und Evaluierungsdaten

Zur Evaluierung des Modells muss der Datensatz in Trainings- und Evaluierungsdaten aufgeteilt werden. Dabei sind einige besondere Aspekte zu berücksichtigen. Würde eine Aufteilung des entstandenen Datensatzes durch einfaches „sampling“, also dem Herausnehmen von beispielsweise 20 % der Verbindungen vorgenommen werden, so entstehen zwei grundlegende Probleme. Zum einen liegt so keine zeitliche Trennung vor. Das bedeutet, dass Verbindungen im Evaluierungsdatensatz vorkommen würden, die chronologisch vor den Verbindungen im Trainingsdatensatz stattgefunden haben. So könnte das Modell einen unrealistischen Vorteil haben, da es mit Daten aus der Zukunft trainiert wird. Zum anderen würden im Evaluierungsdatensatz auch nur die ursprünglichen 1000 Routen vorkommen, sodass mit den gleichen Zugverbindungen nur an verschiedenen Zeitpunkten evaluiert werden würde. Das Modell hat aber die Anforderung, für beliebige Verbindungen zu funktionieren, weshalb dies kein geeignetes Vorgehen ist. Zur Vermeidung dieses Problems wird der Datensatz zweimal geteilt. Zunächst wird eine zeitliche Trennung am 31.07.2022 vorgenommen. So wird gewährleistet, dass nur die Daten bis zu diesem Tag zum Lernen verwendet werden und alle Verbindungen im Testdatensatz sich zeitlich nach den Trainingsdaten ereignet haben. Aktuell kommen aber in beiden Datensätzen noch die gleichen knapp 1000 Zugverbindungen vor. Deshalb werden im zweiten Schritt 80 % dieser Verbindungen herausgenommen und nur noch für die Trainingsdaten verwendet. Aus den Testdaten werden alle Datensätze mit den ausgewählten 80 % entfernt, sodass hier die restlichen 20 % verbleiben. Dadurch entstehen zwei Datenmengen, die sowohl im Hinblick auf den Zeitraum als auch in der Zusammensetzung

der Routen vollständig disjunkt sind. Ein Nachteil dieser Vorgehensweise ist jedoch, dass eine große Menge an erhobenen Daten nicht mit in den Trainingsdatensatz einfließt. Außerdem können saisonale Effekte auf diese Weise nicht vollständig berücksichtigt werden, da das Modell aus Daten von Januar bis Juli lernt und auf Daten von August bis November getestet wird. Eine Aufteilung auf verschiedene Jahre, die dies ermöglichen würde, hätte wiederum zur Folge, dass die Routenberechnung im Vorjahr gemäß dem damals geltenden Fahrplan erfolgen müsste. Da öffentlich verfügbare Fahrplanschnittstellen diese Möglichkeiten nicht bieten und eine eigene Umsetzung der Verbindungssuche nicht trivial ist, wurde der beschriebene Weg mit Daten aus nur einem Fahrplanjahr gewählt. In den folgenden Quelltexten 4.6 und 4.7 ist die Implementierung dieser Trennung gezeigt.

```
# split by train routes (departure,arrival)
# to prevent learning on the same journey
grouped journeys = df.groupby(['dep', 'arr'])
print(f'{len(grouped journeys)} unique train routes over 365 days')

n = int(grouped journeys.ngroups * 0.8)
train groups = [g[1] for g in list(grouped journeys)[:n]]

n = grouped journeys.ngroups - n
test groups = [g[1] for g in list(grouped journeys)[-n:]]

df train = pd.concat(train groups)
df test = pd.concat(test groups)
```

Quelltext 4.6: Implementierung der Aufteilung nach Zugverbindung

```
# split time segments (on 31.07.2022)
# to prevent learning on future data
df train = df train[df train.dep month < 8]
df test = df test[df test.dep month >= 8]

train x = df train.drop(columns=['delayed 60', 'final delay'])
train y = df train[['delayed 60']]
test x = df test.drop(columns=['delayed 60', 'final delay'])
test y = df test[['delayed 60']]
```

Quelltext 4.7: Implementierung der Aufteilung nach Reisedatum

Final entstehen so jeweils zwei Dataframes für den Trainings- und Testdatensatz, wie in Quelltext 4.7 gezeigt. Die Merkmale werden mit `train x` und `test x` bezeichnet und die Labels mit `train y` und `test y`. Zu beachten ist, dass die Labels nicht die Verspätungsminuten sind, sondern lediglich die als Boolean gespeicherte Information, ob eine Verbindung mit mehr als 60 Minuten Verspätung eingetroffen ist. Dies ist die Zielklasse, die das Modell vorhersagen soll. Für die anstehenden Versuche wird außerdem das Klassengewicht oder auch Klassenverhältnis berechnet. Obwohl einige Algorithmen dieses automatisch ermitteln, wird es wenn möglich mit übergeben. Dazu wird das Verhältnis aus pünktlichen und verspäteten

Verbindungen berechnet, was zu einem Klassengewicht von $3,4283$ führt. Dieser Wert wird für alle folgenden Modelle verwendet und, wie in Quelltext 4.8 gezeigt, in der Variable `weight` gespeichert.

```
1  # weight multiplier (sum negative / sum positive)
2  weight = (len(df_train) - df_train['delayed_60'].values.sum()) / df_train['delayed_60'].
       values.sum()
```

Quelltext 4.8: Berechnung des Klassengewichts für verspätete Verbindungen im Trainingsdatensatz

4.4 Evaluierung der Merkmale

Um die genutzten Daten und Merkmale hinsichtlich ihrer Aussagekraft bewerten zu können, wurde initial nach dem Hinzufügen die sogenannte Feature Importance verglichen. Es existieren verschiedene Möglichkeiten diese zu ermitteln. In diesem Fall wurde die mittlere Reduzierung der sogenannten Gini-Impurity in einem Random Forest herangezogen. Das genaue Verfahren zum Erstellen eines solchen Klassifikators, die verwendeten Bibliotheken und die Ermittelung der Gini-Impurity sind in Kapitel 4.5.2 beschrieben. Die Eigenschaft `feature_importances` des trainierten Modells zeigt schließlich den Einfluss der Merkmale auf die Klassifikation. Es wurde außerdem noch ein weiteres Verfahren namens Permutation Based Feature Importance getestet, da dieses in einigen Fällen akkuratere Ergebnisse liefern soll [44]. Dies führte allerdings zu den gleichen Ergebnissen. Diese sind in Abbildung 4.1 dargestellt.

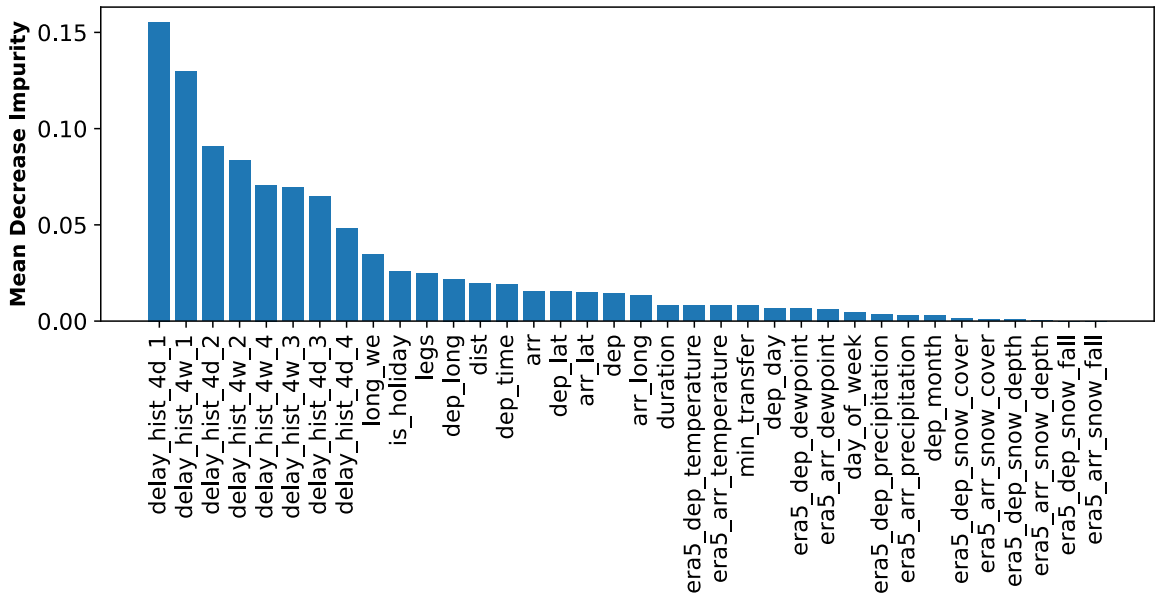


Abbildung 4.1: Feature Importance einschließlich historischer Daten, Bildquelle: eigene Darstellung

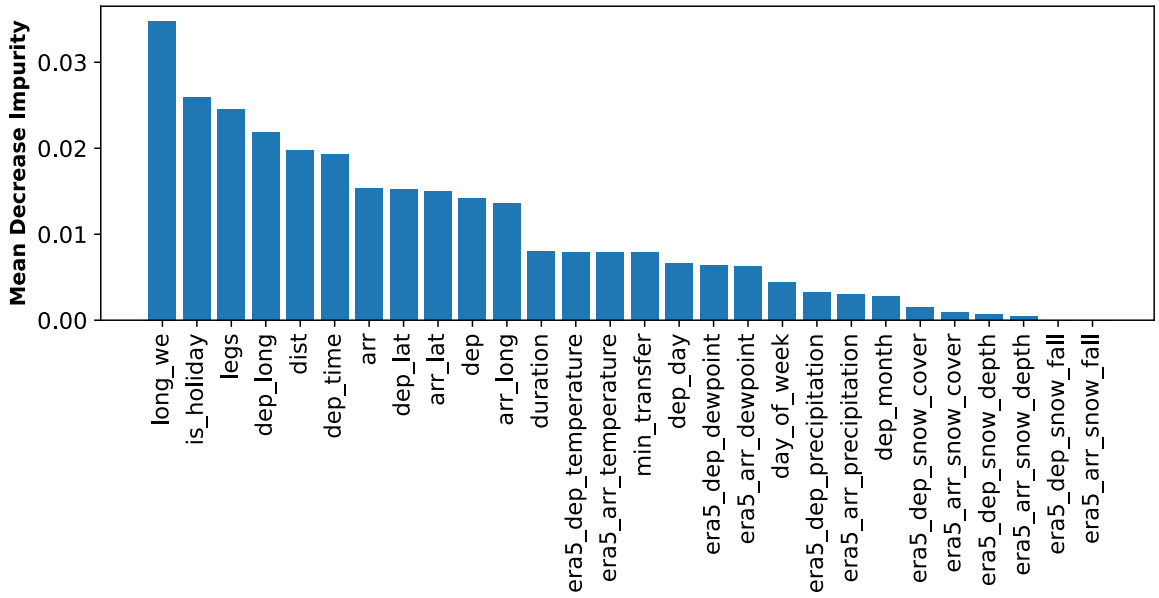


Abbildung 4.2: Feature Importance **ohne** historische Daten, Bildquelle: eigene Darstellung

In Abbildung 4.2 sind die Ergebnisse noch einmal ohne die historischen Verspätungen gezeigt, um einen besseren Eindruck von den eher spezifischen Merkmalen zu erhalten. Da die Historie und - wie sich zeigt - vor allem die Fahrten vor 24 Stunden und vor 7 Tagen, eine recht starke Aussagekraft haben, ist es schwierig, die Bedeutung der anderen Merkmale überhaupt zu erkennen. Durch die separate Betrachtung, die zwar für die weitere Modellentwicklung keine Rolle spielt, können aber einige interessante Aspekte hinsichtlich des Verspätungsrisikos einer Bahnverbindung gezeigt werden. Auffällig ist die starke Aussagekraft von Feiertagen und langen Wochenenden. Dies bestätigt die Annahme, dass an solchen Tagen tatsächlich eine erhöhte Auslastung für Verspätungen sorgt. Weniger überraschend ist die Tatsache, dass die Anzahl der Umstiege, die aus dem Merkmal `legs` hervorgeht, eine entsprechend große Bedeutung hat. Zu erkennen ist auch die Aussagekraft des Längengrades des Abfahrtsbahnhofs. Dies lässt auf Unterschiede in der Pünktlichkeit zwischen westlichen und östlichen Teilen Deutschlands schließen. Auch die Abfahrtszeit `dep time` hat eine nennenswerte Bedeutung. Diese gibt die Tageszeit in Minuten an. Es spielt also durchaus eine Rolle ob eine Zugfahrt um 6:00 oder erst um 17:00 angetreten wird. Von den Wetterinformationen scheint besonders die Temperatur (`era5 dep temperature`) und der Taupunkt (`era5 dep dewpoint`) relevant zu sein. Dies deckt sich mit der Erwartung durch die bestehenden Arbeiten [5][6]. Weiterhin ist zu erkennen, dass die Wetterdaten für Start- und Zielbahnhöfe jeweils immer einen ähnlichen Einfluss zeigen. Denkbar ist, dass die Wetterlage an den beteiligten Bahnhöfen bei den meisten Fahrten im Datensatz sehr ähnlich ist.

4.5 Untersuchung verschiedener Algorithmen

4.5.1 Lazy Predict

Zur Erstellung einer ersten Übersicht, welche Arten von Algorithmen eventuell besser für die vorliegende Problematik geeignet sind, wurde auf ein Projekt namens „Lazy Predict“ zurückgegriffen [45]. Dies ist eine Python-Bibliothek, welche verschiedene Modelle in Standardkonfiguration nacheinander erstellt, diese anhand typischer Metriken evaluiert und die Ergebnisse anschließend ausgibt. Auch wenn diese Modelle wirklich nur sehr grundlegende Tendenzen zeigen, so lässt sich dennoch ableiten, bei welchen Algorithmen eine nähere Betrachtung lohnenswert erscheint. Der Vorteil ist in erster Linie die Zeitersparnis, da -sofern die Datenvorbereitung abgeschlossen ist- mit lediglich einem Aufruf eine Menge Informationen erlangt werden können. In Tabelle 4.5 sind die Ergebnisse dargestellt. Zu beachten ist, dass der F1-Score hier für alle Klassen berechnet wird und dann der Mittelwert gebildet wurde. Dies ist bei binären Klassifikationen im Normalfall nicht erwünscht und es würde hier nur der Wert für die eine positive Klasse, wie in Kapitel 3.3.3 beschrieben, gebildet werden. Der F1-Score ist daher nicht mit den Ergebnissen aus den folgenden Untersuchungen vergleichbar.

Eine interessante Beobachtung ist auch, dass der Bernoulli Naive Bayes Klassifikator den besten Recall erzielt. Erwartungsgemäß wird dieses Verfahren aufgrund der starken Vereinfachung in vielen Fällen durch Ensemble-Learning-Algorithmen übertroffen [46]. In Kapitel 4.5.6 wird daher genauer auf die Arbeitsweise des Verfahrens und auf mögliche Gründe für diese Beobachtung eingegangen.

4.5.2 Decision Tree

Decision Trees, im folgenden als Entscheidungsbäume bezeichnet, sind eine häufig verwendete Methode im Bereich des maschinellen Lernens. Sie sind sowohl für Klassifikation als auch für Regression einsetzbar [47]. Ein Entscheidungsbaum kann durch ein Baumdiagramm dargestellt werden, welches aus Entscheidungsknoten, Blättern und Verzweigungen besteht. Jeder Knoten repräsentiert dabei eine Entscheidung und jedes Blatt eine Vorhersage. Der Entscheidungsbaum wird im Trainingsprozess aufgebaut und bildet Regeln aus den Trainingsdaten, die eine möglichst optimale Trennung der Klassen erlauben. Bei einer Klassifikation entsteht im Idealfall ein Baum, dessen Blätter nur noch Dateneinträge einer Klasse enthalten. An jedem Knoten muss daher das stärkste Trennkriterium für eine optimale Klassifizierung gefunden werden. Dazu werden die Merkmale anhand einer Metrik wie beispielsweise der sogenannten Gini-Impurity miteinander verglichen. Diese wird gebildet aus dem Gini-Index, welcher die Wahrscheinlichkeit angibt, zwei Dateneinträge der selben Klasse aus der Menge zu ziehen. Mit der Gini-Impurity kann daher gemessen werden, wie rein eine bestimmte Menge ist. Entscheidungsbäume sind anfällig für Overfitting und müssen daher durch bestimmte Regeln, wie einer festgelegten minimalen Blattgröße oder Verfahren wie dem sogenannten

Model	Accuracy	ROC AUC	F1 Score	Recall Score
BernoulliNB	0.82	0.72	0.82	0.55
NearestCentroid	0.81	0.69	0.80	0.49
AdaBoostClassifier	0.84	0.67	0.82	0.39
LGBMClassifier	0.84	0.66	0.82	0.36
RandomForestClassifier	0.84	0.65	0.81	0.32
Perceptron	0.64	0.62	0.68	0.57
BaggingClassifier	0.80	0.61	0.78	0.29
XGBClassifier	0.83	0.61	0.79	0.25
KNeighborsClassifier	0.80	0.61	0.78	0.28
GaussianNB	0.80	0.61	0.78	0.28
QuadraticDiscriminantAnalysis	0.78	0.60	0.76	0.29
SVC	0.82	0.59	0.78	0.20
PassiveAggressiveClassifier	0.80	0.59	0.77	0.24
SGDClassifier	0.81	0.59	0.77	0.21
ExtraTreesClassifier	0.82	0.59	0.78	0.19
DecisionTreeClassifier	0.68	0.59	0.70	0.42
CalibratedClassifierCV	0.81	0.58	0.77	0.19
LogisticRegression	0.81	0.58	0.77	0.18
LinearDiscriminantAnalysis	0.81	0.57	0.76	0.18
LinearSVC	0.81	0.57	0.76	0.17
ExtraTreeClassifier	0.69	0.56	0.70	0.34
RidgeClassifier	0.81	0.56	0.75	0.13
RidgeClassifierCV	0.81	0.56	0.75	0.13
DummyClassifier	0.80	0.50	0.71	0.00

Tabelle 4.5: Ergebnisse der Lazy Predict-Bibliothek, absteigend nach AUC sortiert, über beide Klassen gemittelter F1-Score

„pruning“, also dem Entfernen von Knoten, reguliert werden [47]. Vorteilhaft bei Entscheidungsbäumen ist wiederum, dass eine Achsen-parallele Trennung sehr gut interpretierbar ist und derartige Modelle daher eine hohe Nachvollziehbarkeit der Ergebnisse aufzeigen. Für die Erstellung des Entscheidungsbaums wird aus der Python-Bibliothek scikit-learn die Klasse `DecisionTreeClassifier` verwendet [48]. Diese unterstützt keine direkte Verarbeitung kategorischer Merkmale und generiert ausschließlich binäre Bäume [49]. Das bedeutet, dass alle Merkmale entsprechend vorverarbeitet werden müssen, um korrekt interpretiert zu werden. Bei binären Bäumen entwickelt jeder Knoten immer genau zwei neue Knoten beziehungsweise Blätter.

Es wurden zunächst die Standardparameter verwendet, um die in Kapitel 3.3 beschriebenen Metriken in die Umgebung zu integrieren. Je nach verwendeter Bibliothek müssen die Inputs entsprechend angepasst oder formatiert werden. Wenn sowohl der verwendete Klassifikator als auch die Metrik aus dem scikit-learn-Framework verwendet werden, ist dies nicht erforderlich. Zusätzlich werden mithilfe der Klasse `StandardScaler` alle Merkmale skaliert und zentriert, sodass sie eine Standardnormalverteilung ergeben mit einem Mittelwert von null

und einer Standardabweichung von eins. Dies ist zwar speziell für den Entscheidungsbaum nicht zwingend erforderlich, wird aber für andere Algorithmen notwendig und kann daher einfach mit in die generelle Verarbeitungsabfolge integriert werden.

Mit Blick auf die Metriken wurde im nächsten Schritt die Parametrisierung des Entscheidungsbaums angepasst. Aufrufe ohne eine Spezifizierung dieser führen zur Verwendung von Standardwerten. Durch systematische Änderungen soll für jeden Parameter das lokale Optimum gefunden werden. Im Folgenden sind die Parameter gezeigt, mit welchen die besten Ergebnisse erzielt werden konnten.

```
dtree = tree.DecisionTreeClassifier(
    max depth=9,
    min samples leaf=400,
    class weight={0: 1,1: weight}
)
```

Quelltext 4.9: Parameter des Entscheidungsbaum-Klassifikators aus der scikit-learn-Bibliothek

Die Bibliothek bietet zudem bereits eine Funktion, um den entstandenen Klassifikator grafisch darzustellen. In Abbildung 4.3 ist ein Ausschnitt des konstruierten Entscheidungsbaums abgebildet. Aufgrund der gegebenen Parameter endet die Unterteilung und der Baum bildet ein Blatt aus, sobald eine weitere Trennung zu weniger als 400 Samples in einem Blatt führen würde oder die maximale Tiefe von neun erreicht wurde. Dadurch klassifiziert der Baum die Trainingsdaten weniger genau, erhält aber eine stärkere Fähigkeit zur Generalisierung, sodass die allgemeine Performance steigt. Der konstruierte Baum besitzt insgesamt 233 Knoten und 116 Verzweigungen.

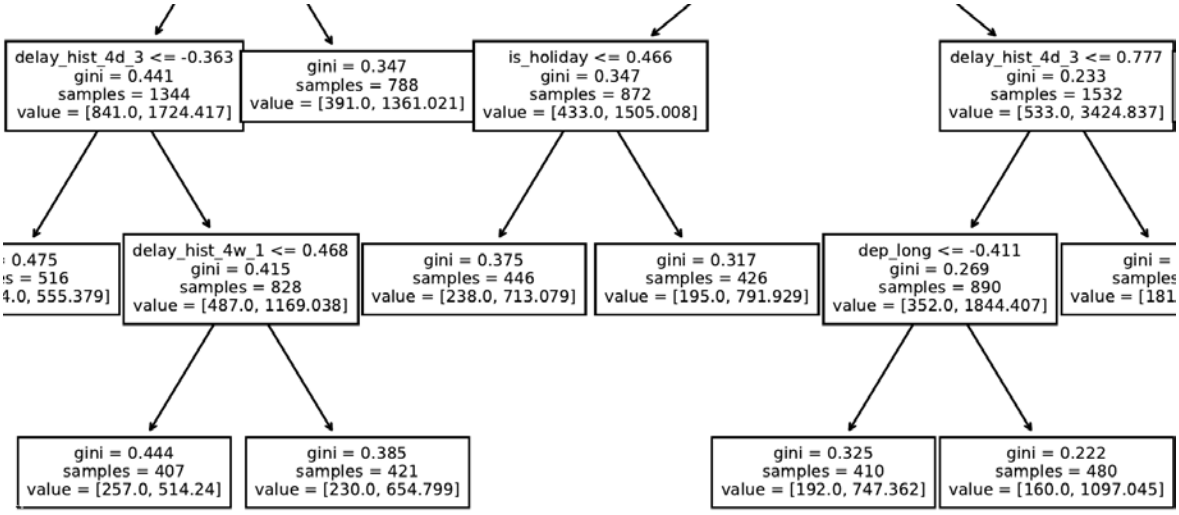


Abbildung 4.3: Ausschnitt des konstruierten Entscheidungsbaums, Bildquelle: eigene Darstellung

Im Folgenden ist die Performance dieses Entscheidungsbaum-Klassifikators anhand der in Kapitel 3.3 beschriebenen Metriken gezeigt.

$$Recall = 0.6926242 \qquad F1\ Score = 0.5381872 \qquad AUC = 0.805259$$

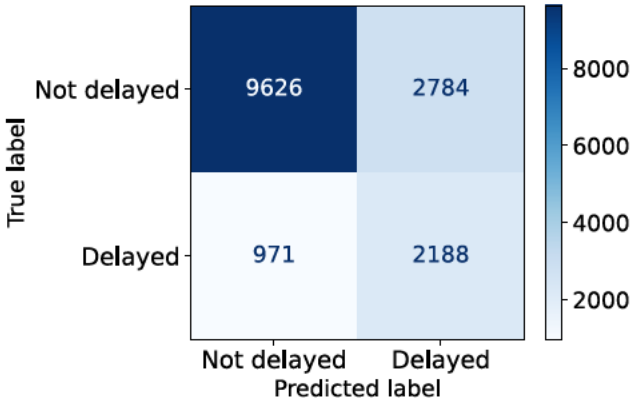


Abbildung 4.4: Konfusionsmatrix des Entscheidungsbaum-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

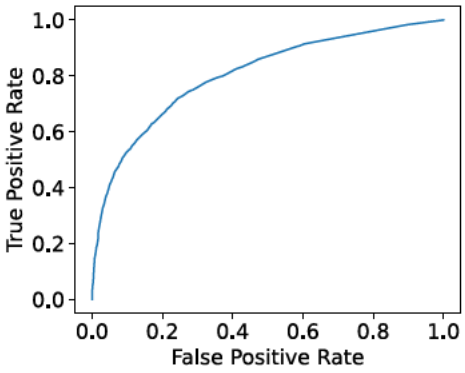


Abbildung 4.5: ROC-Chart des Entscheidungsbaum-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

4.5.3 Random Forest

Random Forest ist ein Algorithmus, der eine definierte Menge von Entscheidungsbäumen erstellt. Deshalb wird dieser auch als sogenannter Ensemble-Learning-Algorithmus bezeichnet. Ein Ensemble-Learning-Algorithmus kombiniert mehrere schwache Klassifikatoren zu einem starken Klassifikator. In diesem Fall stellen einzelne Entscheidungsbäume die schwachen Klassifikatoren dar [50]. Es gibt verschiedene Arten, die Ergebnisse bei Ensemble-Learning-Algorithmen zu kombinieren. Bei Random Forest kommt in der Regel das sogenannte Bagging zum Einsatz. Dies steht für Bootstrap Aggregation und meint eine gleichgewichtete Kombination der einzelnen Klassifikationen im Ensemble [51]. Bei Random Forest geschieht dies einfach mit dem Mittelwert aller Klassifikationen. Das Besondere neben dem Ensemble-Verfahren bei Random Forest ist außerdem, dass die einzelnen Bäume ihre zur Klassifikation verwendeten Merkmale zufällig auswählen. Dies führt zu einer geringeren Varianz und einer besseren Allgemeingültigkeit des Modells. Für die Anwendung eines solchen Klassifikators wird auf die Implementierung der scikit-learn-Bibliothek zurückgegriffen. Dazu wird die Klasse `ensemble.RandomForestClassifier` verwendet. Durch systematische Änderungen der Parameter sollen diese wieder dem lokalen Optimum angenähert werden. Im Folgenden sind die Parameter gezeigt, mit welchen die besten Ergebnisse erzielt werden konnten.

```
1  forest = RandomForestClassifier(
2      n estimators=1000,
3      max depth=12,
4      min samples leaf=100,
5      class weight={0: 1,1: weight},
6      n jobs=16
7  )
```

Quelltext 4.10: Parameter des Random Forest-Klassifikators aus der scikit-learn-Bibliothek

Im Folgenden ist die Performance dieses Random-Forest-Klassifikators anhand der in Kapitel 3.3 beschriebenen Metriken gezeigt.

$$Recall = 0.6862931 \qquad F1\ Score = 0.5477514 \qquad AUC = 0.8141633$$

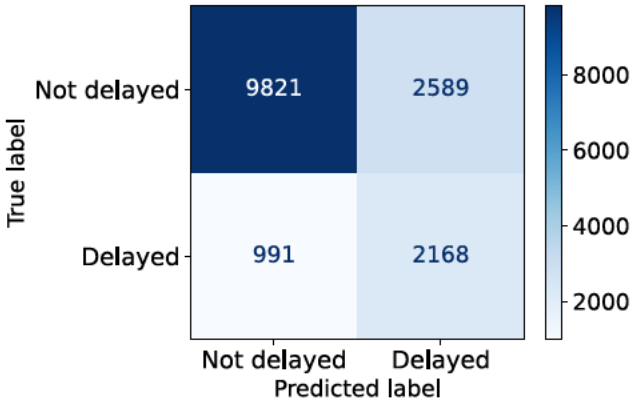


Abbildung 4.6: Konfusionsmatrix des Random-Forest-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

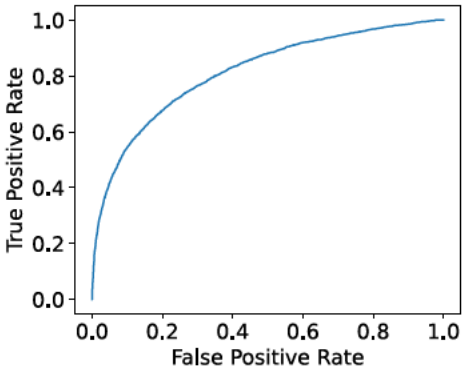


Abbildung 4.7: ROC-Chart des Random-Forest-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

4.5.4 CatBoost

CatBoost ist ein relativ neuer Algorithmus, welcher 2017 vom Unternehmen Yandex veröffentlicht wurde [52]. Es handelt sich dabei ebenfalls um einen Ensemble-Learning-Algorithmus, welcher das sogenannte Gradient-Boosting verwendet. Dabei werden schrittweise neue Bäume hinzugefügt, welche die Fehler des vorherigen Klassifikators minimieren sollen [53]. Dieser Prozess kann mehrere tausend Male wiederholt werden, um am Ende das Modell zu verwenden, welches entsprechend der verwendeten Verlustfunktion am besten abgeschnitten hat. Durch Parallelisierung kann dieser Prozess außerdem stark beschleunigt werden, wenn beispielsweise eine dedizierte Grafikkarte für den Trainingsprozess zur Verfügung steht [54]. Die Dauer für einen Aufruf der `fit()`-Funktion belief sich dadurch auch bei mehreren tausend Iterationen auf weniger als zehn Minuten.

CatBoost wird als gleichnamige Python-Bibliothek zur Verfügung gestellt. Eine Besonderheit ist, dass Merkmale über den Parameter `cat features` als kategorisches Merkmal gekennzeichnet werden können. Obwohl die Merkmale `dep` und `arr`, welche die Bahnhofsnummern beinhalten, auf den ersten Blick kategorischer Natur zu sein scheinen, schnitt das Modell besser ab, wenn diese nicht mit angegeben wurden. Bei verschiedenen Identifikationsnummern sollte theoretisch die zahlenmäßige Relation keine Rolle spielen, da es sich um eine reine Bezeichnung handelt. Tatsächlich ist aber, wie in Kapitel 4.4 gezeigt, eine Aussagekraft in Bezug auf das Verspätungsrisiko im Datensatz vorhanden. Die Ursache dafür liegt höchstwahrscheinlich im Vergabeverfahren dieser Bahnhofsnummern. Da diese nicht zufällig sind, lassen sich Zusammenhänge aus dem Zahlenwert ableiten. Teilweise ist eine alphabetische

Anordnung erkennbar, die sich aber nicht vollständig durchzieht. Dadurch liegen aber viele Bahnhöfe, welche beispielsweise mit „Berlin“ beginnen, in einem bestimmten Wertebereich, sodass hier ein entsprechender Informationsgehalt im Hinblick auf die Region vorhanden ist.

Zunächst wurden mehrere Versuche mit dem Parameter `iterations=500` durchgeführt, um schnell die Auswirkungen der Parameter beobachten zu können. Anschließend wurde dieser für die finale Modellerstellung und Auswertung auf 5000 erhöht.

```
cb = CatBoostClassifier(
    random seed=8926712,
    iterations=5000,
    leaf estimation iterations=20,
    depth=7,
    learning rate=0.01,
    cat features=['is holiday', 'long we'],
    class weights=[1,weight],
    custom loss='Recall',
    use best model=True,
    verbose=False,
    task type='GPU'
)
```

Quelltext 4.11: Parameter des CatBoost-Klassifikators

Im Folgenden ist die Performance dieses CatBoost-Klassifikators anhand der in Kapitel 3.3 beschriebenen Metriken gezeigt.

$$Recall = 0.6581197 \qquad F1\ Score = 0.5551402 \qquad AUC = 0.8145402$$

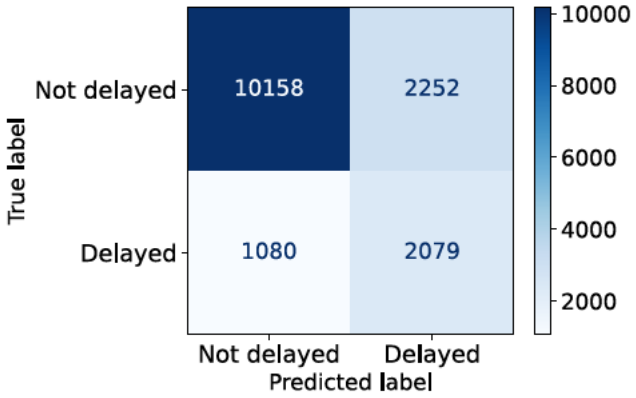


Abbildung 4.8: Konfusionsmatrix des CatBoost-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

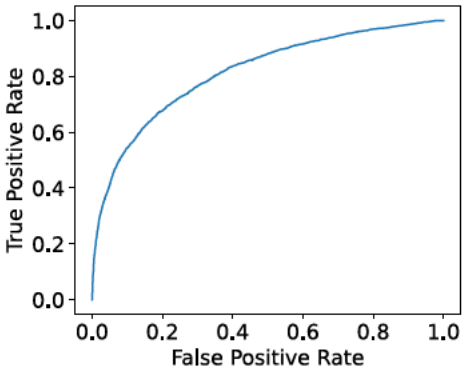


Abbildung 4.9: ROC Chart des CatBoost-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

4.5.5 Support Vector Machine

Eine Support Vector Machine (SVM) ist eine Methodik die sowohl zur Klassifikation als auch zur Regression verwendet werden kann. Aufgrund der günstigen Eigenschaften speziell bei kleineren Datenmengen, ist die Anwendung weit verbreitet [55]. Das Grundprinzip einer SVM basiert auf dem Finden einer Entscheidungsgrenze, welche die einzelnen Datenpunkte optimal aufteilt. Diese Grenze ist im zweidimensionalen Raum eine Gerade, die so verläuft, dass die Abstände zu den nächstgelegenen Datenpunkten, den sogenannten Support Vectors, maximal sind [56]. Zusätzlich wird dieses Prinzip durch eine sogenannte Kernelfunktion erweitert, um die Daten in einen höherdimensionalen Raum zu transformieren. Die Entscheidungsgrenze wird dann durch eine Hyperfläche dargestellt, was eine möglichst optimale Trennung erleichtert. Vorteilhaft beim Einsatz einer SVM ist auch die Resistenz gegen Overfitting und Ausreißer in den Daten. Allerdings erfordert die Wahl des Kernels und der Kostenparameter Erfahrung und Einfühlungsvermögen. Außerdem kann der Trainingsprozess bei einem größeren Datenumfang zeitaufwendiger sein [57]. Insbesondere spielt dabei die Anzahl der Merkmale eine Rolle. Für die Parametrisierung wurde die RBF-Kernelfunktion gewählt, da diese das beste Ergebnis lieferte.

```
1  svm = svm.SVC(
2      probability=True,
3      max iter=5000,
4      kernel='rbf',
5      class weight={0: 1,1: weight}
6  )
```

Quelltext 4.12: Parameter des SVM-Klassifikators

Im Folgenden ist die Performance dieses SVM-Klassifikators anhand der in Kapitel 3.3 beschriebenen Metriken gezeigt.

$$Recall = 0.5064894 \qquad F1\ Score = 0.5184705 \qquad AUC = 0.7700909$$

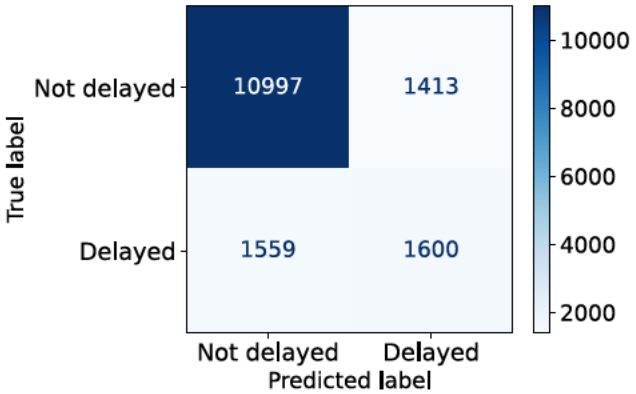


Abbildung 4.10: Konfusionsmatrix des SVM-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

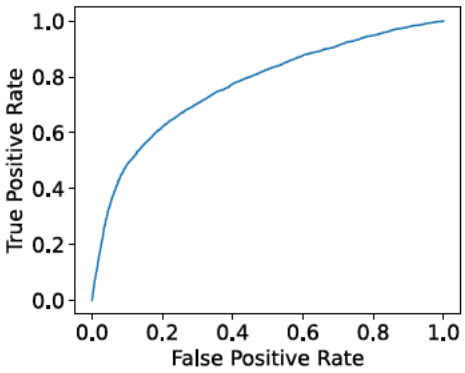


Abbildung 4.11: ROC Chart des SVM-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

4.5.6 Bernoulli Naive Bayes

Naive Bayes ist ein probabilistisches Verfahren zur Klassifikation, welches das Bayes Theorem verwendet, um die Wahrscheinlichkeit einer Klasse für jedes einzelne Merkmal zu bestimmen. Dabei werden die Merkmale komplett unabhängig voneinander betrachtet [58]. Generell gilt das Verfahren als ungeeignet für probabilistische Klassifikationen [59]. Bernoulli Naive Bayes erzielte jedoch in den Versuchen mit Lazy Predict gute Ergebnisse. Um dies zu ergründen, soll es auch in die Untersuchung mit einfließen und wird im Folgenden ebenfalls getestet. Bei der Form des Bernoulli Naive Bayes wird lediglich die bedingte Wahrscheinlichkeit dafür berechnet, ob ein Merkmal ausgeprägt ist oder nicht, sodass alle Merkmale letztlich in eine binäre Repräsentation umgewandelt werden müssen. In der Standardkonfiguration ist jedes Merkmal mit einem Wert größer null gleich eins, also ausgeprägt, wobei dieser Grenzwert beliebig wählbar ist. Im vorliegenden Datensatz ist daher der Großteil der Merkmale immer ausgeprägt, da beispielsweise die Anzahl der Reiseabschnitte, die Dauer der Reise oder die Koordinaten der Bahnhöfe ausschließlich einen Wert größer null haben. Daher scheint das Verfahren auf den ersten Blick ungeeignet. Außerdem wird davon ausgegangen, dass jedes Merkmal denselben Einfluss auf die Klassenzugehörigkeit hat [58]. Vorteilhaft bei der Verwendung von Bernoulli Naive Bayes ist die sehr einfache Anwendung und der geringe Trainingsaufwand. Das Verfahren ist auch bei einer großen Anzahl an Merkmalen im Datensatz sehr effizient. Naive Bayes ist zu dem robust gegenüber fehlenden oder ausreißenden Merkmalen. Aufgrund der starken Vereinfachung wird der Klassifikator aber in vielen Fällen durch Ensemble-Learning-Algorithmen geschlagen [46].

Für die in Kapitel 4.5.1 beobachteten Ergebnisse des Bernoulli Naive Bayes-Klassifikators, bei welchen die meisten Merkmale gar nicht berücksichtigt werden können, scheinen zwei Hauptursachen naheliegend. Wie bereits gezeigt, haben die Merkmale der historischen Verspätungen mit großem Abstand die größte Aussagekraft. Wenn eine Zugverbindung selten Verspätungen aufweist, sind auch die entsprechenden Merkmale nicht ausgeprägt und werden im Modell entsprechend interpretiert. Eine weitere Ursache könnte in der fehlenden Konfiguration anderer Klassifikatoren liegen. Naive Bayes-Verfahren besitzen keine Parameter, die einer speziellen Abstimmung auf das vorliegende Problem bedürfen, wie es beispielsweise bei SVM-Klassifikatoren der Fall ist. Andere Verfahren könnten an dieser Stelle noch Raum für Verbesserungen bieten, welcher bei Naive Bayes nicht vorhanden ist.

Für die Anwendung von Bernoulli Naive Bayes kann wieder auf eine Implementierung in der scikit-learn-Bibliothek zurückgegriffen werden. Dazu wird die Klasse `naive bayes.BernoulliNB` verwendet. Aufgrund der geringen Komplexität, gibt es auch keine in diesem Fall relevanten Konfigurationsmöglichkeiten, sodass der Aufruf letztlich mit den Voreinstellungen erfolgt.

Im Folgenden ist die Performance dieses Klassifikators anhand der in Kapitel 3.3 beschriebenen Metriken gezeigt.

$$Recall = 0.546692 \qquad F1\ Score = 0.5522865 \qquad AUC = 0.8037307$$

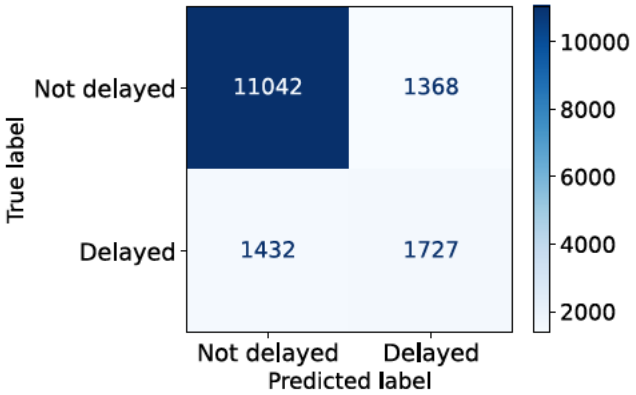


Abbildung 4.12: Konfusionsmatrix des Bernoulli Naive Bayes-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

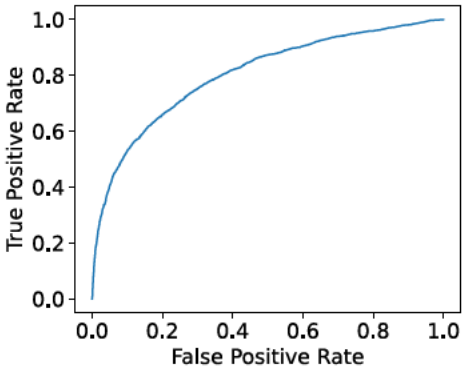


Abbildung 4.13: ROC Chart des Bernoulli Naive Bayes-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

4.6 Ergebnisanalyse und Vergleich mit Baseline-Modell

Es wurde ein Modell erstellt, welches aus den über 60 Minuten verspäteten Verbindungen in der Vergangenheit die Wahrscheinlichkeit für eine erneute Verspätung anteilig ermittelt. Wenn beispielsweise zwei von acht vergangenen Fahrten eine Verspätung aufweisen, beträgt die Wahrscheinlichkeit 25 % und die betreffende Verbindung würde entsprechend als nicht verspätet klassifiziert werden. Dieses Modell dient als sogenannte Baseline. Ein Baseline-Modell ist ein Modell, welches als Referenz und Ausgangsbasis für die Entwicklung weiterer Modelle dient. Es werden dazu ebenfalls die beschriebenen Metriken ermittelt und so eine Vergleichsmöglichkeit der Klassifikatoren geschaffen. Im Folgenden sind diese dargestellt.

$$Recall = 0.3782843 \qquad F1\ Score = 0.4849838 \qquad AUC = 0.7867563$$

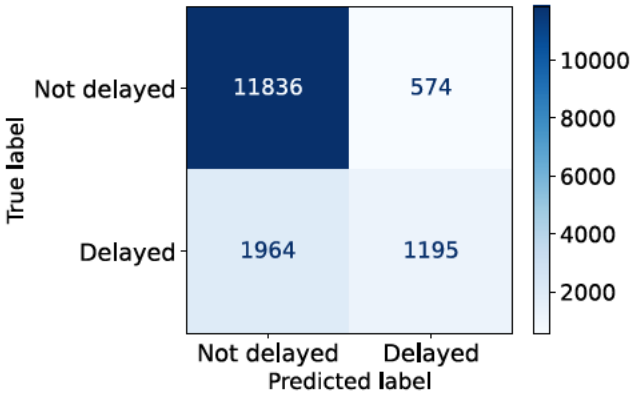


Abbildung 4.14: Konfusionsmatrix des Baseline-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

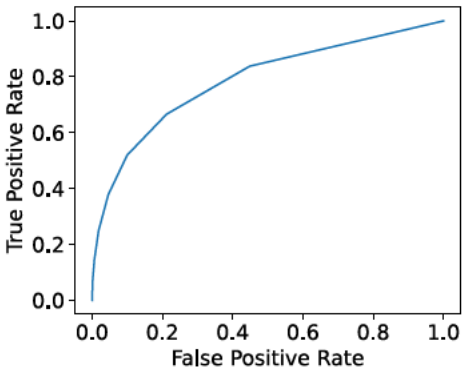


Abbildung 4.15: ROC Chart des Baseline-Klassifikators über die Evaluierungsdaten, Bildquelle: eigene Darstellung

Das Modell soll letztlich die Frage beantworten, ob ein tatsächlicher Mehrgewinn durch die Anwendung der Algorithmen und dem Hinzuziehen der ermittelten Merkmale erzielt wird. Wenn ein Modell nicht besser abschneidet als das Baseline-Modell, besteht kein Grund, es in der Praxis zu verwenden. Um nun eine relative Verbesserung oder Verschlechterung zu zeigen, wird die Veränderung des AUC- und des F1-Scores der einzelnen Modelle prozentual angegeben. Die finalen Ergebnisse sind aus der Tabelle 4.6 zu entnehmen.

Model	F1	Verbesserung
Catboost	0.5551402	14,47 %
Bernoulli Naive Bayes	0.5522865	13,88 %
Random Forest	0.5477514	12,94 %
Decision Tree	0.5381872	10,97 %
SVM	0.5184705	6,9 %

Tabelle 4.6: Vergleich des F1-Scores der Modelle gegenüber dem Baseline Model, absteigend sortiert

Model	AUC	Verbesserung
Catboost	0.8145402	3,53 %
Random Forest	0.8141633	3,48 %
Decision Tree	0.805259	2,35 %
Bernoulli Naive Bayes	0.8037307	2,16 %
SVM	0.7700909	-2,12 %

Tabelle 4.7: Vergleich des AUC-Scores der Modelle gegenüber dem Baseline Model, absteigend sortiert

Wie sich in der tabellarischen Auflistung zeigt, ist tatsächlich ein Mehrgewinn nachzuweisen. Aufgrund der Ausführungen in Kapitel 3.3 ist für die finale Auswertung der F1-Score entscheidend. Das Catboost-Modell erzielt in beiden Bewertungen das beste Ergebnis, mit einer

Verbesserung des F1-Scores von 14,5 %. In der Praxis bedeutet die deutliche Verbesserung des F1-Scores, dass durch die Modelle nicht nur mehr verspätete Zugfahrten erkannt werden, sondern auch weniger pünktliche Fahrten fälschlicherweise als verspätet prognostiziert werden. Der F1-Score des CatBoost-Modells setzt sich aus der Precision von 0.48 und dem Recall von 0.6581 zusammen. Das bedeutet einerseits, dass nur knapp die Hälfte aller als verspätet klassifizierten Verbindungen, auch wirklich zu spät war. Andererseits können von drei tatsächlich verspäteten Zügen, zwei bereits vorher erkannt werden.

Im Hinblick auf die Risikoberechnung für eine Versicherung bedeutet dies, dass aufgrund der eher geringen Precision, die Prognosen des Modells tendenziell pessimistisch ausfallen. Jeder zweite Zug, der eine Verspätungswahrscheinlichkeit von über 50 % hatte, war letztlich pünktlich und hätte potenziell eine attraktive Versicherungspolice ermöglicht. Wird eine Verbindung zu riskant eingestuft, sinkt die Auszahlung bis schließlich keine Police mehr angeboten werden kann, wenn die Wirtschaftlichkeit gewährleistet bleiben soll. Es werden demnach weniger Versicherungspolicen verkauft. Der Recall sagt in diesem Zusammenhang aus, dass für zwei Drittel der verspäteten Verbindungen bereits ein Risiko von über 50 % prognostiziert wurde. Ein Angebot einer Police ist bei diesen Verbindungen unwahrscheinlich. Dies ist wichtig, um zu verhindern, Policen für praktisch unversicherbare Risiken anzubieten. Das Modell kann demnach diese hochriskanten Verbindungen in den meisten Fällen identifizieren. Aufgrund dieser Auswirkungen von Precision und Recall ist ein erhöhter F1-Score als klare wirtschaftliche Verbesserung für eine mögliche Versicherung zu interpretieren. In 81,4 % aller Fälle ist das Modell in der Lage, verspätete Verbindungen von pünktlichen zu unterscheiden. Das sagt der AUC-Score über die Gesamtleistung des Klassifikators aus.

Eine Verbesserung gegenüber dem Baselinemodell ist bei allen Algorithmen erkennbar, außer in der AUC-Bewertung des SVM-Modells. Die Unterschiede lassen sich in erster Linie durch die Eignung eines Algorithmus für eine bestimmte Problemstellung erklären. Unter Umständen sind im SVM-Modell weitere Feinabstimmungen in der Vorverarbeitung und der Kernelfunktion erforderlich. Weiterhin muss bei der Interpretation der Ergbenisse das zu Grunde liegende Baselinemodell, welches hier als Referenz dient, berücksichtigt werden. Würde als Baselinemodell ein sogenannter Dummy-Klassifikator verwendet werden, welcher immer als „pünktlich“ klassifiziert, so wären entsprechend größere Unterschiede erkennbar. In diesem Fall wurde aber als Baseline ein Modell gewählt, welches bereits eine solide Grundlage zur Vorhersage bietet, um eine möglichst realistische Bewertung, insbesondere durch das Hinzuziehen weiterer Merkmale, zu erzielen. Von den getesteten Algorithmen haben alle Entscheidungsbaum-basierten Verfahren gute Ergebnisse geliefert.

4.7 Deployment des Modells

Das Catboost-Modell soll aufgrund der besten Ergebnisse nun in die Praxis überführt werden. Dabei soll die Verspätungsprognose als eigenständige Komponente in eine größere Architektur eingebettet werden. Dazu muss das finale Modell aus der jupyter-Umgebung in eine

Bereitstellungsumgebung übertragen werden. Ein Ansatz wäre der Export zum Beispiel in ein Dateiformat und anschließender Import in die Bereitstellungsumgebung. Dazu können Standards wie die Predictive Model Markup Language (PMML) verwendet werden. Bei der Verwendung von scikit-learn bietet sich auch die mitgelieferte Möglichkeit `pickle` zur Serialisierung von trainierten Modellen an [60]. Die Catboost-Bibliothek bietet ebenfalls eigene Funktionen zum Speichern und Laden von Modellen an, auf die hier zurückgegriffen wurde.

Der wichtigste Teil des Deployments ist die Implementierung eines Service, welcher die Featureaggregation und Vorverarbeitung übernimmt. Nur so kann das Modell überhaupt für Vorhersagen verwendet werden. Die Eingabeparameter sind durch die Definition einer Zugverbindung bestimmt, welche mindestens aus Abfahrtsbahnhof, Ankunftsbahnhof und einem Zeitpunkt besteht. Aus diesen Parametern müssen neben der genauen Verbindung, alle für das Modell erforderlichen Merkmale in Echtzeit aggregiert werden. Hier ist es ebenfalls vorteilhaft, diesen Service in Python zu implementieren, da viele der bereits implementierten Funktionen übernommen werden können. Während der Entwicklung stellte sich die Aggregation der Wetterdaten als problematisch heraus. Entsprechende Prognosen sind entweder nicht für alle in der Untersuchung verwendeten Merkmale verfügbar oder nur mit erheblich höherem Aufwand von verschiedenen Anbietern zu aggregieren. Aufgrund dieser Schwierigkeiten, werden die Wettermerkmale für die Echtzeitprognose zunächst außen vor gelassen. Alle anderen Merkmale konnten weitestgehend analog zur Aggregation in Kapitel 4.1 umgesetzt werden.

Letztlich wird der Service unter Verwendung des FastAPI-Frameworks und dem uvicorn-Webserver implementiert und in einem Docker-Container bereitgestellt [61]. Die Basis für diese Architektur stellt das Framework `katana` zur Verfügung [62]. Der Einsatz eines solchen Aufbaus bietet mehrere Vorteile. Beispielsweise können verschiedene Modelle gleichzeitig bereitgestellt werden. Wenn diese unterschiedliche Parameter haben, überschneiden sich immer bestimmte Teile der Aggregationslogik. Die Streckendistanz muss beispielsweise immer ermittelt werden, egal welches Modell und welcher Datensatz zum Einsatz kommt. Durch die Verwendung einer Klasse `aggregator.py` können derartige Funktionen an einer zentralen Stelle implementiert werden. Weiterhin ist eine einfache Dokumentation der Schnittstelle, insbesondere der Eingabeparameter, durch das Swagger-Interface von FastAPI möglich und die API kann direkt über die Weboberfläche getestet werden. Da hier der Kern dieser Arbeit liegt, ist die Dokumentation und Vorführmöglichkeit der Prediction-API entsprechend relevant. Zunächst wurde die Aggregation (`/aggregate`) und die Vorhersage (`/predict`) in zwei getrennten Abfragen umgesetzt. Dies hat den Vorteil, dass die verwendeten Merkmale vor der Vorhersage noch einmal angepasst oder kontrolliert werden könnten. Schließlich wurde aber in einer zweiten Version unter dem Endpunkt `/predict` beides vereint. So kann bereits mit minimalen Eingaben direkt die Verspätungswahrscheinlichkeit abgefragt werden. In Abbildung 4.16 ist das Interface des fertigen Services gezeigt.

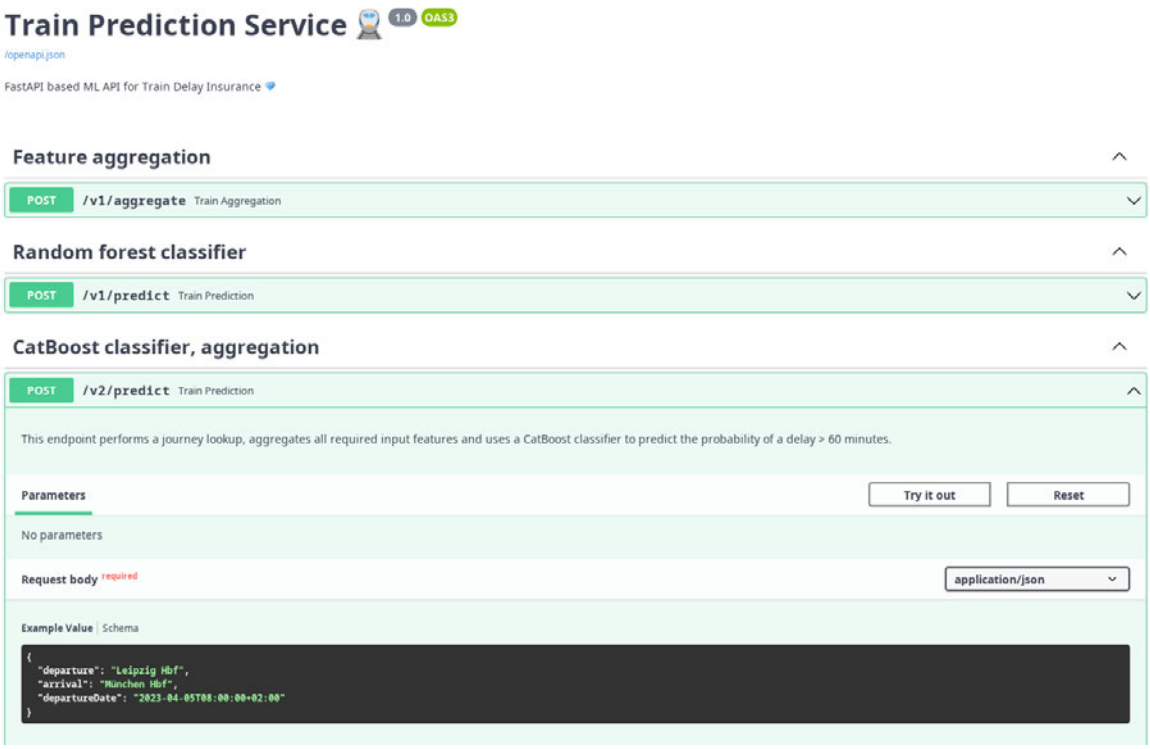


Abbildung 4.16: Swagger-Interface des Prediction-Service, Bildquelle: eigene Darstellung

4.8 Integration in einen Smart Contract

Um die Ergebnisse des Modells in einem Smart Contract zu verarbeiten, muss eine Art der eingangs beschriebenen Schnittstelle (Orakel) geschaffen werden. Dazu wird das Chainlink Framework verwendet [63]. Es existieren noch weitere Frameworks, die eine ähnliche Funktionalität bieten. Da Chainlink aber die am weitesten verbreitete Lösung ist und eine umfangreiche Dokumentation bietet, wurde es für dieses Projekt gewählt. Das Chainlink Framework realisiert den grundlegenden Prozess, um auf ein bestimmtes im Smart Contract ausgelöstes Event zu reagieren und eine definierte Callback-Funktion aufzurufen. Zur Konfiguration und Statusabfrage kann eine Weboberfläche verwendet werden, was die Handhabung erleichtert. Eine bestimmte Art der Datenanfrage wird in Chainlink als sogenannter Job definiert. Der Job bekommt in diesem Fall die Parameter für eine Zugverbindung und soll das Risiko einer Verspätung zurückliefern. Er muss also den in Kapitel 4.7 beschriebenen Endpunkt aufrufen. Dafür wird ein Chainlink Node (Knoten) benötigt, der diesen Job kennt und bearbeiten kann. Es ist denkbar, den Job bei bestehenden Nodes zu registrieren oder eine Eigene zu betreiben. Hierbei ist auf das gewünschte Netzwerk zu achten, in welchem später die Anwendung ausgeführt werden soll. Das administrative Einrichten eines solchen Nodes, beispielsweise als Docker-basierter Service auf einem Cloudserver, wurde nicht vom Verfasser vorgenommen und ist somit nicht Teil dieser Arbeit.

Der Job wird als sogenannter „Direct Request“ in einer Chainlink-Jobspezifikation definiert, welche das TOML-Format verwendet. Die finale Jobspezifikation ist in Anhang A.6 zu finden. Um den Job nun in einem Smart Contract zu verwenden, wird er über die Weboberfläche auf dem Chainlink-Node registriert.

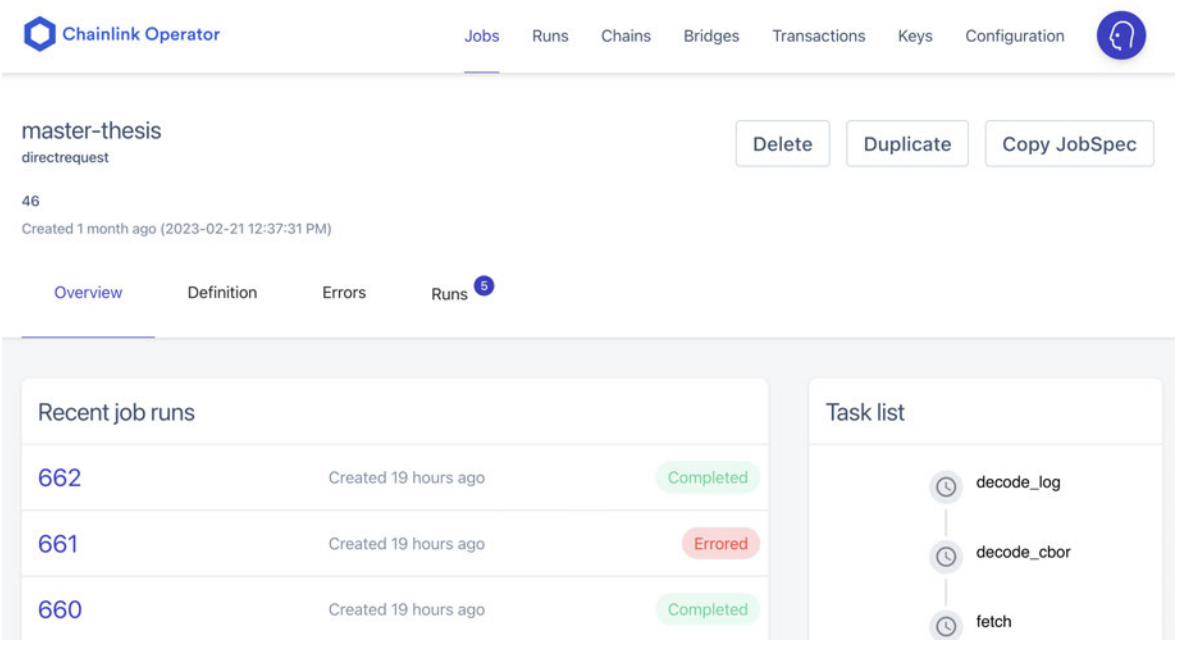


Abbildung 4.17: Weboberfläche des Chainlink-Nodes, Bildquelle: eigene Darstellung

Ein Smart Contract kann dann die ChainlinkClient-Bibliothek verwenden, um mit dem Operator Contract des Chainlink-Nodes zu interagieren. Dieser Operator Contract wird vom Betreiber der Node deployed und stellt die Blockchainseite des Orakels dar (siehe Abbildung 4.18). In der Funktion `requestPrediction()` wird eine Anfrage an den Operator Contract gesendet. Dieser löst ein Event aus, welches dem Chainlink-Node die Parameter für den Job übergibt. Daraufhin kann der Chainlink-Node mit einem entsprechenden Aufruf an die API die angefragten Informationen beschaffen. Die Funktion ist in Quelltext 4.13 gezeigt.

```
1  function requestPrediction(
2      string memory  departure,
3      string memory  arrival,
4      string memory  departureDate
5  ) public onlyOwner {
6      Chainlink.Request memory req = buildChainlinkRequest(
7          stringToBytes32(jobId),         // job id
8          address(this),                  // contract address
9          this.fulfillPrediction.selector // selector of the callback function
10     );
11
12     req.add("departure",  departure);         // "Leipzig Hbf"
13     req.add("arrival",  arrival);             // "München Hbf"
14     req.add("departureDate",  departureDate); // "2023-01-28T09:12:00"
15
```

```
16      sendChainlinkRequestTo(oracle, req, ORACLE PAYMENT);
17  }
```

Quelltext 4.13: Request-Funktion des Prediction Contracts (ChainlinkClient-Bibliothek)

Die Ergebnisse werden über die Callback-Funktion `fulfillPrediction()` zurückgeliefert und könnten im nächsten Schritt zur Preisberechnung einer Versicherungspolice verwendet werden. In diesem Fall wird das Ergebnis einfach in die Variable `prediction` geschrieben. In Quelltext 4.14 ist die Callback-Funktion gezeigt. Der vollständige Smart Contract, mit welchem das System getestet werden kann, ist in Anhang A.7 zu finden.

```
1  function fulfillPrediction(
2      bytes32  requestId,
3      uint256  prediction
4  ) public recordChainlinkFulfillment( requestId) {
5      prediction =  prediction;
6  }
```

Quelltext 4.14: Callback-Funktion für die Verspätungswahrscheinlichkeit des Prediction Contracts

In Abbildung 4.18 sind die einzelnen Komponenten und der Informationsfluss dargestellt. Die Prediction-API unten rechts ist die in Kapitel 4.7 beschriebene FastAPI-Komponente. Hier wird die eigentliche Vorhersage mit Hilfe des entwickelten CatBoost-Modells durchgeführt. Aufgerufen wird diese durch den Chainlink-Node, sobald auf der Blockchain das entsprechende Event ausgelöst wurde. Diese Verbindung ist bei EVM-basierten Blockchains durch eine JSON-RPC-Schnittstelle realisiert, mit welcher zum Beispiel auch aufgetretene Events in einer Transaktion abrufbar sind. Dementsprechend muss der Chainlink-Node, analog zu einer Wallet, einen solchen JSON-RPC-Endpunkt kennen, da dieser selbst keinen Full Node betreibt. Erkennbar ist außerdem, dass das Event nicht direkt vom Konsumenten des Orakels, also dem Prediction-Contract, ausgeht. Die Kommunikation ist an dieser Stelle weitestgehend durch das Chainlink-Framework abstrahiert. Theoretisch befindet sich noch ein weiterer Smart Contract zwischen Konsument und Operator, welcher die Vergütungsfunktionalitäten von Chainlink abbildet [64]. Dieser spielt aber in der gezeigten Implementierung mit einem zentralen Orakel keine Rolle, da hier nur eine Partei am Prozess beteiligt ist.

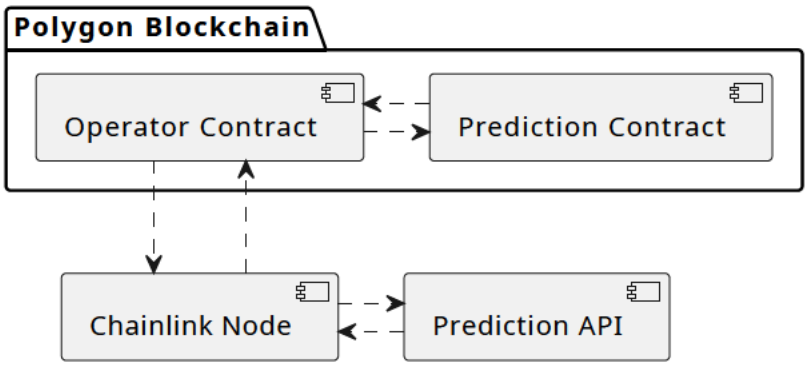


Abbildung 4.18: Komponentenübersicht des Basic Request Models mittels Chainlink (LINK Token nicht dargestellt), Bildquelle: eigene Darstellung

Um den kompletten Prozess zu demonstrieren, eignet sich die Verwendung der Remix IDE [65]. Hierfür wird der Smart Contract `Prediction.sol` in die IDE eingefügt und kompiliert. Nun kennt Remix das ABI und kann mit bereits deployten Versionen dieses Contracts auf der Blockchain interagieren. Der Contract ist auf dem Polygon-Testnetzwerk unter der Adresse `0x164679066b30508042Bb1835E89E1a9aF55e18b8` verfügbar. Der Browser sollte daher eine Ethereum-Erweiterung installiert haben, um mit dem Polygon-Netzwerk zu interagieren. Außerdem ist eine geringe Menge Testwährung erforderlich, um die Transaktionskosten zu bezahlen.

In der Oberfläche von Remix kann nun auf „Injected Provider“ gewechselt werden, um mit dem Contract zu interagieren und so die Funktion `requestPrediction()` aufrufen zu können. Dabei müssen die Parameter für die Zugverbindung angegeben werden. Um beispielsweise für die Verbindung von Berlin nach Bonn am 31.03.2023 um 9:00 Uhr das Verspätungsrisiko abzufragen, sind die Parameter `Berlin Hbf, Bonn Hbf, 2023-03-31T09:00:00` zu übergeben. Kleine Tippfehler oder abweichende Stationsnamen werden während der Aggregation durch den Abruf der Stationsnummer in der Funktion `fetch ibnr` korrigiert. Es könnte daher auch „Bonn Hauptbahnhof“ übergeben werden. Das Ergebnis wird schließlich in der Variable `prediction` gespeichert und kann nach einigen Sekunden ebenfalls über Remix mit einem Call, also einem reinen Lesezugriff auf die Blockchain, abgerufen werden. In Abbildung 4.19 ist die Remix-Umgebung mit abgefragter Verspätungswahrscheinlichkeit gezeigt.

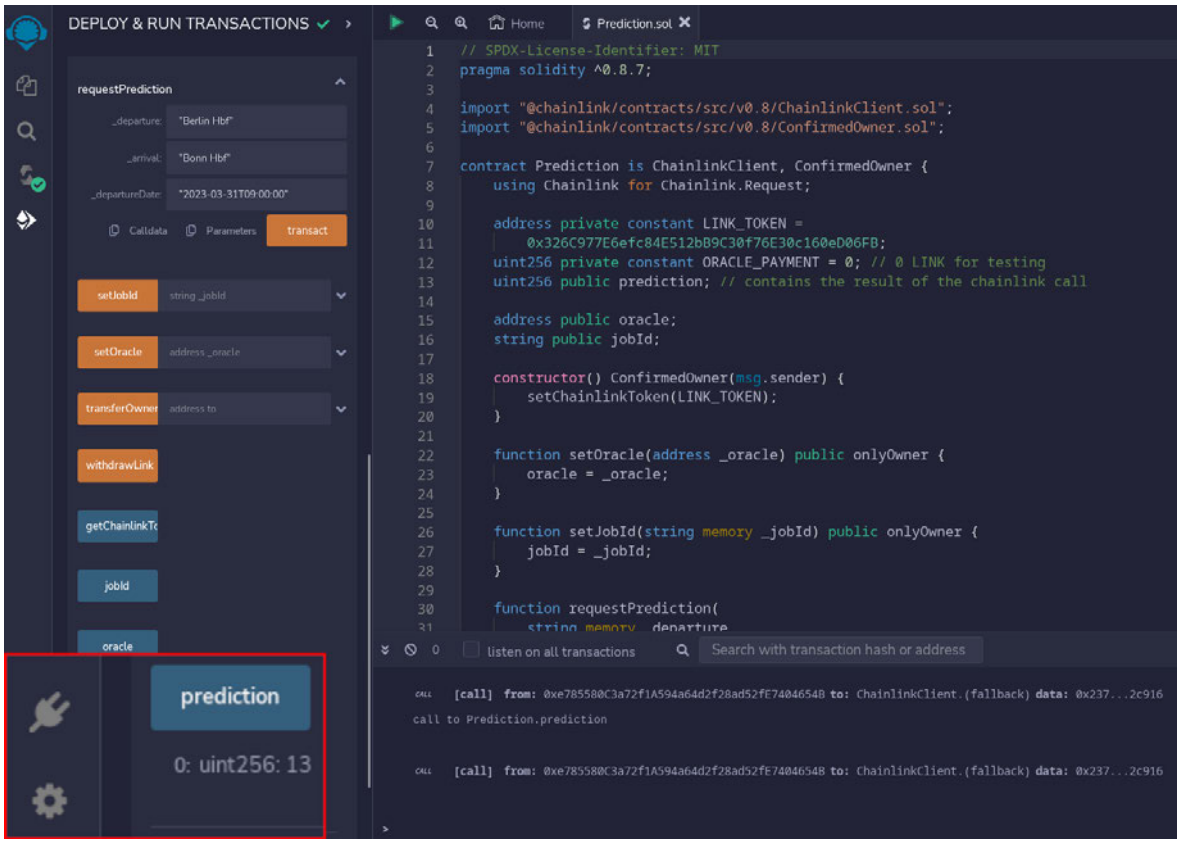


Abbildung 4.19: Remix IDE mit Prediction Contract und abgerufener Verspätungswahrscheinlichkeit von 13 % (unten links), Bildquelle: eigene Darstellung

5 Fazit und Ausblick

5.1 Fazit

Das Ergebnis des Projekts ist ein Modell, welches zwei aus drei verspäteten Zugreisen, korrekt vorhersehen kann. Die besten Ergebnisse konnten mit dem CatBoost-Algorithmus erzielt werden, sodass dieser letztlich für die Umsetzung der Echtzeitprognose ausgewählt wurde. Die wichtigsten Einflussfaktoren sind neben der Verspätung der gleichen Verbindung an vergangenen Tagen, insbesondere vor einem Tag und vor exakt einer Woche, vorliegende Feiertage, die Anzahl der Umstiege, der Längengrad des Abfahrtsbahnhof, die Streckendistanz und die Tageszeit. Bei der Umsetzung der Echtzeitprognose wurde das Wetter aufgrund des Aufwandes nicht mit einbezogen, wie es ursprünglich geplant war. Hier wäre die Anbindung weiterer Datenquellen notwendig, speziell für robuste Prognosedaten für einen Zeithorizont von bis zu 10 Tagen für Taupunkt und Temperatur. Der Einfluss dieser Wetterparameter konnte bestätigt werden. Insgesamt konnte eine Verbesserung durch die Anwendung des maschinellen Lernens gezeigt werden. Während der F1-Score für die Vorhersage der Verspätung im Vergleich zur bloßen Auswertung der Historie um 14 % gesteigert wurde, ist der AUC-Score um 3,5 % besser ausgefallen. Es konnte weiterhin eine Komponente zur Echtzeitprognose mit dem entwickelten Modell umgesetzt und demonstriert werden. Der Zeithorizont der Vorhersage ist auf Kosten der Genauigkeit beliebig erweiterbar. Die Echtzeit-Featureaggregation ist in der finalen Version ohne Wettermerkmale von zwei Schnittstellen abhängig. Diese sind für die historischen Verspätungsdaten und für die initiale Ermittelung der Verbindung gemäß Fahrplan notwendig. Alle weiteren Features werden auf Basis dieser Daten innerhalb weniger Sekunden ermittelt. Auch die Anbindung an den Smart Contract konnte wie in den Anforderungen beschrieben, umgesetzt werden. Insofern ist die technische Basis für das assoziierte Forschungsprojekt wie angedacht geschaffen worden.

5.2 Ausblick

Auch wenn die formulierten Anforderungen erfüllt wurden, gibt es eine ganze Reihe von Möglichkeiten, das Modell weiter zu verbessern und zu erweitern. Einige dieser Punkte sind entweder nicht oder nur teilweise im Rahmen dieser Arbeit abgedeckt worden. Das betrifft zunächst die Möglichkeit, das Modell um zusätzliche Features zu erweitern. Weitere Überlegungen könnten hier sein, auch zugbezogene Informationen, wie den Betreiber oder die Zuglänge mit Hilfe zusätzlicher Vorverarbeitung in die Vorhersage mit einzubeziehen. Mögliche Ansätze dafür wären die Durchführung einer Vor-Klassifikation oder der Einbezug besonderer Werte, wie es beispielsweise bei der kürzesten Umstiegszeit als Minimum von möglicherweise weiteren vorhandenen Umstiegen umgesetzt wurde.

Hinsichtlich der Metriken sollte außerdem das Aufstellen einer Kostenmatrix in Betracht gezogen werden, wenn eine Bezifferung der Kosten bei der anstehenden Verwertung des Modells zur Berechnung von Versicherungspolicen möglich ist. Mit dieser Metrik könnten die Modelle hinsichtlich ihrer wirtschaftlichen Profitabilität bewertet werden.

In Bezug auf die einzelnen Modelle wären ebenfalls Optimierungen denkbar, beispielsweise mit der Durchführung eines sogenannten Hyperparameter-Tunings, um kleine Verbesserungen in der Leistung zu erzielen. Die Hyperparameter umfassen Parameter eines Modells, die durch Ausprobieren verschiedener Kombinationen optimiert werden und so zu besseren Ergebnissen führen.

In der aktuellen Echtzeitprognose werden keine Streiks berücksichtigt. Hier wäre eine Blockierung des Abfahrtsdatums für einen angekündigten Streiktermin sinnvoll. Weiterhin könnte die Echtzeitprognose robuster gegenüber fehlenden und inkonsistenten Daten gemacht werden. Der Einsatz von Verfahren zur Ersetzung fehlender Daten durch Füllwerte ist hier denkbar und wurde in der dargestellten Komponente nicht umgesetzt.

Auch die Datenbasis bietet Raum für Verbesserungen. Das Einbeziehen der Wetterdaten von Dezember 2022 wäre ein sinnvoller nächster Schritt. Diese konnten aufgrund mangelnder Verfügbarkeit nicht mit in die Modellentwicklung einfließen. Trotz tiefgreifender Überlegungen für die Aggregation realistischer Zugverbindungen und mehrerer Iterationen der Erhebung könnte ein Datensatz, welcher auf tatsächlich verkauften Tickets beruht, eine weitere Verbesserung darstellen. Die Auswahl der Bahnhöfe konnte lediglich an das Passagieraufkommen angepasst werden. Die entstandenen Verbindungen bieten daher noch Raum für eine umfassendere Validierung hinsichtlich der Repräsentanz tatsächlich angetretener Bahnreisen, beispielsweise durch Umfragen. Wäre das Modell bei einem Bahnunternehmen in Betrieb, könnte die Datenbasis durch intern verfügbare Statistiken dahingehend optimiert werden.

Das entwickelte Modell hat gezeigt, dass es möglich ist, basierend auf verschiedenen Merkmalen von Zugverbindungen eine Vorhersage über das Verspätungsrisiko zu treffen und, dass die Anwendung von maschinellem Lernen sich für diese Aufgabe als sinnvoll erweist. Das Arbeitsergebnis wird im Rahmen des Forschungsprojekts Community Supported Insurance weiter verwertet. Mit der Anbindung an einen Smart Contract soll eine Möglichkeit entstehen, eine Versicherungspolice vollständig transparent berechnen zu können. Weiterhin kann das Prognosemodell auch isoliert der Gemeinschaft zur Verfügung gestellt werden. Denkbar ist hier eine schmale Webanwendung, welche die entwickelte Prognoseschnittstelle abstrahiert und eine einfache Nutzung beispielsweise für Bahnreisende oder Interessenten aus der Wirtschaft ermöglicht.

Anhang A: Quelltexte

A.1 journey_helper.py

```
###
# this script uses the dumped station data (stations.json) from a public
# transport endpoint to generate a set of journeys

import pandas as pd
import random, datetime

# returns typical (weighted) departure times
#   weights are based on typical train utilization
#   https://www.welt.de/wirtschaft/article106381146/Mit-diesen-Tricks-umgehen-Sie-rappelvolle-Zuege.html
def generate departure time():
    hours = list(range(0, 24))
    hour weights = [0, 0, 0, 0, 1, 4, 6, 6, 6, 6, 5, 4, 4, 4, 4, 4, 5, 6, 4, 2, 1, 1, 1, 0]
    dep hour = random.choices(hours, weights=hour weights, k=1)[0]
    dep minute = random.randrange(0, 60)
    return datetime.time(hour=dep hour, minute=dep minute)

# load raw station data
stations = pd.read json('preparation/stations.json', orient="index")
stations.set index('id', inplace=True)

# invert category to use it as weight
all stations = stations[['name', 'category']].copy()
all stations['weights'] = 8 - all stations['category']

# setup weights (customizable)
all stations.loc[all stations['weights'] == 7, 'weights'] = 700
all stations.loc[all stations['weights'] == 6, 'weights'] = 200
all stations.loc[all stations['weights'] == 5, 'weights'] = 20
all stations.loc[all stations['weights'] == 4, 'weights'] = 10
all stations.loc[all stations['weights'] == 3, 'weights'] = 10
all stations.loc[all stations['weights'] == 2, 'weights'] = 5
all stations.loc[all stations['weights'] == 1, 'weights'] = 5

# order by weight and delete duplicates
all stations.sort values(by='weights', ascending=False, inplace=True)
all stations.drop duplicates(inplace=True)

# generate pairs weighted by category of stations
print('generating 500 journeys with custom weights')
departures = all stations.sample(500, replace=True, weights='weights')
arrivals = all stations.sample(500, replace=True, weights='weights')
departures['weights'] = departures['weights'].astype('int')
```

```
arrivals['weights'] = arrivals['weights'].astype('int')

# reset index and add suitable column names
departures.reset index(inplace=True)
arrivals.reset index(inplace=True)
departures.rename(columns={'id': 'departure id', 'name': 'departure name', 'category': '
    departure category'}, inplace=True)
arrivals.rename(columns={'id': 'arrival id', 'name': 'arrival name', 'category': '
    arrival category'}, inplace=True)

# merge departures and arrivals
journeys = pd.concat([departures, arrivals], axis=1)
journeys.drop(['weights'], axis=1, inplace=True)

# delete duplicates
journeys.set index(['departure id', 'arrival id'], inplace=True)
journeys.drop duplicates(inplace=True)

# add a departure time
journeys['departure time'] = journeys.apply(lambda x: generate departure time(), axis=1)

#write to file
print('writing to preparation/journeys.csv')
journeys.to csv('preparation/journeys.csv')
print('done')
```

Quelltext A.1: Generieren von 1000 Verbindungen mit gewichtetem Sampling

A.2 journey_builder.py

```
###
# This script is used to retrieve journeys from public timetable endpoints.
# It needs START DATE and END DATE to be set below.

import random, math, json, requests, sys
from datetime import datetime, timedelta, date, time
from tqdm import tqdm
import pandas as pd
import multiprocessing as mp

PUBLIC TRANSPORT = '' # public transport endpoint
JOURNEYS CSV= 'preparation/journeys.csv'
WORKER COUNT = 8

START DATE = date(2022, 1, 13)
END DATE = date(2022, 1, 14)

# filter unwanted walking legs
def filter walking(l):
    if 'walking' in l and l['walking'] == True:
        return False
    else:
```

```
23          return True
24
25  # searches for a journey on a public transport endpoint and returns the legs (first
        result)
26  def fetch journey(departure, arrival, dep time):
27      request = PUBLIC TRANSPORT + "/journeys?from=" + departure + "&to=" + arrival + "&
            departure=" + dep time + '&results=3&bus=false&subway=false&tram=false&taxi=
            false&ferry=false'
28      #print('request: {}'.format(request))
29
30      # retrieve journey
31      req = requests.get(request)
32
33      if req.status code == 429:
34          print('{}: waiting 10 seconds due to status code 429'.format(mp.current process
                ().name))
35          time.sleep(10)
36          return 1
37      elif req.status code == 502:
38          #print('{}: got status code 502'.format(mp.current process().name))
39          response = req.json()
40          return 2
41      elif req.status code != 200:
42          return 2
43
44      response = req.json()
45
46      if 'journeys' in response:
47          #print('received journey')
48          journeys = response['journeys']
49          flegs = filter(filter walking, journeys[0]['legs']) # filter real legs
50          legs = list(flegs)
51
52          for leg in legs:
53              if 'productName' in leg['line'].keys() and leg['line']['productName'] == '
                    Bus':
54                  #print('discarding due to bus legs...')
55                  return 2
56
57          return legs
58      else:
59          return 2
60
61  def fetch journey wrapper(dep, arr, d):
62      while True:
63      # retrieve journey from public transport endpoint
64          legs = fetch journey(str(dep), str(arr), str(d))
65          if legs == 1:
66              continue
67          elif legs == 2:
68              return 9999
69              break
70          else:
```

```
71              break
72      return legs
73
74  def aggregate(key, values):
75      j eval = eval(key)
76      dep = j eval[0]
77      arr = j eval[1]
78
79      global current date
80
81      t = time.fromisoformat(values['departure time'])
82      d = datetime(current date.year, current date.month, current date.day, t.hour, t.
            minute)
83
84      journey = fetch journey wrapper(dep, arr, d)
85
86      return (dep, arr, str(d), journey)
87
88  # main function
89  if   name   == '  main  ':
90      print('importing journey data')
91      journeys df = pd.read csv(JOURNEYS CSV)
92      journeys df.set index(['departure id', 'arrival id'], inplace=True)
93      current date = START DATE
94      print('starting up...')
95
96      json data = journeys df.to json(orient='index')
97      journeys = json.loads(json data)
98      print(len(journeys))
99
100     # loop over days
101     days done = 0
102     chunks = (END DATE - START DATE).days
103     while days done < chunks:
104         print('starting chunk {}/{} ({})'.format(days done+1, chunks, current date))
105
106         pbar = tqdm(total=len(journeys))
107         result = {}
108
109         # safe result and update progress bar
110         def update(*a):
111             global result
112             result[str(a[0][0:3])] = a[0][3]
113             pbar.update()
114
115         pool = mp.Pool(processes=WORKER COUNT)
116         for key, value in journeys.items():
117             pool.apply async(aggregate, (key, value), callback=update)
118         pool.close()
119         pool.join()
120
121         print('writing day to file...')
```

```
122         with open('collection/journeys {}.json'.format(current date.strftime('%Y-%m-%d')
                ), 'x') as f:
123             json.dump(result, f)
124
125         print('chunk {}/{} written succesfully to {}'.format(days done+1, chunks, f.name
                ))
126         days done = days done + 1
127         current date = current date + timedelta(days=1)
128     exit(0)
```

Quelltext A.2: Abfragen von Fahrplandaten für 300000 Verbindungen

A.3 dataset_builder.py

```
1  import math, requests, time, json, csv, re, random, holidays, pytz, sys
2  import multiprocessing as mp
3  import pandas as pd
4  import numpy as np
5  from tqdm import tqdm
6  from datetime import datetime, timedelta, date, time
7  from geopy.distance import distance
8  from long weekends.long weekends import spot holiday bridges
9
10 ZUG FINDER = '' # private endpoint
11 WORKER COUNT = 4
12 START DATE = date(2022, 10, 1)
13 END DATE = date(2022, 10, 31)
14
15 # load holidays and long weekends
16 holidaysDE = holidays.DE(years=[2021, 2022])
17 bridges, long weekends = spot holiday bridges(start='2021-01-01', end='2022-12-31',
       holidays=holidaysDE)
18
19 # parse time from date
20 def parse time(date):
21     parsed = datetime.fromisoformat(date)
22     time = parsed.strftime("%H:%M") # we only need hours and minutes
23     return time
24
25 # parse minutes since 0:00 from a given date
26 def parse minutes of day(date):
27   return int(date[11:13]) * 60 + int(date[14:16])
28
29 # get the number of legs (parts of a full journey)
30 def get leg number(legs):
31   i = 0
32   for leg in legs:
33     i += 1
34   return i
35
36 # get distance between stations in km
37 def get distance(legs, leg number):
```

```
38    dist = distance(
39        (legs[0]['origin']['location']['latitude'],
40         legs[0]['origin']['location']['longitude']),
41        (legs[leg number - 1]['destination']['location']['latitude'],
42         legs[leg number - 1]['destination']['location']['longitude'])
43    ).km
44    return int(dist)
45
46  # returns the smallest transfer time on any station of the journey
47  def get minimal transfer minutes(legs):
48    min transfer = 9999
49    i = 0
50    j = get leg number(legs) - 1
51    while i < j:
52      inbound = datetime.fromisoformat(legs[i]['arrival'])
53      outbound = datetime.fromisoformat(legs[i+1]['departure'])
54      diff = outbound - inbound
55      diff = int(diff.total seconds() / 60)
56      if diff < min transfer:
57          min transfer = diff
58      i += 1
59      if min transfer == 9999:
60          print(inbound, outbound, diff, i, j)
61          raise ValueError('minTransfer')
62
63    return min transfer
64
65  # checks if date is a holiday
66  def is holiday(date):
67    return (date in holidaysDE)
68
69  # checks if date is one day before a long weekend
70  def is long weekend(date):
71    nextDay = date + timedelta(days=1)
72    if pd.Timestamp(date.date()) not in long weekends and
73      pd.Timestamp(nextDay.date()) in long weekends:
74      return True
75    else:
76      return False
77
78  # parses the final arrival date from the zugfinder response format and localizes it
79  def retrieve arrival date(real journey):
80    arrival leg = real journey['leg ' + str(get leg number(real journey))]
81    tz = pytz.timezone('Europe/Berlin')
82    d = datetime.fromisoformat(arrival leg['arrival date'] + 'T' + arrival leg['
          arrival time'])
83    return tz.localize(d)
84
85  # get final delay in minutes (is 0 if train arrived earlier)
86  def get final delay(real arr date, sched arr date):
87    diff = real arr date - sched arr date
88    diff = int(diff.total seconds() / 60)
89    if diff < 0:
```

```
90      diff = 0
91    return diff
92
93  # uses given legs from a journey to build a request for zugfinder, fires it and returns
        the result
94  def fetch zugfinder(legs):
95      leg number = get leg number(legs)
96      req body = {}
97      i = 0
98      for leg in legs:
99          i = i + 1
100         key = 'leg ' + str(i)
101         req body[key] = {
102             "train": leg['line']['name'],
103             "start stop": leg['origin']['name'],
104             "start time": parse time(leg['departure']),
105             "start date": leg['departure'].split('T', 1)[0],
106             "arrival stop": leg['destination']['name'],
107             "arrival time": parse time(leg['arrival']),
108             "arrival date": leg['arrival'].split('T', 1)[0]
109         }
110
111     # send request
112     req = requests.post(ZUG FINDER, data=json.dumps(req body))
113
114     # extract json with regex (in case the response contains php warnings etc.)
115     match = re.search(r"{\n*.*}", req.text, re.MULTILINE | re.DOTALL)
116     result = ''
117     try:
118         result = json.loads(match.group())
119     except json.decoder.JSONDecodeError:
120         tqdm.write('zugfinder json parse failed: {}, {}'. format(match.group(), match.
              string))
121         return 1
122
123     # check if data is valid
124     arrival leg = result['leg ' + str(get leg number(result))]
125     if arrival leg['arrival time'] == None:
126         tqdm.write('discarding due to missing/invalid ZF arrival time')
127         #tqdm.write(str(result))
128         return 1
129
130     if 'Bus SEV' in str(result):
131         tqdm.write('discarding due to replacement bus leg in historical data...')
132         #tqdm.write(str(result))
133         return 1
134
135     return result
136
137 def generate set(key, value):
138     # check for missing data
139     if isinstance(value, list) == False:
140         tqdm.write('discarding due to missing input data')
```

```
141         tqdm.write(str(key))
142         return 1
143 
144     # get real journey
145     zf = fetch zugfinder(value)
146     #tqdm.write(str(zf))
147     if zf == 1:
148         return 1
149 
150     journey = eval(key)
151     dep = journey[0]
152     arr = journey[1]
153 
154     legNumber = len(value)
155     sched dep date = datetime.fromisoformat(value[0]['departure'])
156     sched arr date = datetime.fromisoformat(value[legNumber - 1]['arrival'])
157 
158     journey data = {
159         'dep': dep,
160         'arr': arr,
161         'dep year': sched dep date.year,
162         'dep month': sched dep date.month,
163         'dep day': sched dep date.day,
164         'dep time': parse minutes of day(sched dep date.isoformat()),
165         'dep lat': value[0]['origin']['location']['latitude'],
166         'dep long': value[0]['origin']['location']['longitude'],
167         'arr lat': value[legNumber - 1]['destination']['location']['latitude'],
168         'arr long': value[legNumber - 1]['destination']['location']['longitude'],
169         'dist': get distance(value, legNumber),
170         'legs': legNumber,
171         'min transfer': get minimal transfer minutes(value),
172         'day of week': sched dep date.weekday(),
173         'is holiday': is holiday(sched dep date),
174         'long we': is long weekend(sched dep date),
175         'final delay': get final delay(retrieve arrival date(zf), sched arr date)
176     }
177     #print(journeyData)
178     new row df = pd.DataFrame([journey data])
179     return new row df
180 
181 # main function
182 if   name   == '  main  ':
183 
184     # first parameter is the output file
185     tqdm.write('results will be appended to {}'.format(sys.argv[1]))
186 
187     # iterate over range of dates
188     while (START DATE <= END DATE):
189         path = 'collection 1/journeys ' + str(START DATE) + '.json'
190         tqdm.write('reading ' + path)
191         with open(path) as f:
192             journeys = json.load(f)
193 
```

```
194         path = 'collection 2/journeys ' + str(START DATE) + '.json'
195         tqdm.write('reading ' + path)
196         with open(path) as f:
197             journeys 2 = json.load(f)
198
199         journeys.update(journeys 2)
200
201         tqdm.write('starting chunk #{}'.format(START DATE))
202         pbar = tqdm(total=len(journeys.items()), desc='collecting data for {}'.format(
                START DATE), unit='journeys', colour='green')
203         result df = pd.DataFrame()
204
205         # concat result and update progress bar
206         def update(a):
207             global result df
208             if isinstance(a, pd.DataFrame) == True:
209                 result df = pd.concat([result df, a])
210             pbar.update()
211
212         # start worker pool
213         pool = mp.Pool(processes=WORKER COUNT)
214         for key, value in journeys.items():
215             pool.apply async(generate set, (key, value), callback=update)
216         pool.close()
217         pool.join()
218         result df.to csv(sys.argv[1], mode='a', header=False, index=False)
219
220         tqdm.write('chunk #{} completed'.format(START DATE))
221         START DATE += timedelta(days=1)
222
223     tqdm.write('done')
224     exit(0)
```

Quelltext A.3: Aggregation der Basisfeatures

A.4 delay_history.py

```
1  import math
2  import pandas as pd
3  from datetime import datetime, timedelta
4  from pandarallel import pandarallel
5
6  pandarallel.initialize(progress bar=True)
7
8  # returns delay history for a given row
9  def aggregate delay history(row):
10     #print(row.values)
11     row date = datetime(row.dep year,
12         row.dep month,
13         row.dep day,
14         math.floor(row.dep time / 60),
15         row.dep time % 60)
```

```
16
17      # last 4 weeks
18      delayHist4W1 = get historic delay(row.dep, row.arr, row date - timedelta(days=7))
19      delayHist4W2 = get historic delay(row.dep, row.arr, row date - timedelta(days=14))
20      delayHist4W3 = get historic delay(row.dep, row.arr, row date - timedelta(days=21))
21      delayHist4W4 = get historic delay(row.dep, row.arr, row date - timedelta(days=28))
22
23      # last 4 days
24      delayHist4D1 = get historic delay(row.dep, row.arr, row date - timedelta(days=1))
25      delayHist4D2 = get historic delay(row.dep, row.arr, row date - timedelta(days=2))
26      delayHist4D3 = get historic delay(row.dep, row.arr, row date - timedelta(days=3))
27      delayHist4D4 = get historic delay(row.dep, row.arr, row date - timedelta(days=4))
28
29      #print('{}, {}, {}, {}, {}, {}, {}, {}'.format(delayHist4W1, delayHist4W2,
            delayHist4W3, delayHist4W4, delayHist4D1, delayHist4D2, delayHist4D3,
            delayHist4D4))
30
31      return pd.Series([
32          delayHist4W1, delayHist4W2,
33          delayHist4W3, delayHist4W4,
34          delayHist4D1, delayHist4D2,
35          delayHist4D3, delayHist4D4],
36          index =['delayHist 4w 1', 'delayHist 4w 2',
37          'delayHist 4w 3', 'delayHist 4w 4',
38          'delayHist 4d 1', 'delayHist 4d 2',
39          'delayHist 4d 3', 'delayHist 4d 4'])
40
41  def get historic delay(departure, arrival, date):
42      #print(departure, arrival, date)
43      try:
44          delay = df[(df.dep == departure) &
45              (df.arr == arrival) &
46              (df.dep year == date.year) &
47              (df.dep month == date.month) &
48              (df.dep day == date.day)].final delay.values[0]
49      except IndexError:
50          delay = pd.NA
51      return delay
52
53  # main function
54  if   name   == '  main  ':
55      df = pd.read csv('dataset/dataset merged.csv')
56      df.reset index(inplace=True)
57      result df = df.assign(**df.parallel apply(aggregate delay history, result type='
            expand', axis=1))
58      result df.to csv('dataset/dataset v2 final.csv')
```

Quelltext A.4: Aggregation der Verspätungshistorie

A.5 weather_helper.py

```
1  import math
```

```
import pandas as pd
import numpy as np
import xarray as xr
from datetime import datetime, timedelta
from tqdm import tqdm

INPUT FILE = 'dataset v4 part5.csv'
OUTPUT FILE = 'dataset v4 final.csv'

SHORT NAME = 'sf'
FEATURE NAME = 'snow fall'
#ERA5 FILE = 'data/cds data 230318 2m temperature.nc'
#ERA5 FILE = 'data/cds data 230318 2m dewpoint temperature.nc'
#ERA5 FILE = 'data/cds data 230318 total precipitation.nc'
#ERA5 FILE = 'data/cds data 230318 snow cover.nc'
#ERA5 FILE = 'data/cds data 230318 snow depth.nc'
ERA5 FILE = 'data/cds data 230318 snowfall.nc'

# load ERA5 data
f = xr.open dataset(ERA5 FILE)

# retrieves ERA5 data from netcdf File
def aggregate weather data(row):
    #print(row)
    t hours = round(row.dep time / 60)
    if (t hours == 24):
        t hours = 23
    t = datetime(row.dep year, row.dep month, row.dep day, t hours)

    sel array dep = f.sel(time=t,
        longitude=row.dep long,
        latitude=row.dep lat,
        method='nearest')
    sel array arr = f.sel(time=t,
        longitude=row.arr long,
        latitude=row.arr lat,
        method='nearest')

    val dep = sel array dep[SHORT NAME].values[0]
    val arr = sel array arr[SHORT NAME].values[0]
    #print([valDep, valArr])

    return pd.Series([val dep, val arr],
        index =['era5 dep ' + FEATURE NAME, 'era5 arr ' + FEATURE NAME])

# read extend and write dataset
df = pd.read csv(INPUT FILE)

tqdm.pandas()
era5 df = df.progress apply(aggregate weather data, axis=1)
resultDF = pd.concat([df, era5 df], axis=1)
resultDF.to csv(OUTPUT FILE,mode='x', index=False)
```

```
54 print('done')
```

Quelltext A.5: Aggregation der Wetterfeatures

A.6 chainlink-job.toml

```
1  type = "directrequest"
2  schemaVersion = 1
3  name = "master-thesis"
4  contractAddress = "0x4a82D5B73f22bF09AeCBA78c206ccb858E41f4b2"
5  maxTaskDuration = "0s"
6  observationSource = """
7  decode log   [
8  type="ethabidecodelog" abi="OracleRequest(bytes32 indexed specId, address requester,
       bytes32 requestId, uint256 payment, address callbackAddr, bytes4 callbackFunctionId,
        uint256 cancelExpiration, uint256 dataVersion, bytes data)"
9  data="$(jobRun.logData)"
10 topics="$(jobRun.logTopics)"
11 ]
12 decode cbor  [type="cborparse" data="$(decode log.data)"]
13 fetch [
14 type="bridge"
15 name="masterthesis"
16 requestData="{\\"departure\\": $(decode cbor.departure), \\"arrival\\": $(decode cbor.
       arrival), \\"departureDate\\": $(decode cbor.departureDate)}"
17 ]
18     prediction   [type="jsonparse" path="delayProbability" data="$(fetch)"]
19     multiply     [type="multiply" input="$(prediction)" times=100]
20     encode data  [type="ethabiencode" abi="(bytes32 requestId, uint256 prediction)" data
           ="{ \\"requestId\\": $(decode log.requestId), \\"prediction\\": $(multiply)}"]
21     encode tx    [type="ethabiencode"
22                   abi="fulfillOracleRequest2(bytes32 requestId, uint256 payment, address
                         callbackAddress, bytes4 callbackFunctionId, uint256 expiration,
                        bytes calldata data)"
23                   data="{\\"requestId\\": $(decode log.requestId), \\"payment\\": $(
                        decode log.payment), \\"callbackAddress\\": $(decode log.
                        callbackAddr), \\"callbackFunctionId\\": $(decode log.
                        callbackFunctionId), \\"expiration\\": $(decode log.
                        cancelExpiration), \\"data\\": $(encode data)}"
24                  ]
25     submit tx    [type="ethtx" to="0x4a82D5B73f22bF09AeCBA78c206ccb858E41f4b2" data="$(
           encode tx)"]
26     decode log -> decode cbor -> fetch -> prediction -> multiply -> encode data ->
           encode tx -> submit tx
27 """
```

Quelltext A.6: Chainlink-Jobspezifikation des Direct Request

A.7 Prediction.sol

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.7;
```

```
3
4  import "@chainlink/contracts/src/v0.8/ChainlinkClient.sol";
5  import "@chainlink/contracts/src/v0.8/ConfirmedOwner.sol";
6
7  contract Prediction is ChainlinkClient, ConfirmedOwner {
8      using Chainlink for Chainlink.Request;
9
10     address private constant LINK TOKEN =
11         0x326C977E6efc84E512bB9C30f76E30c160eD06FB;
12     uint256 private constant ORACLE PAYMENT = 0; // 0 LINK for testing
13     uint256 public prediction; // contains the result of the chainlink call
14
15     address public oracle;
16     string public jobId;
17
18     constructor() ConfirmedOwner(msg.sender) {
19         setChainlinkToken(LINK TOKEN);
20     }
21
22     function setOracle(address  oracle) public onlyOwner {
23         oracle =  oracle;
24     }
25
26     function setJobId(string memory  jobId) public onlyOwner {
27         jobId =  jobId;
28     }
29
30     function requestPrediction(
31         string memory  departure,
32         string memory  arrival,
33         string memory  departureDate
34     ) public onlyOwner {
35         Chainlink.Request memory req = buildChainlinkRequest(
36             stringToBytes32(jobId),         // job id
37             address(this),              // contract address
38             this.fulfillPrediction.selector         // selector of the callback function
39         );
40
41         req.add("departure",  departure);     // "Leipzig Hbf"
42         req.add("arrival",  arrival);       // "München Hbf"
43         req.add("departureDate",  departureDate); // "2023-01-28T09:12:00"
44
45         sendChainlinkRequestTo(oracle, req, ORACLE PAYMENT);
46     }
47
48     // callback function, called by the oracle when the response is ready
49     function fulfillPrediction(
50         bytes32  requestId,
51         uint256  prediction
52     ) public recordChainlinkFulfillment( requestId) {
53         prediction =  prediction;
54     }
55
```

```
56      function getChainlinkToken() public view returns (address) {
57          return chainlinkTokenAddress();
58      }
59
60      function withdrawLink() public onlyOwner {
61          LinkTokenInterface link = LinkTokenInterface(chainlinkTokenAddress());
62          require(
63              link.transfer(msg.sender, link.balanceOf(address(this))),
64              "Unable to transfer"
65          );
66      }
67
68      function cancelRequest(
69          bytes32  requestId,
70          uint256  payment,
71          bytes4  callbackFunctionId,
72          uint256  expiration
73      ) public onlyOwner {
74          cancelChainlinkRequest(
75               requestId,
76               payment,
77               callbackFunctionId,
78               expiration
79          );
80      }
81
82      function stringToBytes32(string memory source)
83          private
84          pure
85          returns (bytes32 result)
86      {
87          bytes memory tempEmptyStringTest = bytes(source);
88          if (tempEmptyStringTest.length == 0) {
89              return 0x0;
90          }
91
92          assembly {
93              // solhint-disable-line no-inline-assembly
94              result := mload(add(source, 32))
95          }
96      }
97  }
```

Quelltext A.7: Solidity-Quelltext des Prediction-Contracts

Literaturverzeichnis

[1] ZDFheute, *Statistik: 2021 Züge wieder unpünktlicher*, https://www.zdf.de/nachrichten/wirtschaft/verspaetete-zuege-bahn-wieder-unpuenktlicher-100.html, 2022. (besucht am 15. 04. 2023).

[2] Etherisc GmbH Blockchain Competence Center Mittweida, *Community Supported Insurance - Gemeinschaftsorientierte Versicherung mittels Smart Contracts*, https://csi.hs-mittweida.de/files/csi-onepager.pdf, 2021. (besucht am 17. 04. 2023).

[3] Lori Perri, *What's New in the 2022 Gartner Hype Cycle for Emerging Technologies*, https://www.gartner.com/en/articles/what-s-new-in-the-2022-gartner-hype-cycle-for-emerging-technologies, 2022. (besucht am 24. 03. 2023).

[4] Birthe Sönnichsen und Gabor Halasz, *Verspätung als Dauerzustand: Wie die Bahn pünktlich werden könnte*, https://www.tagesschau.de/inland/gesellschaft/deutsche-bahn-puenktlichkeit-101.html, 2023. (besucht am 11. 04. 2023).

[5] Pu Wang und Qing-peng Zhang, „Train delay analysis and prediction based on big data fusion", *Transportation Safety and Environment*, Jg. 1, Nr. 1, S. 79–88, Feb. 2019, ISSN: 2631-4428. DOI: 10.1093/tse/tdy001. eprint: https://academic.oup.com/tse/article-pdf/1/1/79/29227827/tdy001.pdf. Adresse: https://doi.org/10.1093/tse/tdy001.

[6] Ping Huang, Chao Wen, Liping Fu, Qiyuan Peng und Yixiong Tang, „A deep learning approach for multi-attribute data: A study of train delay prediction in railway systems", *Information Sciences*, Jg. 516, S. 234–253, 2020, ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2019.12.053. Adresse: https://www.sciencedirect.com/science/article/pii/S0020025519311715.

[7] E. Alpaydin, *Introduction to Machine Learning, second edition* (Adaptive Computation and Machine Learning series). MIT Press, 2009, ISBN: 9780262303262. Adresse: https://books.google.de/books?id=TtrxCwAAQBAJ.

[8] Trevor Hastie, Robert Tibshirani und Jerome Friedman, *The Elements of Statistical Learning* (Springer Series in Statistics). New York, NY, USA: Springer New York Inc., 2001.

[9] Thomas A. Runkler, „Einführung", in *Data Mining: Modelle und Algorithmen intelligenter Datenanalyse*. Wiesbaden: Springer Fachmedien Wiesbaden, 2015, S. 1–3, ISBN: 978-3-8348-2171-3. DOI: 10.1007/978-3-8348-2171-3_1. Adresse: https://doi.org/10.1007/978-3-8348-2171-3_1.

[10] Cristiano Breuel, *ML Ops: Machine Learning as an Engineering Discipline*, https://towardsdatascience.com/ml-ops-machine-learning-as-an-engineering-discipline-b86ca4874a3f, 2023. (besucht am 12. 04. 2020).

[11] Andreas M. Antonopoulos, *Mastering Bitcoin*, https://github.com/bitcoinbook/bitcoinbook, 2017. (besucht am 27. 01. 2023).

[12] CoinMarketCap, *Today's Cryptocurrency Prices by Market Cap, 2023-01-29 17:56:15 UTC*, https://web.archive.org/web/20230129175615/https://coinmarketcap.com/, 2023. (besucht am 30. 01. 2023).

[13] Patrick Collins, *What Is a Blockchain Oracle?*, https://betterprogramming.pub/what-is-a-blockchain-oracle-f5ccab8dbd72, 2020. (besucht am 31. 01. 2023).

[14] Cem Dilmegani, *Guide to Oracles in 2023: What Are They, Types & Use Cases*, https://research.aimultiple.com/blockchain-oracle/, 2023. (besucht am 27. 01. 2023).

[15] Chainlink Labs, *Chainlink 2.0: Next Steps in the Evolution of Decentralized Oracle Networks*, https://research.chain.link/whitepaper-v2.pdf, 2021. (besucht am 30. 01. 2023).

[16] Chainlink Foundation, *Decentralized Data Model*, https://docs.chain.link/architecture-overview/architecture-decentralized-model/, 2022. (besucht am 31. 01. 2023).

[17] Achim Tack Holger Dambeck Maximilian Schäfer, *Wie pünktlich sind Bus, Bahn und Flugzeug wirklich?*, https://www.spiegel.de/reise/aktuell/verspaetungsquoten-wie-puenktlich-flugzeug-bus-und-bahn-wirklich-sind-a-1299889.html, 2023. (besucht am 11. 02. 2023).

[18] Zugfinder.net, *Verspätungsdaten und Pünktlichkeit von Zug ICE 501*, https://www.zugfinder.net/de/zug-ICE 501-365, 2023. (besucht am 11. 02. 2023).

[19] Deutsche Bahn, *Fahrplan API*, https://data.deutschebahn.com/dataset/api-fahrplan.html, 20´16. (besucht am 25. 02. 2023).

[20] David Kriesel, *Lecture: BahnMining - Pünktlichkeit ist eine Zier*, https://media.ccc.de/v/36c3-10652-bahnmining - punktlichkeit ist eine zier, 2019. (besucht am 24. 02. 2023).

[21] Fred Wagner, „R", in *Gabler Versicherungslexikon*, Fred Wagner, Hrsg. Wiesbaden: Springer Fachmedien Wiesbaden, 2017, S. 715–799, ISBN: 978-3-8349-4625-6. DOI: 10.1007/978-3-8349-4625-6 18. Adresse: https://doi.org/10.1007/978-3-8349-4625-6 18.

[22] Carolyn Kousky Rohini Sengupta, *Parametric Insurance for Disasters*, https://riskcenter.wharton.upenn.edu/wp-content/uploads/2020/09/Parametric-Insurance-for-Disasters Sep-2020.pdf, 2020. (besucht am 19. 04. 2023).

[23] Charles E. Metz, „Basic principles of ROC analysis", *Seminars in Nuclear Medicine*, Jg. 8, Nr. 4, S. 283–298, 1978, ISSN: 0001-2998. DOI: https://doi.org/10.1016/S0001-2998(78)80014-2. Adresse: https://www.sciencedirect.com/science/article/pii/S0001299878800142.

[24] Tom Fawcett, *An introduction to ROC analysis*, https://www.sciencedirect.com/science/article/abs/pii/S016786550500303X, 2005. (besucht am 01. 12. 2022).

[25] Jorge M. Lobo, Alberto Jiménez-Valverde und Raimundo Real, „AUC: a misleading measure of the performance of predictive distribution models“, *Global Ecology and Biogeography*, Jg. 17, Nr. 2, S. 145–151, 2008. DOI: https://doi.org/10.1111/j.1466-8238.2007.00358.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1466-8238.2007.00358.x. Adresse: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1466-8238.2007.00358.x.

[26] Joos Korstanje, *The F1 score*, https://towardsdatascience.com/the-f1-score-bec2bbc38aa6, 2021. (besucht am 13. 03. 2023).

[27] Deutsche Bahn AG, *Haltestellen (Stand 01/2020)*, https://data.deutschebahn.com/dataset/data-haltestellen/resource/21edf505-e97d-4c99-bcc9-a46e85f8620f.html, 2020. (besucht am 11. 04. 2023).

[28] handelsdaten.de, *Anzahl der Besucher und Reisenden ausgewählter Bahnhöfe in Deutschland pro Tag im Jahr 2017*, https://www.handelsdaten.de/deutschsprachiger-einzelhandel/travel-retail-taegliche-anzahl-besucher-reisenden-bahnhoefe, 2017. (besucht am 28. 11. 2022).

[29] Bundesregierung, *Antwort der Bundesregierung auf die Kleine Anfrage der Abgeordneten Torsten Herbst, Frank Sitta, Dr. Christian Jung, weiterer Abgeordneter und der Fraktion der FDP – Drucksache 19/19475 – Verlässlichkeit des Schienenverkehrs an Knotenbahnhöfen*, https://dserver.bundestag.de/btd/19/204/1920455.pdf, 2020. (besucht am 28. 11. 2022).

[30] Deutsche Bahn AG, *Die sieben Bahnhofskategorien*, https://web.archive.org/web/20160922164726/http://www.deutschebahn.com/de/geschaefte/infrastruktur/bahnhof/stationsnutzung/11878066/bahnhofs_kategorien.html, 2016. (besucht am 22. 09. 2016).

[31] Sascha Geldermann Lukas von Hoyer Tiana Zoric, *Alle Infos zum 9-Euro-Ticket: Wo gilt es und was muss man beachten?*, https://www.augsburger-allgemeine.de/wirtschaft/9-euro-ticket-wo-kaufen-wo-gueltig-fuer-wen-wie-lange-id62259056.html, 2022. (besucht am 06. 04. 2023).

[32] Nikolaus Doll, *Mit diesen Tricks umgehen Sie rappelvolle Züge*, https://www.welt.de/wirtschaft/article106381146/Mit-diesen-Tricks-umgehen-Sie-rappelvolle-Zuege.html, 2012. (besucht am 29. 11. 2022).

[33] MDR Sachsen-Anhalt, *Fahrplanwechsel Mehr Direktzüge aus Mitteldeutschland an die Ostsee*, https://www.mdr.de/nachrichten/deutschland/wirtschaft/bahn-fahrplanwechsel-neue-verbindungen-fernzuege-preise-100.html, 2022. (besucht am 06. 04. 2023).

[34] Wes McKinney, „Data Structures for Statistical Computing in Python“, in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt und Jarrod Millman, Hrsg., 2010, S. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[35] Public Transport, *transport.rest transit APIs*, https://github.com/public-transport, 2022. (besucht am 20. 12. 2022).

[36] NumFOCUS xarray Developers, *Xarray documentation: xarray.DataArray.sel*, https://docs.xarray.dev/en/stable/generated/xarray.DataArray.sel.html, 2022. (besucht am 22. 12. 2022).

[37] J. Muñoz Sabater, *ERA5-Land hourly data from 1981 to present. Copernicus Climate Change Service (C3S) Climate Data Store (CDS)*, 2019. DOI: 10.24381/cds.e2161bac. (besucht am 22. 12. 2022).

[38] Guillaume Dueymes, *Netcdf: Select closest points using CKDTree*, https://www.guillaumedueymes.com/post/ckdtree netcdf/, 2020. (besucht am 09. 12. 2022).

[39] Maurizio Montel, *holidays - Generate and work with holidays in Python*, https://pypi.org/project/holidays/, 2014. (besucht am 08. 03. 2023).

[40] Giulia Savorgnan, *python-long-weekends - Spot holiday "bridgesänd long weekends*, https://pypi.org/project/python-long-weekends/, 2021. (besucht am 08. 03. 2023).

[41] Kostya Esmukov, *geopy*, https://pypi.org/project/geopy/, 2006. (besucht am 17. 04. 2023).

[42] Thomas Kluyver u. a., „Jupyter Notebooks – a publishing format for reproducible computational workflows“, in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides und B. Schmidt, Hrsg., IOS Press, 2016, S. 87–90.

[43] Dario Radečić, *Imputing Numerical Data: Top 5 Techniques Every Data Scientist Must Know*, https://towardsdatascience.com/imputing-numerical-data-top-5-techniques-every-data-scientist-must-know-587c0f51552a, 2021. (besucht am 23. 04. 2023).

[44] Piotr Płoński, *Random Forest Feature Importance Computed in 3 Ways with Python*, https://mljar.com/blog/feature-importance-in-random-forest/, 2020. (besucht am 10. 03. 2023).

[45] Shankar Rao Pandala, *Lazy Predict*, https://github.com/shankarpandala/lazypredict, 2019. (besucht am 15. 03. 2023).

[46] Rich Caruana und Alexandru Niculescu-Mizil, „An Empirical Comparison of Supervised Learning Algorithms“, in *Proceedings of the 23rd International Conference on Machine Learning*, Ser. ICML '06, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2006, S. 161–168, ISBN: 1595933832. DOI: 10.1145/1143844.1143865. Adresse: https://doi.org/10.1145/1143844.1143865.

[47] L. Breiman, J. Friedman, C.J. Stone und R.A. Olshen, *Classification and Regression Trees*. Taylor & Francis, 1984, ISBN: 9780412048418. Adresse: https://books.google.de/books?id=JwQx-WOmSyQC.

[48] F. Pedregosa u. a., „Scikit-learn: Machine Learning in Python“, *Journal of Machine Learning Research*, Jg. 12, S. 2825–2830, 2011.

[49] scikit-learn developers, *Decision Trees - Tree algorithms: ID3, C4.5, C5.0 and CART*, https://scikit-learn.org/stable/modules/tree.html#tree-algorithms-id3-c4-5-c5-0-and-cart, 2023. (besucht am 12. 05. 2023).

[50] Nima Beheshti, *Random Forest Classification*, https://towardsdatascience.com/random-forest-classification-678e551462f5, 2022. (besucht am 17. 04. 2023).

[51] Leo Breiman, „Bagging Predictors“, *Machine Learning*, Jg. 24, Nr. 2, S. 123–140, 1996, ISSN: 1573-0565. DOI: 10.1023/A:1018054314350. Adresse: https://doi.org/10.1023/A:1018054314350.

[52] Ingrid Lunden, *Yandex open sources CatBoost, a gradient boosting machine learning library*, https://techcrunch.com/2017/07/18/yandex-open-sources-catboost-a-gradient-boosting-machine-learning-librar/, 2017. (besucht am 06. 02. 2023).

[53] Cheng Li, *A Gentle Introduction to Gradient Boosting*, http://www.chengli.io/tutorials/gradient_boosting.pdf, 2019. (besucht am 06. 02. 2023).

[54] Vasiliy Ershov, *CatBoost Enables Fast Gradient Boosting on Decision Trees Using GPUs*, https://developer.nvidia.com/blog/catboost-fast-gradient-boosting-decision-trees/, 2018. (besucht am 06. 02. 2023).

[55] Corinna Cortes und Vladimir Vapnik, „Support-vector networks“, *Machine Learning*, Jg. 20, Nr. 3, S. 273–297, Sep. 1995. DOI: 10.1007/bf00994018. Adresse: https://doi.org/10.1007/bf00994018.

[56] Nello Cristianini und John Shawe-Taylor, „Support Vector Machines“, in *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000, S. 93–124. DOI: 10.1017/CBO9780511801389.008.

[57] Bernhard Schölkopf und Alexander J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. The MIT Press, Juni 2018, ISBN: 9780262256933. DOI: 10.7551/mitpress/4175.001.0001. Adresse: https://doi.org/10.7551/mitpress/4175.001.0001.

[58] Andrew McCallum und Kamal Nigam, „A comparison of event models for naive bayes text classification“, in *AAAI Conference on Artificial Intelligence*, 1998.

[59] Paul Bennett, „Assessing the Calibration of Naive Bayes' Posterior Estimates“, Okt. 2000.

[60] scikit-learn developers, *Python specific serialization*, https://scikit-learn.org/stable/model_persistence.html, 2023. (besucht am 07. 03. 2023).

[61] Sebastián Ramírez, *FastAPI*, https://fastapi.tiangolo.com/, 2018. (besucht am 27. 03. 2023).

[62] Mohammad Shahebaz, *katana*, https://github.com/shaz13/katana, 2021. (besucht am 27. 03. 2023).

[63] Chainlink Foundation, *External Adapters in Solidity*, https://docs.chain.link/chainlink-nodes/external-adapters/contract-creators, 2022. (besucht am 28. 03. 2023).

[64] Chainlink Foundation, *Basic Request Model*, https://docs.chain.link/architecture-overview/architecture-request-model/, 2022. (besucht am 03. 04. 2023).

[65] Remix Project Team, *Remix IDE - The Native IDE for Web3 Development*, https://remix.ethereum.org, 2022. (besucht am 23. 04. 2023).

Eidesstattliche Erklärung

Hiermit versichere ich – Lucas Johns – an Eides statt, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt oder anderweitig veröffentlicht.

Lichtentanne, 18.05.2023

Ort, Datum

