InformatiCup 2018 - IntelliTank

Studienarbeit

für die Prüfung zum

Bachelor of Science

des Studienganges Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Robin Baumann, Carlo Götz, Tom Ganz

Abgabedatum 22. Mai 2018

Zeitraum	3. Studienjahr
Kurs	TINF15B5
Gutachter der Studienakademie	Prof. Dr. Johannes Freudenmann

Erklärung

Wir versichern hiermit, dass wir unsere Studienarbeit mit dem Thema: „InformatiCup 2018 - IntelliTank“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort Datum Unterschriften

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface 7

ARMA Autoregressive-Moving Average II

ARIMA Autoregressive Integrated Moving Average II

CLI Command Line Interface 18

CORS Cross-Origin Resource Sharing 8

CSV Comma Separated Values 11

DI Dependency Injection 6

HTTP Hypertext Transfer Protocol 6

IOC Inversion Of Control 6

JSF Java Server Faces 6

JSON Javascript Object Notation 6

OSM Open Street Map 9

RDBMS Relational Database Management System 8

RMSE Root Mean Squared Error 17

SPA Single Page Application 11

1. Einleitung

1.1 InformatiCup

Die hier beschriebene Arbeit behandelt die Lösung der Aufgabe IntelliTank. Diese ist Bestandteil des 13. InformatiCup-Wettbewerbs, welcher von der Gesellschaft für Informatik e.V. jährlich organisiert wird. Anlässlich des InformatiCups treten Teams aus verschiedenen Universitäten gegeneinander an, um Ihre Lösungen vorzustellen. Die besten vier werden durch eine Jury gekürt und erhalten einen Preis, sowie eine Einladung zu einer Fachtagung der Gesellschaft für Informatik, auf welcher sie ihre Lösung einem Publikum präsentieren dürfen.

1.2 Aufgabenstellung

Die diesjährige Problemstellung trägt den Namen IntelliTank. Aufgabe ist es, ein Modell zur Vorhersage zukünftiger Benzinpreis(Superkraftstoff E5)-Entwicklungen an allen Tankstellen Deutschlands zu entwickeln. Die Vorhersagen sollen dabei bis zu einem Monat voraus mit einer gewissen Signifikanz berechnet werden. Als Grundlage dieser Arbeit wird von Tankerkönig.de eine Datenbasis zur Verfügung gestellt. Diese beinhaltet die unbereinigten historischen Benzinpreise von rund 15.000 Tankstellen in Deutschland seit 2013. Mithilfe dieser Grundlage ist es möglich eine Korrelationsanalyse durchzuführen. Zusätzlich können weitere Datenquellen mit in das Modell einbezogen werden, um die allgemeine Vorhersagegüte des statistischen Modells zu verbessern.

Abhängig davon soll eine Anwendung entworfen werden, welche auf Basis des Vorhersagemodells zu einer gegebenen Route eine optimale Tankstrategie liefert. Zur Einfachheit wird eine Route definiert durch verschiedene Stopps an Tankstellen und dem jeweiligen Zeitpunkt an den man ihn erreicht. Die Anwendung soll also zu diesen Zeitpunkten der jeweiligen Tankstelle einen vorhergesagten Preis festlegen und als Ergebnis zu jeder Tankstelle eine optimale Menge Benzin ermitteln.

1.3 Motivation

Das Auto ist in Deutschland das Fortbewegungsmittel Nummer 1. Einer Umfrage des Umweltbundesamtes (UBA) zufolge sind 37 Prozent der Menschen in Deutschland (ab 14 Jahre) täglich mit dem Kfz unterwegs [1]. Außerdem gibt ein Privathaushalt in Deutschland jeden Monat rund 100 Euro für Autokraftstoff aus. Auch wenn die Elektromobilität stark auf dem Vormarsch ist, besitzen die meisten Deutschen noch ein Fahrzeug mit einem Verbrennungsmotor. Durch eine intelligente, vorausschauende Tankstrategie lässt sich bares Geld sparen.

2. Theoretische Grundlagen

In diesem Kapitel werden die Theoretischen Grundlagen zur in dieser Arbeit vorgestellten Lösung zum 13. InformatiCup der Gesellschaft für Informatik e.V. vorgestellt. Zunächst wird das Fixed-Path-Gas-Station Problem erklärt, welches die Basis des zu entwerfenden Routing-Algorithmus darstellt. Anschließend werden die Grundlagen der Zeitreihenanalyse und Regressionsanalyse vorgestellt.

2.1 Fixed-Path-Gas-Station Problem

Die hier verwendete Tankstrategie wurde implementiert nach dem Algorithmus aus der Publikation 'To Fill or Not To Fill'[2]. Gegeben sei eine Route, welche definiert ist durch eine Menge von Tankstellen und eine maximale Tankkapazität des Fahrzeuges. Jede Tankstelle ist abzufahren, wobei die erste Tankstelle dem Startpunkt der Route entspricht. Dadurch ergibt sich folgender Algorithmus:

1. Sei b die aktuelle Position gegeben durch eine Tankstelle
2. Sei l eine Liste von Tankstellen die durch die maximale Tankkapazität von b aus erreichbar sind
3. Sortiere die Liste nach den Benzinpreisen und der Nähe zum Ziel aufsteigend
4. Tanke genau so viel, dass die günstigste (erste) Tankstelle c aus l erreicht wird
5. Falls c nicht die End-Tankstelle ist, wiederhole Punkt 1

Ebenfalls benutzen wir eine naive Tankstrategie (immer volltanken) um sie mit unserer zu vergleichen. Der Algorithmus wird beschrieben durch:

1. Sei $l_{i=0}$ die aktuelle Position gegeben durch eine Tankstelle
2. Tanke voll (Kapazität-Füllstand)
3. Erreiche die nächste Tankstelle l_{i+1}
4. Tanke die während der Distanz von l_i nach l_{i+1} verbrauchten Liter wieder auf
5. Falls l_{i+1} nicht die End-Tankstelle ist, wiederhole Punkt 3

Die Distanz zwischen zwei Tankstellen wird mittels der Haversine-Formel zwischen zwei Geo-Punkten berechnet [3].

2.2 Zeitreihenanalyse

Bei einer Zeitreihe handelt es sich um eine zeitlich geordnete Sequenz von Messungen eines Systems [4, S. 145]. Das in dieser Arbeit behandelte Problem kann als eine Zeitreihenanalyse betrachtet werden. Viele Algorithmen zur Analyse von Zeitreihen funktionieren stabiler, wenn die zu untersuchende Zeitreihe mit einer gleichmäßigen Frequenz vorliegt [4, S. 146]. Dies ist nicht immer gegeben. In dem verwendeten Datensatz sind die Benzinpreise zu den jeweiligen Änderungszeitpunkten in der Datenbank gespeichert. Aus diesem Grund wechseln die Abstände zwischen zwei aufeinanderfolgenden Messungen von wenigen Minuten bis zu mehreren Stunden. Manche Tankstellen weisen sogar Lücken von mehreren Tagen auf. Aus diesem Grund ist eine Vorverarbeitung des Datensatzes notwendig, um die Messungen auf eine einheitliche Frequenz zu bringen. Genaueres hierzu wird in Abschnitt 3.3 erläutert. Auf den vorbereiteten Daten lassen sich nun unterschiedliche Methoden zur Analyse der Zeitreihe anwenden, welche nachfolgend beschrieben werden.

2.2.1 Gleitende Durchschnitte

Die meisten Zeitreihen setzen sich aus den folgenden drei Komponenten zusammen [4, S. 151]:

1. *Trend*: Systematische zu- oder Abnahme.
2. *Saisonalität*: Periodische Schwankungen.
3. *Fluktuation*: Zufällige Abweichungen vom Langzeittrend.

Zur Analyse der zugrundeliegenden Zeitreihe werden diese einzelnen Komponenten sukzessive, nicht simultan eliminiert [5, S. 397]. Bei der Methode der gleitenden Durchschnitte wird dabei der Trend der zugrundeliegenden Zeitreihe berechnet. Gleitende Durchschnitte werden dabei als eine Folge arithmetischer Mittel von jeweils p aufeinanderfolgenden Werten der Zeitreihe gebildet werden [5, S. 403]. Durch den p-gliedrigen gleitenden Durchschnitt entsteht eine geglättete Zeitreihe, die mit größerem p immer glatter wird.

2.2.2 Autokorrelation

Autokorrelation beschreibt die Korrelation einer Funktion mit sich selber zu einem früheren Zeitpunkt [4, S. 155]. Um die Saisonalität herauszufinden wird die Funktion um ein sogenanntes *lag*-Intervall verschoben und auf Korrelation mit der originalen Funktion geprüft. Dies wird mit unterschiedlichen Intervallen wiederholt, um entsprechend auf tägliche, wöchentliche oder monatliche Trends schließen zu können. Für jede dieser Verschiebungen wird dann der Korrelationskoeffizient im Wertebereich [0,1] berechnet. Anhand der berechneten Werte kann eine statistische Signifikanz der eventuell enthaltenen Saisonalitäten bestimmt werden.

2.2.3 Autoregressive-Moving Average (ARMA)-Modelle

Ein ARMA(p, q)- Modell setzt sich aus einem Autoregressiven AR(p) - und einem Moving Average MA(q)-Modell zusammen [6, S. 18]. Die Autoregressive Komponente berechnet zukünftige Werte

aus der Linearkombination von p historischen Messwerten, einem zufälligen Fehler ϵ und einer Konstante c. Mathematisch wird das wie folgt formuliert:

$$y_t = c + \sum_{i=1}^{p} \phi_i y_{t-i} + \epsilon_t \tag{2.1}$$

Die Werte ϕ_i $(i = 1, 2, ..., p)$ sind Modellparameter. Das MA(q)-Modell benutzt vergangene Fehler des Modells als erklärende Variable [6, S. 19] und ist wie folgt definiert:

$$y_t = \mu + \sum_{i=1}^{q} \Theta_i \epsilon_{t-i} + \epsilon_t \tag{2.2}$$

In dieser Definition beschreibt μ den Mittelwert über q gleitende Durchschnitte und Θ_i $(i = 1, 2, ..., q)$ sind Modellparameter. Die Fluktuationen werden als standardnormalverteilt angenommen. Konzeptuell entspricht das Moving-Average Modell also einer linearen Regression des aktuell betrachteten Messwertes gegen die Fluktuationen eines oder mehrerer früherer Messwerte(s) [6, S. 19]. Die Kombination der beiden Einzelmodelle liefert dann das gewünschte ARMA(p,q)-Modell, welches eine Vorhersage nach folgender mathematischen Vorschrift berechnet:

$$y_t = c + \epsilon_t + \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{i=1}^{q} \Theta_i \epsilon_{t-i} \tag{2.3}$$

2.2.4 Stationarität

Üblicherweise werden ARMA-Modelle mit dem Lag-Operator definiert [6, S. 19], welcher mathematisch als $Ly_t = y_{t-1}$ definiert ist. Somit lassen sich ARMA-Modelle mittels Lag-Polynomen wie folgt definieren:

$$AR(p) - Modell : \epsilon_t = \phi(L) y_t$$
$$MA(q) - Modell : y_t = \Theta(L) \epsilon_t$$

$$ARMA(p, q) - Modell : \phi(L) y_t = \Theta(L) \epsilon_t \tag{2.4}$$

In diesem Fall gilt:

$$\phi(L) = 1 - \sum_{i=1}^{p} \phi_i L_i \qquad \text{und} \qquad \Theta(L) = 1 + \sum_{j=1}^{q} \Theta_j L_j$$

Ein solches Modell wird als stationär bezeichnet, genau dann wenn die Modellparameter Zeitunabhängig sind [4, S. 160]. Ein ARMA(p,q)-Modell ist stationär, wenn für alle Wurzeln aus der charakteristischen Gleichung $\phi(L) = 0$ außerhalb des Einheitskreises liegen [6, S. 20]. ARMA-Modelle arbeiten nur auf stationären Zeitreihen zuverlässig.

Von Natur aus besitzen viele Zeitreihen ein nicht-stationäres Verhalten. Dies ist inhärent durch den Trend und die Saisonalität einer Zeitreihe gegeben. Aus diesem Grund können für den in dieser Arbeit verwendeten Datensatz keine einfachen ARMA-Modelle verwendet werden. Es gibt jedoch Erweiterungen dieser ARMA-Modelle, welche mit nicht stationären Zeitreihen

umgehen können. Eine dieser Erweiterungen stellen die ARIMA-Modelle dar, welche nachfolgend kurz beschrieben werden.

2.2.5 Autoregressive Integrated Moving Average (ARIMA)-Modelle

In ARIMA-Modellen wird die Stationarität der Zeitreihe durch finite-Differenzierung erreicht. Mathematisch ist ein ARIMA(p, d, q)-Modell wie folgt mit Hilfe der oben definierten Lag-Polynome definiert:

$$\phi(L)(1-L)^d y_t = \Theta(L)\epsilon_t$$

Das heißt ausformuliert:

$$\left(1 - \sum_{i=1}^{p} \phi_i L_i\right)(1-L)^d y_t = \left(1 + \sum_{j=1}^{q} \Theta_j L_j\right)\epsilon_t \tag{2.5}$$

- p, d und q sind dabei Integer ≥ 0 und bezeichnen die Ordnung der Autoregressiven- , Integrierten- und gleitenden Durchschnitts-Komponente des Modells.
- d beschreibt das Differenziationslevel. Im Umkehrschluss erhält man durch d-fache Integration die Werte der ursprünglichen Zeitreihe zurück.

Durch ein geeignetes ARIMA-Modell können auf Basis der vorhandenen Benzinpreise bereits Vorhersagen getroffen werden. Jedoch besteht der größte Nachteil dieser Modell-Klasse darin, dass Vorhersagen allein aus den historischen Daten generiert werden. Für die Lösung der Benzinpreisvorhersage sollen jedoch zusätzliche Parameter in das Vorhersagemodell eingebunden werden, um die allgemeine Vorhersagegüte zu verbessern. Aus diesem Grund wurden noch andere Modelle untersucht, mit deren Hilfe Feature-Vektoren mit zusätzlichen Parametern für die Benzinpreisvorhersage verwendet werden können. Diese Modelle nutzen Regressionsanalyse zur Vorhersage, welche nachfolgend beschrieben wird.

2.3 Regressionsanalyse

3. Implementierungsdetails

Zur Lösung der Aufgabenstellung wurde eine Software entwickelt, deren Aufbau im Folgenden beschrieben wird. Das Design und die Implementierung der Anwendung kann in die Teile Backend, Frontend, Fixed-Path-Gas-Station-Problem und Preisvorhersage unterteilt werden. Die einzelnen Teile werden in den nächsten Abschnitten jeweils erläutert.

3.1 Backend

Das Backend hat die Aufgabe, die benötigten Daten für das Frontend bereitzustellen. Zur Kommunikation zwischen Front- und Backend wird Hypertext Transfer Protocol (HTTP) in Verbindung mit dem Datenformat Javascript Object Notation (JSON) verwendet. Das Backend ist dabei nicht für die Generierung einzelner Teile des Frontends zuständig, wie es bei Technologien wie Java Server Faces (JSF) der Fall ist.

Diese Entkoppelung ermöglicht die Implementierung weiterer spezialisierter Frontends. So wäre zum Beispiel die Entwicklung eines Frontends für Mobilgeräte ohne große Anpassungen des Backends möglich.

Für die Umsetzung des Backends wurde die Programmiersprache Java gewählt. Java bietet viele Open Source Bibliotheken, die die Entwicklung der Problemlösung unterstützen und beschleunigen können. Neben der großen Auswahl an Bibliotheken, bietet Java sowohl ausreichende Performance als auch viele Werkzeuge, die den Entwicklungsprozess vereinfachen. Durch die Plattformunabhängigkeit ist es auch möglich, dass die Entwicklung auf jedem der im Team eingesetzten Betriebssysteme reibungslos und ohne spezielle Anpassungen möglich ist.

Im nächsten Abschnitt wird auf die Architektur des Backends eingegangen.

3.1.1 Architektur

Abbildung 3.1 zeigt die Architektur des Backends. Es werden die einzelnen Packages und ihre Abhängigkeiten untereinander aufgeführt. `Main` ist die einzige Klasse, die nicht in einem weiteren Sub-Package enthalten ist und wird deshalb separat aufgeführt.

Zur Aufgabe von `Main` gehört es, die Applikation zu starten und die Implementierungen der einzelnen Interfaces zu instanziieren. Durch Verwendung des Inversion Of Control (IOC)-Prinzips wird sichergestellt, dass die einzelnen Sub-Packages jeweils nur vom Interface abhängen und nicht von einer konkreten Implementierung des Interfaces. Die Abhängigkeiten einer Klasse werden zur Laufzeit per Constructor-Injection zur Verfügung gestellt. Auf die Verwendung eines Dependency Injection (DI)-Containers wurde verzichtet, da er bei dieser Programmgröße und -komplexität keine Vorteile bringt.

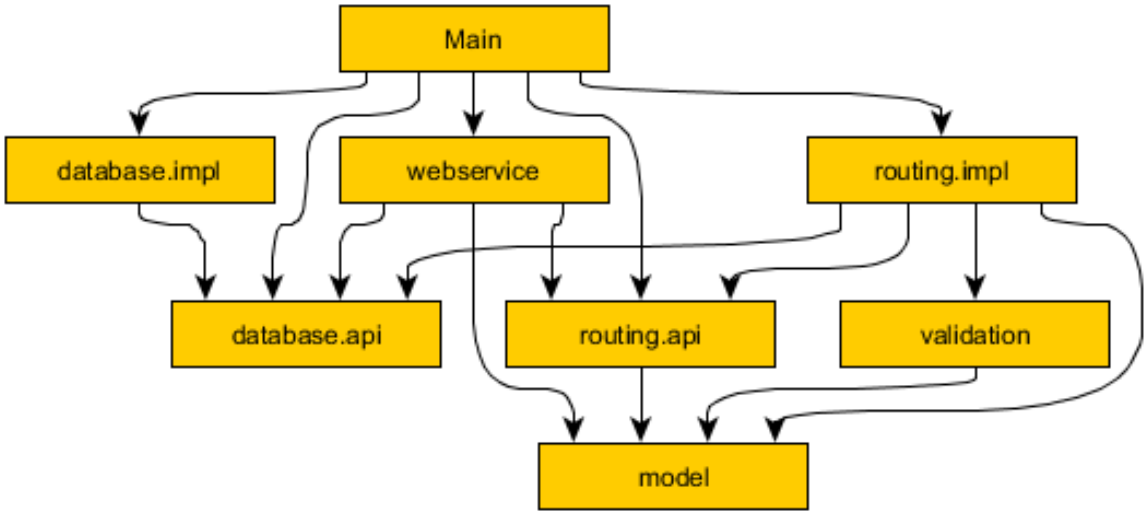

Abbildung 3.1: Abhängigkeiten der einzelnen Packages des Backends

Ein Nebeneffekt dieser Aufteilung ist die Möglichkeit, Unit Tests für die einzelnen Klassen zu schreiben ohne zum Beispiel eine aktive Datenbankverbindung zu benötigen. Über Mocking Bibliotheken kann in Tests statt einer konkreten Implementierung eines Interfaces ein Mock Objekt konstruiert und verwendet werden. Das Mock Objekt kann mit dem für den Test nötigen Verhalten konfiguriert werden.

Die folgenden Abschnitte erläutern jeweils die Aufgabe der einzelnen Packages.

model

Im Package `model` werden Klassen definiert, die für den Datenfluss im Programm notwendig sind. Die enthaltenen Klassen definieren die Daten der Problemdomäne (Domain Layer), der externen HTTP-Application Programming Interface (API) (Web Layer) und der Datenbank (Persistence Layer). Auf eine weitere Unterteilung der Klassen und Einteilung in die entsprechenden Layer wurde bewusst verzichtet. Der Vorteil einer Einteilung in die verschiedenen Layer hätte den Vorteil, dass sich die Datendefinitionen der einzelnen Layer unabhängig voneinander entwickeln können. Dies würde allerdings bedeuten, dass zwischen den einzelnen Layern eine Übersetzung stattfinden müsste. Diese Unterteilung könnte bei Bedarf vorgenommen werden.

Die Entscheidung für ein Anemic Domain Model wurde bewusst getroffen und soll eine spätere Einteilung der Klassen in die Layer erleichtern.

webservice

Das Package `webservice` enthält die Klassen `Router` und `ApiHandler`. `Router` ist für die Konfiguration der Bibliothek Spark zuständig. Spark ist ein Micro Framework zur Erstellung von Web Anwendungen. Da Spark alle Anforderungen der Anwendung in diesem Bereich erfüllen kann, wurde auf den Einsatz von Spring oder der JavaEE-Spezifikation in Verbindung mit einem Application Server verzichtet, um die Komplexität dieser Varianten zu vermeiden.

In `Router` werden unter anderem die Routen der HTTP-API mit den entsprechenden Methoden der Klasse `ApiHandler` verknüft. Des Weiteren wird für die lokale Entwicklung Cross-Origin Resource Sharing (CORS) aktiviert. Dies ermöglicht die Bereitstellung des Frontends während der Entwicklung von einem anderen Host (Webpack Dev Server) aus. Dadurch ist es nicht nötig, bei Änderungen des Frontends den Java-Build zu verwenden und verringert so die Feedback Loop.

Die Klasse `ApiHandler` ist für die Deserialisierung der Daten aus der Web Request zuständig. Die deserialisierten Daten werden anschließend an die Services des Domain Layers übergeben. Das Ergebnis dieser Operation wird schließlich wieder serialisiert. Sollten während der Bearbeitung der Anfragen Exceptions auftreten, werden diese an dieser Stelle auch in Instanzen der Klasse `ProblemResponse` umgewandelt. `ProblemResponse` richtet sich nach [7] und bietet so ein konsistentes Vorgehen zur Übermittlung von Fehlern des Backends.

routing

Das Package `routing`, mit seinen Sub Packages `api` und `impl`, ist für die Lösung der eigentlichen Aufgabe der Anwendung zuständig. Konkret bedeutet das, dass sowohl die Tankstrategien als auch die Preisvorhersagen hier berechnet werden. Durch die Trennung in `api` und `impl` ist es möglich, dass der Web Layer nicht von einer konkreten Implementierung der Logik abhängig ist. Bei der Implementierung dieses Layers wurde die Konvention verwendet, dass Fehler in den Eingabedaten durch Java's Checked Exceptions Mechanismus ein Bestandteil der API werden. Dies macht den Kontrollfluss explizit.

Der `SimpleRoutingService` verwendet das Strategy Pattern [8], um den verwendeten Algorithmus bei Bedarf durch einen anderen zu ersetzen.

Die Preisvorhersage verwendet ein Model der Bibliothek H2O. Dieses Model wird aus den `resources` geladen und beinhaltet das Ergebnis des Trainings. Mit dem Model wird ein Predictor instanziiert, der wiederum den Preis gemäß der Anfrage vorhersagt.

database

Das Package `database` definiert die Schnittstelle zur Datenbank und bietet eine konkrete Implementierung für eine Postgresql Datenbank an. Hierzu werden die Bibliotheken HikariCP und sql2o verwendet. Die Aufgabe von HikariCP ist es, einen Connection Pool für die Datenbank Verbindungen zur Verfügung zu stellen. Die Bibliothek sql2o bietet hingegen die Möglichkeit, das Ergebnis einer Datenbankabfrage in Instanzen der entsprechenden Klassen zu deserialisieren. Eine Besonderheit von sql2o ist die Verwendung von SQL Statements, die im jeweiligen Dialekt der Datenbank verfasst werden. Im Gegensatz zu Bibliotheken wie Hibernate bedeutet dies, dass zur Unterstützung einer neuen Datenbank sämtliche Queries angepasst werden müssen. Allerdings ermöglicht die Verwendung des Datenbank-spezifischen Dialekts auch eine Nutzung der Datenbank-spezifischen Features.

3.1.2 Persistenz

Als Relational Database Management System (RDBMS) wird Postgresql eingesetzt. Postgresql ist nicht nur Open Source, sondern verfügt auch über eine Erweiterung für Geodaten: PostGIS. Dadurch war es möglich, die vorhandenen Daten über Tankstellen mit weiteren Daten zu erweitern, um möglichst viele Features für die Preisvorhersage zu erhalten.

Das genaue Vorgehen zur Ermittlung der Daten wurde in der Datei `orga.org` genau dokumentiert. Damit diese recht zeitintensiven Schritte nicht wiederholt werden müssen, liegt diesem Dokument eine Sicherung der kompletten Datenbank bei.

Im laufenden Betrieb sind nur wenige Tabellen der Datenbank notwendig. Die Tabelle `stations` ist die wichtigste Tabelle, denn sie enthält die Daten zu den einzelnen Tankstellen. Tabelle 3.1 zeigt die verwendeten Spalten.

Spalte	**Type**	**Bemerkungen**
id	smallint	Id wie in Eingabedaten
lat	double	Breitengrad in WGS84
lon	double	Längengrad in WGS84
station_name	varchar(255)	Name
street	varchar(255)	Adresse
house_number	varchar(255)	Hausnummer
zip_code	varchar(5)	PLZ
city	varchar(255)	Ort
brand	varchar(255)	Marke
brand_no	int	Id der Marke
bland_no	int	Id des Bundeslands
kreis	varchar(12)	Verwaltungsschlüssel des Landkreises
abahn_id	int	Id der nächsten Autobahn
bstr_id	int	Id der nächsten Bundesstraße
sstr_id	int	Id der nächsten Schnellstraße

Tabelle 3.1: Im Backend verwendete Spalten der Tabelle stations

Neben den Informationen, die bereits in den Daten von Tankerkönig enthalten waren, wurden die Tankstellen mit weiteren Daten versehen. Ein großer Teil der zusätzlichen Daten wurde aus Daten von Open Street Map (OSM) gewonnen. Auf diese Daten wird im nächsten Abschnitt genauer eingegangen.

Geodaten

Um weitere Features der Tankstellen zu gewinnen, wurden die OSM Daten von Deutschland verwendet. Hierzu wurden Daten zu Autobahnen, Bundes- und Schnellstraßen und den Verwaltungsgrenzen aus dem OSM Datensatz extrahiert.

Die Informationen zu nahegelegenen Autobahnen, Bundes- und Schnellstraßen wurden mithilfe der Hilfstabellen `bundesstr`, `autobahn` und `schnellstr` ermittelt. Zuerst wurden die entsprechenden Daten aus dem OSM Datensatz in diese Tabellen geschrieben. Anschließend wurden die enthaltenen Geometrien von der Projektion `EPSG:4326` (`WGS84`) in `EPSG:4839` umgewandelt. Auch die Punktgeometrien der Tankstellen wurden konvertiert. Funktionen, die mit der Distanz zwischen zwei Geometrien arbeiten, verwenden in PostGIS immer die Einheit der Projektion. Die Einheit von `EPSG:4326` ist Grad und somit für die Berechnung von Distanzen in Metern ungeeignet. `EPSG:4326` hingegen verwendet Meter und hat eine Genauigkeit von etwa 1 Meter im betrachteten Bereich. Dies ist für die Anforderungen ausreichend. Auf den Geometriespalten wurde jeweils ein Spatial Index erstellt, um die Suche nach nahegelegenen Geometrien zu beschleunigen. Anschließend wurden die Tankstellen mit den Geometrien der

verschiedenen Straßen gejoined. Als Bedingung für den Join wurde eine Distanz kleiner 5000 Meter verwendet. Dies bedeutet, dass die entsprechende Spalte in der Tabelle `stations` den Wert `null` aufweist, falls sich kein Teil einer Straße in mindestens 5 km Nähe befindet.

Um die Zugehörigkeit zu einem Landkreis und Bundesland festzustellen, wurden die Informationen über Gemeindegrenzen aus den OSM Daten extrahiert und in die Hilfstabellen `bundeslaender` und `kreise` gespeichert. Die enthaltenen Geometrien wurden mit der Funktion `ST_polygonize` in Polygone umgewandelt. Dieser Schritt ist notwendig, um mit der Funktion `ST_contains` zu ermitteln, ob eine Tankstelle in einem Bundesland bzw. Kreis liegt. Auch hier wurde die Abfragegeschwindigkeit durch die Verwendung von Spatial Indices erhöht.

3.1.3 Tests

Um während der Entwicklung die Korrektheit des Programms sicherzustellen, wurden Tests implementiert. Es wurden Unit Tests implementiert, um komplexere Klassen zu testen. Unit Tests enden mit dem Suffix `Test`.

Neben den Unit Tests wurden mehrere Integration Tests implementiert. Diese Integration Tests enden mit dem Suffix `IT`. Die Integration Tests erfordern eine aktive Datenbank Verbindung und werden deshalb während des Builds nicht ausgeführt. Um die Tests mit Maven zu starten, kann das Maven Profil `it` aktiviert werden. Dies kann mit dem Kommandozeilenargument `-P it` erreicht werden.

Um den verwendeten Algorithmus gegen die naive Tankstrategie zu benchmarken kann die Klasse `FixedVsNaiveIT` verwendet werden. Diese Klasse instanziiert den `ApiHandler` zwei mal und verwendet jeweils eine unterschiedliche Tankstrategie. Anschließend wird der Gesamtpreis verglichen.

3.2 Frontend

Das Frontend stellt die Schnittstelle der Anwendung zum Benutzer dar. Ein Benutzer kann über das Frontend die verschiedenen Comma Separated Values (CSV) Dateien einlesen. Die Daten werden anschließend zur Bearbeitung an das Backend gesendet. Das Ergebnis wird schließlich im Frontend dargestellt.

3.2.1 Design

Abbildung 3.2 zeigt die Ansicht, die der Benutzer beim ersten Aufruf der Anwendung sieht. Der Bereich ist in zwei Teile aufgeteilt. Oben wird der Titel des Programms angezeigt, sowie ein Button `Menu` angeboten, um das seitliche Menü auszuklappen. In diesem Zustand ist das Menü leer. Der zweite Bereich besteht aus einer Kartenansicht und einem Button `Actions`. Die Karte soll beim Start auf die aktuelle Position des Benutzers, falls vorhanden, zentriert werden. Ansonsten erfolgt eine Zentrierung auf Karlsruhe. Der Button `Actions` ist der zentrale Anlaufpunkt für den Benutzer, um die Anwendung zu verwenden. Ein Click auf den Button lässt zwei weitere Buttons erscheinen, um Dateien zur Ermittlung einer optimalen Route bzw. einer Preisvorhersage an die Anwendung zu übermitteln. Dieser Zustand wird in Abbildung 3.3 gezeigt.

Neben den Buttons zur Übermittlung von Dateien, zeigt diese Abbildung auch das seitliche Menü mit Inhalt. Der Button « dient dazu das Menü wieder zu schließen. Wenn der Benutzer eine Route anfragt, wird das Ergebnis im Seitenmenü dargestellt. Dazu wird ein Abschnitt Routen erstellt, der sich bei Bedarf einklappen lässt. Dieser Bereich enthält für jede einzelne Anfrage des Benutzers das jeweilige Ergebnis in Tabellenform. Unter der Tabelle wird jeweils ein Button zum Download der Ergebnis-CSV und zum Löschen der Route angeboten. Eine Route wird auf der Karte durch Punkte für die angefahrenen Tankstellen angezeigt. Diese Punkte sind miteinander verbunden. Der Abschnitt zur Anzeige von Preisvorhersagen soll dazu analog funktionieren. Allerdings werden die angefragten Tankstellen als einzelne Punkte angezeigt. Am unteren Bildschirmrand ist die Anzeige eines Fehlers zu sehen. Es wird dem Benutzer eine kurze Information gegeben, die er mit einem Click auf den Button `x` schließen kann.

3.2.2 Implementierung

Zur Implementierung des Frontends wurde auf Web-Technologien gesetzt. Dies erlaubt eine kurze Feedback Loop bei der Entwicklung. Zur Entwicklung wurde die Sprache TypeScript verwendet. Bei TypeScript handelt es sich um ein Superset von Javascript, das um ein starkes Typsystem erweitert wurde. Durch die starke Typisierung ist es möglich, bereits während der Entwicklung Fehler zu erkennen, die bei der Verwendung von JavaScript erst zur Laufzeit auftreten.

Zu den nennenswerten eingesetzten Bibliotheken zählen OpenLayers 4 zur Darstellung von Karten, vue.js als Framework zur Erstellung von Single Page Applications (SPAs) und vue-material als Komponenten Bibliothek.

Die Anwendung wurde in vue.js Komponenten unterteilt. Diese Komponenten befinden sich im Ordner `frontend/vue` und bestehen jeweils aus einer `.html` und einer `.ts` Datei. Die `.ts` Datei definiert das Verhalten der Komponente und die `.html` Datei das Aussehen.

Die Hauptkomponente, die die untergeordneten Komponenten instanziiert und für die Orchestration der Anwendung zuständig ist, befindet sich in den Dateien `index.ts` und

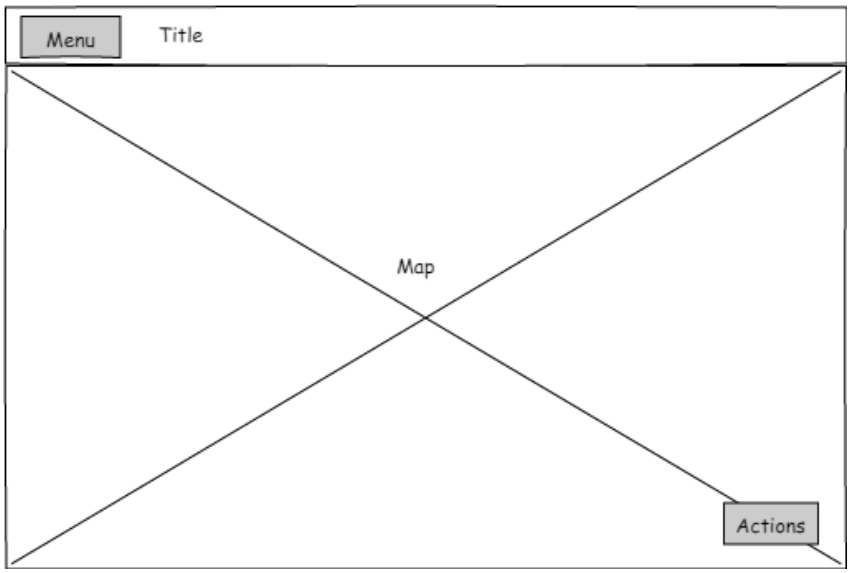

Abbildung 3.2: Start Ansicht des Frontends

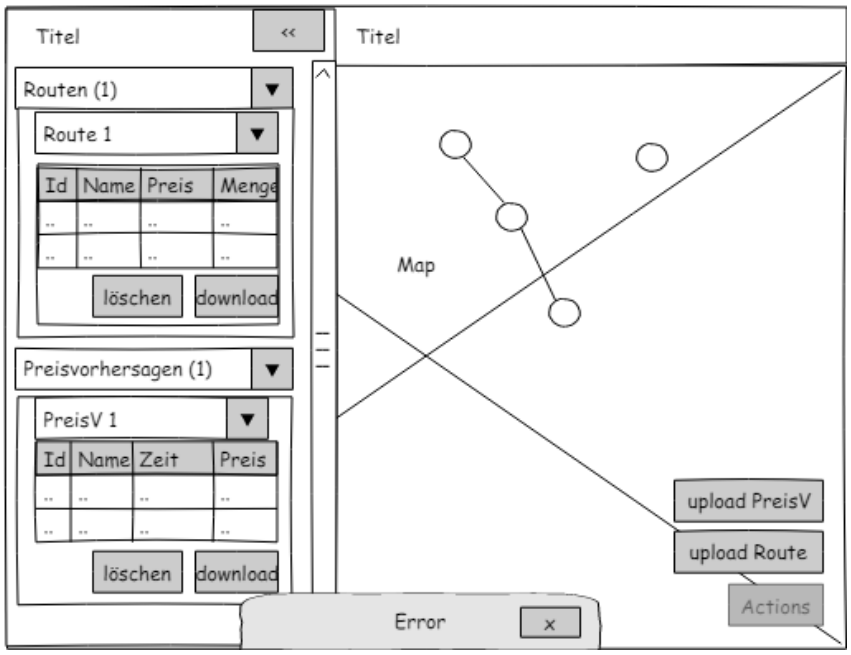

Abbildung 3.3: Frontend während Verwendung

`index-template.html`. Funktionalitäten, die nicht die Darstellung der Benutzeroberfläche beeinflussen, wurden im Ordner `frontend/app` abgelegt.

Die Darstellung der Karte verwenden OpenLayers 4. Die angezeigte Basiskarte ist von OSM und erfordert eine aktive Internetverbindung während der Verwendung der Anwendung.

Während der Entwicklung der Anwendung kann der Webpack Dev Server mit dem Script `start` über das Tool yarn gestartet werden. Der Dev Server erstellt das Frontend nach Änderungen am Quellcode neu und triggert einen Reload des Browsers, falls nötig. Für die Auslieferung der Anwendung startet der Maven Build das Script `build` und kopiert die resultierenden Dateien in den Ordner `resources/public`. Spark ist so konfiguriert, dass der Inhalt dieses Ordners als statischer Inhalt zur Verfügung steht.

3.2.3 Tests

Für das Frontend wurden keine automatisierten Tests implementiert. Aufgrund der Überschaubarkeit der Anwendung wurde hierbei auf ein manuelles Testprotokoll zurückgegriffen. Das Testprotokoll beinhaltet folgende Schritte:

1. Frontend öffnen
2. Action Route Uploaden aufrufen
3. Prüfen, ob Karte auf Route zoomt
4. Prüfen, ob Ergebnis in seitlichem Menü angezeigt wird
5. Action Preisvorhersage Uploaden aufrufen
6. Prüfen, ob Karte auf angefragte Tankstellen zoomt
7. Prüfen, ob Ergebnis in seitlichem Menü angezeigt wird
8. Weitere Route hinzufügen
9. Erste Route löschen
10. Prüfen, ob erste Route nicht mehr in Ergebnisliste ist
11. Weitere Preisvorhersage hinzufügen
12. Erste Preisvorhersage löschen
13. Prüfen, ob erste Route nicht mehr in Ergebnisliste ist
14. Action Route Downloaden aufrufen
15. Prüfen, ob CSV korrekt ist
16. Action Preisvorhersage Downloaden aufrufen
17. Prüfen, ob CSV korrekt ist

Merkmal	Ursprung
Feiertage	`holidays` Python Package [9]
Schulferien	Schulferien.org [10]
Bundesland	OpenStreetMap [11]
Landkreis	OpenStreetMap [11]
nächste Hauptverkehrsstraße	OpenStreetMap [11]

Tabelle 3.2: Auflistung der zusätzlich verwendeten Merkmale pro Benzinpreis und Tankstelle.

3.2.4 Einschränkungen

Während der Entwicklung wurde der aktuellste Chrome Browser von Google verwendet. Andere Browser werden momentan nicht unterstützt. Zum Zeitpunkt der Erstellung dieser Dokumentation, ist zumindest ein Bug in Firefox bekannt, der das Downloaden von Routen verhindert.

3.3 Datenverarbeitung / Preisvorhersage

In diesem Kapitel werden unsere Entscheidungen bezüglich der verwendeten Lernalgorithmen und der Datenverarbeitung dokumentiert.

3.3.1 Feature Engineering

Der erste Vorbereitungsschritt war, die vorhandenen Daten auf eine einheitliche Frequenz zu bringen. Um dies zu erreichen, wurde eine Datenbanktabelle mit stündlichen Zeitstempeln von Beginn der Aufzeichnung an bemustert. Anschließend wurde für jeden Zeitstempel ein Preis $p(t)$ nach folgendem Kriterium eingetragen:

$$p(t) = \begin{cases} p_{neu}, & \text{falls für } [t, t+1] \text{ ein neuer Preis } p_{neu} \text{ vorhanden ist} \\ p(t-1), & \text{sonst} \end{cases}$$

Dieses Grundgerüst eines Feature-Vektors wurde um die in Tabelle 3.2 aufgeführten zusätzlichen Merkmale erweitert. Durch die Persistierung der Daten in einer Datenbank kann effizient darauf zugegriffen werden. Bei den vorhandenen Datenmengen wäre ein Verbund mehrerer Datenbanktabellen zu zeitaufwendig, weshalb die redundante Speicherung mancher Daten in Kauf genommen wird. Für das Training des Regressionsalgorithmus wird aus dem Datensatz eine Trainingsmenge mit 75% des gesamten Datensatzes erstellt. Die restlichen 25 % ergeben die Testmenge zur Validierung unseres Modells.

Feature Importance

Mit dem eingesetzten Analysewerkzeug H2O können die für das Training verwendeten Features hinsichtlich ihrer Relevanz für die gegebene Problemstellung untersucht werden. Die Ergebnisse dieser Analyse sind in Abbildung 3.4 grafisch dargestellt. Interessant ist hierbei, dass das Jahr der Messung das wichtigste Merkmal für die Vorhersage von Benzinpreisen darstellt. Diese Tatsache ist auf die heftige Schwankung des Benzinpreises in den letzten Jahren zurückzuführen. In Abbildung 3.5 sind die Jahresdurchschnittspreise für einen Liter Super Benzin der letzten 6 Jahre zu sehen. In dieser Grafik ist zu erkennen, dass der Benzinpreis im Jahr 2012 noch sehr

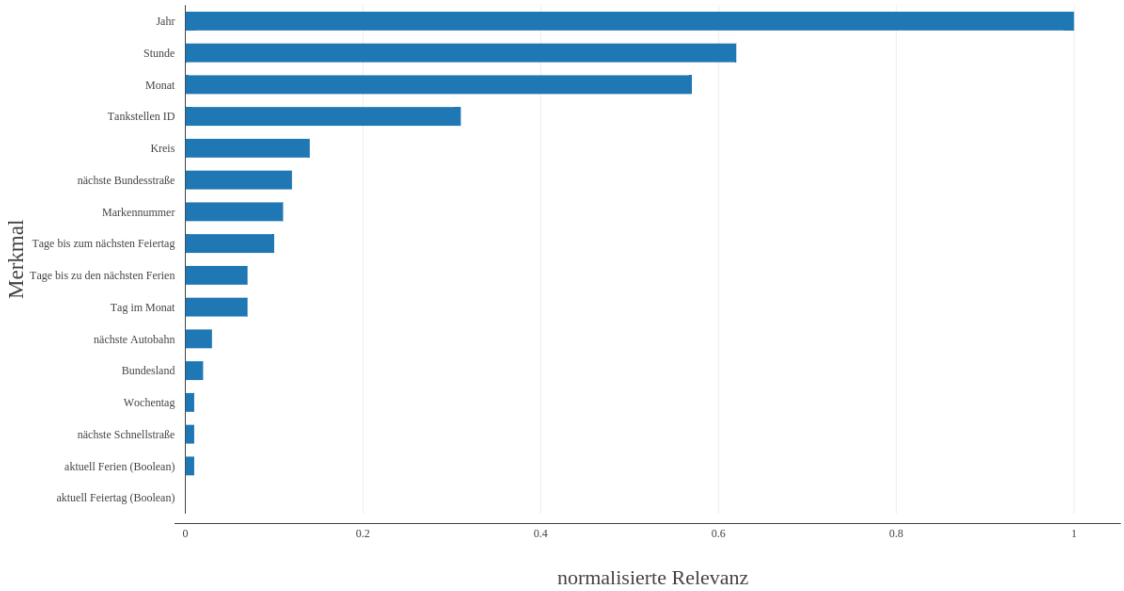

Abbildung 3.4: Relative Relevanz der einzelnen Merkmale für die Vorhersage von Benzinpreisen.

hoch war, wohingegen er von 2014 bis 2016 stetig gesunken ist. Die Frankfurter Allgemeine Zeitung führt dies auf die geringe Nachfrage nach Rohöl in den Jahren 2014 und 2015 zurück, sowie auf die Einführung von Fracking in den USA, welche den Rohölpreis deutlich sinken ließ [12]. Jedoch ist auch die Tageszeit, die ID der entsprechenden Tankstelle und der Landkreis wichtig für die Vorhersage des Benzinpreises.

3.3.2 Modellauswahl

Als erster Ansatz wurden unterschiedliche ARMA-Modelle untersucht. Diese lassen jedoch keine Verwendung exogener Variablen zu, welche, wie später gezeigt, eine hohe Relevanz für die Vorhersage von Benzinpreisen besitzen. Anschließend wurden unterschiedliche maschinelle Lernverfahren untersucht, darunter auch einfache Deep-Learning Ansätze. Die manuelle Untersuchung dieser Algorithmen beansprucht jedoch viel Zeit. Um nicht unter Zeitdruck zu geraten, wurde die Modellsuche mit Hilfe der Open-Source Software H2O automatisiert. H2O implementiert Algorithmen aus den Bereichen Statistik, Data Mining und Maschinellem Lernen. Die Software basiert auf dem Hadoop Distributed File System, sodass ein Performance-Gewinn gegenüber anderen Analysewerkzeugen erzielt wird.

H2O kann mehrere Modelle des gleichen Typs mit anderen Modellen vergleichen und so das beste Modell für die gegebene Problemstellung finden. Durch ein Grid-Search werden die Hyperparameter der Modelle optimiert. Somit ist es möglich, in relativ kurzer Zeit ein wettbewerbsfähiges Modell zu erstellen. Für die gegebene Problemstellung hat sich ein Gradient Boosting Machine (GBM) Modell als besonders potent herausgestellt. Die Eigenschaften dieses Modells werden nachfolgend kurz erläutert.

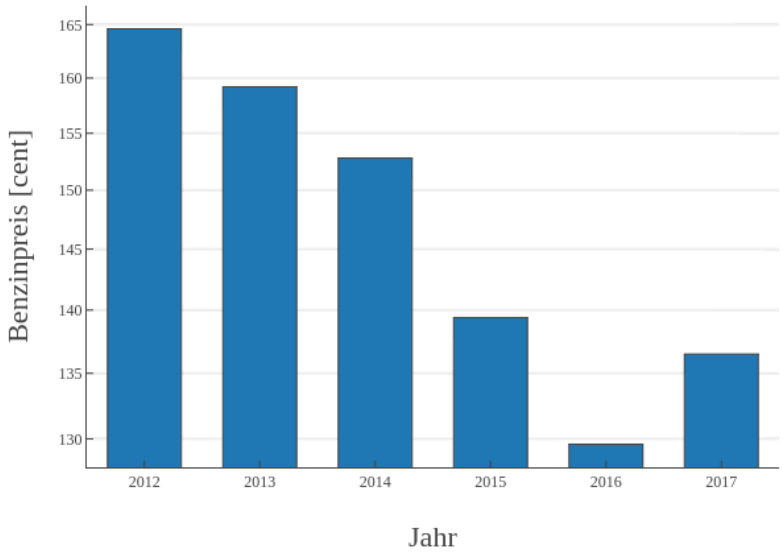

Abbildung 3.5: Entwicklung der Jahresdurchschnittspreise für einen Liter Super Benzin in den Jahren 2012 bis 2017. Die Daten stammen von statista.com [13].

Gradient Boosting Machine

Die Erklärung der Gradient Boosting Machines erfolgt nach Friedmann [14]. Durch Gradient Boosting wird ein Ensemble aus mehreren einfachen Modellen gebildet, welche als Aggregat ein neues und stärkeres Modell bilden. Die beinhalteten Modelle werden dabei so gebildet, dass der Gesamtfehler aller Modelle, inklusive des neu erstellten Modells, minimiert wird. Dies geschieht über den Gradienten der Fehlerfunktion unter Berücksichtigung der Vorhersage. Der Algorithmus funktioniert dabei wie folgt:

1. Ein Modell, üblicherweise ein Entscheidungsbaum, wird auf die Daten angepasst:

$$F_1(x) = y$$

2. Ein zweites Modell wird auf die Restwerte des ersten Modells angepasst:

$$h_1(x) = y - F_1(x)$$

3. Aus diesen Modellen wird additiv ein neues Modell gebildet:

$$F_2(x) = F_1(x) + h_1(x)$$

Die grundlegende Idee dieses Boosting-Verfahrens ist es also, immer neue Modelle hinzuzufügen, die den Fehler des vorherigen Modells korrigieren. Das Modell wird über die Minimierung einer definierten Fehlerfunktion $L(y, F(x))$ trainiert. Diese Optimierung geschieht über ein Gradientenverfahren. Es ergeben sich folgende Aktualisierungsregeln:

$$F_m(x) = F_{m-1}(x) - \gamma_m \sum_{i=1}^{n} \nabla_{F_m(x)} L(y_i, F_{m-1}(x_i))$$

Wobei γ_m die Lernrate des aktuell betrachteten Modells repräsentiert. Diese Lernrate wird adaptiv nach folgender Vorschrift angepasst:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, F_{m-1}(x) - \gamma \nabla_{F_m(x)} L(y_i, F_{m-1}(x_i)))$$

Ergebnisse der Grid Search

Die Hyperparameter der implementierten GBM wurden durch eine Grid Search optimiert. Tabelle 3.3 gibt eine Übersicht über die wichtigsten Hyperparameter-Werte des Modells.

Parameter	**Wert**
Anzahl an Entscheidungsbäumen	50
Anzahl interner Bäume	50
Minimale Tiefe	5
Maximale Tiefe	5
Minimale Anzahl an Blättern	32
Maximale Anzahl an Blättern	32
K-fold cross validation	K=5
Lernrate	0.1

Tabelle 3.3: Die Hyperparameter-Werte des GBM-Modells, welche durch die Grid-Search optimiert wurden.

3.3.3 Training

Da die ursprünglich als Zeitreihenvorhersage gedachte Aufgabenstellung durch eine Regressionsaufgabe substituiert werden konnte, trainiert das implementierte Modell mit dem gesamten Datensatz. Dies ist insbesondere für Preisvorhersagen, welche weit außerhalb der vorhandenen Datenbasis liegen von Vorteil. Der größte Nachteil an dieser Methode ist die benötigte Rechenpower. Die Trainingsmenge umfasst beinahe 15 Gigabyte. Da H2O den Datensatz komplett in den Arbeitsspeicher lädt, wird für das Training eine dementsprechend starke Rechenmaschine benötigt. Um dieses Modell zu trainieren, wurde daher auf eine Deep Learning Workstation mit 64 Gigabyte Arbeitsspeicher zurückgegriffen.

Das Training des Modells erfolgte in einer fünffachen Kreuzvalidierung und dauerte etwas mehr als drei Stunden. Im Mittel besitzt das Modell noch einen Root Mean Squared Error (RMSE) von 42.5 zehntel Cent auf den Testdaten. Dieser Wert ist durchaus passabel, da das Modell Preise unabhängig von deren vorheriger Entwicklung vorhersagen kann. In anderen Worten bedeutet dies, dass wir einen konstanten Fehler auf unseren Vorhersagen haben, welcher sich nicht bei schrittweisen Vorhersagen aufaddiert.

3.4 CLI

Zum automatisierten Vergleich der Ergebnisse der Anwendung mit den Ergebnissen vergleichbarer Anwendungen wurde ein Command Line Interface (CLI) implementiert. Dieses CLI benötigt Python 3.6 und ist im Unterordner `CLI` enthalten. Der Aufruf für die Berechnung einer Tankstrategie ist wie folgt:

```
python cli.py --input pfad_zu_csv --output pfad_zu_output \
--host "http://localhost:4567" --type route
```

Für den Abruf einer Preisvorhersage kann Folgendes verwendet werden:

```
python cli.py --input pfad_zu_csv --output pfad_zu_output \
--host "http://localhost:4567" --type pred
```

4. Besonderheiten / Vorteile bei bestimmten Routen / Preisvorhersagen

Dieses Kapitel geht auf Besonderheiten der Anwendung ein und schlägt Routen und Preisvorhersagen vor, die diese Besonderheiten hervorheben. Sämtliche vorgeschlagenen Routen und Preisvorhersagen sind im Ordner `routingService/src/test/resources/com/github/robinbaumann/informaticup2018` als CSV Dateien enthalten. Die folgenden Abschnitte beschreiben jeweils in Tabellenform solche Daten.

Sämtliche hier beschriebenen Datensätze werden in den Integration Tests verwendet, um die Korrektheit des Verhaltens sicherzustellen.

4.1 Fehlerfälle

4.1.1 Nicht existierende Tankstellen

Typ	Tankstrategie
Dateiname	bertha_bogus_station.csv
Besonderheit	enthält eine Tankstellen Id, die es nicht gibt
Verhalten	Anwendung antwortet mit entsprechender ProblemResponse, kein Absturz

4.1.2 Leere Tankstrategie

Typ	Tankstrategie
Dateiname	bertha_empty.csv
Besonderheit	enthält nur die Kapazität
Verhalten	Anwendung antwortet mit entsprechender ProblemResponse, kein Absturz

4.1.3 Negative Tankkapazität

Typ	Tankstrategie
Dateiname	bertha_negative_capacity.csv
Besonderheit	die Tankkapzität hat einen negativen Wert
Verhalten	Anwendung antwortet mit entsprechender ProblemResponse, kein Absturz

4.1.4 Tankstops sind nicht in zeitlicher Reihenfolge

Typ	Tankstrategie
Dateiname	bertha_out_of_order.csv
Besonderheit	zwei Tankstops wurden vertauscht, die Reihenfolge in der CSV entspricht nicht der zeitlichen Reihenfolge
Verhalten	Anwendung antwortet mit entsprechender ProblemResponse, kein Absturz

4.1.5 Routen mit Tankstellen ohne Preisinformationen

Typ	Tankstrategie
Dateiname	bertha_stations_without_prices.csv
Besonderheit	Es werden Tankstellen angefragt für die in den Originaldaten keine Preise hinterlegt waren
Verhalten	Anwendung antwortet mit entsprechender ProblemResponse, kein Absturz

4.1.6 Daten nach angefragtem Zeitpunkt verfügbar

Typ	Preisvorhersage
Dateiname	price-prediction-historic.csv
Besonderheit	Es wird ein Preis angefragt und es können Daten bis nach dem angefragten Zeitpunkt verwendet werden
Verhalten	Anwendung antwortet mit dem tatsächlichen Preis zu dem Zeitpunkt

4.1.7 Daten bis kurz vor angefragtem Zeitpunkt verfügbar

Typ	Preisvorhersage
Dateiname	price-prediction-very-close.csv
Besonderheit	Es wird ein Preis angefragt und es können Daten bis kurz vor (1h) dem angefragten Zeitpunkt verwendet werden
Verhalten	Anwendung antwortet letztem tatsächlichen Preis zu dem Zeitpunkt

4.1.8 Routen Stops nicht erreichbar mit vollem Tank

Typ	Tankstrategie
Dateiname	not-reachable.csv
Besonderheit	An einer gewissen Position in der Route, ist der nächste Stopp selbst mit vollem Tank nich
Verhalten	Anwendung antwortet mit entsprechender ProblemResponse, kein Absturz

4.2 Evaluation

Zusätzlich zu den oben genannten Routen wurden noch vier weitere erstellt zur Evaluierung der hier implementierten Tankstrategie gegen die oben definierte naive Tankstrategie. Es werden

nun vier Routen verglichen anhand der tatsächlichen Kosten, die nach den zwei Tankstrategien jeweils anfallen würden. Die Euro-Beträge sind auf die zweite Dezimalstelle und die Prozentwerte auf eine ganze Zahl gerundet.

Typ	Tankstrategie
Dateiname	test1.csv
Naive Strategie	50.96 €
Optimale Strategie	25.70€
Günstiger	25.26€
% günstiger	50%

Typ	Tankstrategie
Dateiname	test2.csv
Naive Strategie	138.08€
Optimale Strategie	56.02€
Günstiger	82.06€
% günstiger	40%

Typ	Tankstrategie
Dateiname	test3.csv
Naive Strategie	118.01€
Optimale Strategie	68.17€
Günstiger	49.84€
% günstiger	57%

Typ	Tankstrategie
Dateiname	test4.csv
Naive Strategie	46.36€
Optimale Strategie	28.08€
Differenz	18.28€
% günstiger	60%

Im Schnitt ist bei unseren vier Testversuchen die optimale Tankstrategie 51.75% günstiger als die naive.

5. Ausblick

Die vorgestellte Lösung hat seinen größten praktischen Nutzen sicherlich im Fernverkehr, insbesondere im Güterfernverkehr. Bei letzterem werden Lastkraftfahrzeuge eingesetzt, deren Tankkapazität mehrere hundert Liter umfasst. Durch die Benzinpreis-optimierte Routenplanung können hier erhebliche Kosten gespart werden. Ähnliches gilt für den Personenfernverkehr mit Bussen. Durch die eingesparten Spritkosten können Fahrkarten günstiger verkauft werden, was wiederum eventuell neue Kundschaft anlockt und somit in einer verstärkten Nutzung von Reisebussen resultiert. Dies hätte zusätzlich einen positiven ökologischen Nebeneffekt im Hinblick auf den Klimawandel.

Die Planung des Bundesverkehrsministeriums zur Installation von 5000 Ladestationen für Elektroautos [15] wirft die Überlegung in den Raum, ob Intellitank auch hierauf anwendbar ist. Mit dieser Investition soll auch ein einheitliches Bezahlsystem eingeführt werden. Laut einer Untersuchung der ISPEX AG [16] folgt der Strompreis in seiner Entwicklung dem Ölpreis. Dies passt sehr gut in das Konzept von Intellitank. Probleme bereiten jedoch kostenlose Ladestationen für Supermarktkunden und die insgesamt noch limitierte Reichweite von Elektroautomobilen. Da der Besitzer sein Fahrzeug zu Hause aufladen kann, benötigt er für den Alltagsverkehr nicht zwingend ein Planungssystem für optimale Tankstrategien. Erst die flächendeckende Infrastruktur für Ladestationen macht Intellitank interessant für Besitzer von Elektrofahrzeugen, da sie somit auch über weite Strecken mit ihrem Fahrzeug reisen können.

Ein weiteres spannendes Einsatzgebiet für Intellitank bietet das Forschungsgebiet des Autonomous Driving. Ein selbstfahrendes Auto kann dann die Route Benzinpreis-optimal berechnen, so dass der Mitfahrer nur noch einsteigen muss und direkt losfahren kann.

A. Anleitungen

Installationsanleitung

Für die Installation wird von einem Linux System ausgegangen. Die Schritte sind für andere Betriebssysteme entsprechend anzupassen.

Zur Installation wird eine .zip-Datei namens 'RoutingService.zip' benötigt. Diese beinhaltet das Backend- und das Frontend-Setup, sowie eine fertig kompilierte Jar Datei.

Zusätzlich zu der .ZIP-Datei liegt anbei ein .sql-Dump.gz, welcher auf eine PostgreSql-Datenbank importiert werden muss. Die Anwendung wurde mit Postgresql 9.6.5 und PostGIS 2.3.3 getestet. Um die Datenbank aufzuspielen sind folgende Schritte notwendig.

```
gunzip infocup.sql-dump.gz
psql infocup < infocup.sql-dump
```

Die Datenmenge ist recht groß. Deshalb sollte die Standardkonfiguration von Postgresql entsprechend angepasst werden.

Um die Jar Datei zu bauen wird das JDK8 und Maven benötigt. Während des Builds lädt Maven sowohl NodeJs als auch den Paket-Manager yarn herunter.

Um das Projekt zu generieren wird in den Ordner 'RoutingService' gewechselt und folgender Befehl ausgeführt:

```
mvn clean install
```

Im Ordner *target* liegt nach erfolgreichem Build-Prozess die jar-Datei *RoutingService-1.0-SNAPSHOT.jar* vor.

Bedienungsanleitung

Um mit der Anwendung zu arbeiten, ohne selbst eine Datenbank aufzuspielen, kann die Demo Instanz genutzt werden. Diese ist unter `https://infocup.noshak.de` erreichbar. Der Benutzer `infocup` und das Passwort `predictp` werden für den Zugang benötigt.

Sofern die Installation aus vorherigem Abschnitt erfolgreich beendet wurde, kann man das Projekt mittels

```
java -jar RoutingService-1.0-SNAPSHOT.jar \
  -D infocup.host="localhost:5432" \
  -D infocup.user="infocup"
```

starten. Die übergebenen Properties `infocup.host` und `infocup.user` konfigurieren die Verbindung zur Datenbank und sind entsprechend anzupassen.

Wie in Abbildung A.1 zu sehen ist, gibt das Startup-Log die Adresse an, auf die der Server lauscht. In diesem Fall *0.0.0.0:4567*, also abrufbar über *http://localhost:4567*.

```
tom@pils ~/PycharmProjects/InformatiCup2018/routingService/target master java -jar routingSer
vice-1.0-SNAPSHOT.jar
[main] INFO spark.staticfiles.StaticFilesConfiguration - StaticResourceHandler configured with folder =
/public
[main] INFO com.zaxxer.hikari.HikariDataSource - HikariPool-1 - Starting...
[main] INFO com.zaxxer.hikari.pool.PoolBase - HikariPool-1 - Driver does not support get/set network tim
eout for connections. (Die Methode org.postgresql.jdbc.PgConnection.getNetworkTimeout() ist noch nicht i
mplementiert.)
[main] INFO com.zaxxer.hikari.HikariDataSource - HikariPool-1 - Start completed.
[Thread-2] INFO org.eclipse.jetty.util.log - Logging initialized @470ms to org.eclipse.jetty.util.log.Sl
f4jLog
[Thread-2] INFO spark.embeddedserver.jetty.EmbeddedJettyServer - == Spark has ignited ...
[Thread-2] INFO spark.embeddedserver.jetty.EmbeddedJettyServer - >> Listening on 0.0.0.0:4567
[Thread-2] INFO org.eclipse.jetty.server.Server - jetty-9.4.z-SNAPSHOT
[Thread-2] INFO org.eclipse.jetty.server.session - DefaultSessionIdManager workerName=node0
[Thread-2] INFO org.eclipse.jetty.server.session - No SessionScavenger set, using defaults
[Thread-2] INFO org.eclipse.jetty.server.session - Scavenging every 660000ms
[Thread-2] INFO org.eclipse.jetty.server.AbstractConnector - Started ServerConnector@42aa56e7{HTTP/1.1,[
http/1.1]}{0.0.0.0:4567}
[Thread-2] INFO org.eclipse.jetty.server.Server - Started @546ms
```

Abbildung A.1: Server-Log beim Starten des Projekts

Die Web-Anwendung bietet zwei wesentliche Funktionen.
Im unteren rechten Eck (siehe Abbildung A.2) gibt es einen Button, der per Klick dem Benutzer die Möglichkeit eröffnet, entweder Preise vorherzusagen oder eine optimale Route berechnen zu lassen.

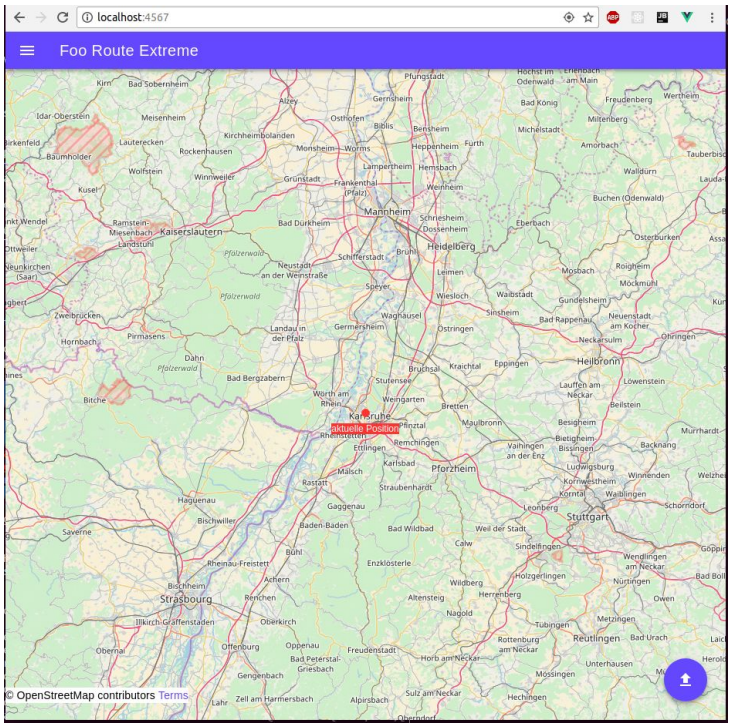

Abbildung A.2: grundlegende Oberfläche der Webanwendung

Die Buttons in Abbildung A.3 öffnen in beiden Fällen bei Klick einen File-Dialog zum Hochladen einer CSV-Datei. Beim Klick auf den oberen Button besteht die Möglichkeit eine Tabelle, wie beispielsweise die Bertha-Benz-Memorial-Route, zu öffnen um eine optimale Tankstrategie zu berechnen. Beim Klick des unteren wird eine CSV-Datei erwartet, die Zeilen beinhaltet mit jeweils einer Tankstellen-ID und einem Datum, woraufhin eine Preisvorhersage berechnet wird.

Abbildung A.3: die Benutzer-Interaktionsmöglichkeiten

A.0.1 Routen-Berechnung

Nach Eingabe der Bertha-Benz-Memorial-Route ist diese auf der Karte zu sehen (siehe Abbildung A.4). Die Geo-Punkte der jeweiligen Tankstellen werden aus der Datenbank entnommen und auf die Karte projiziert. Ein blauer Punkt entspricht einer Tankstelle (Stopp), der grüne Punkt entspricht dem Startpunkt der Route, der beschriftete, rote Punkt der aktuellen Position und der zweite rote Punkt der letzten Station auf der Route.

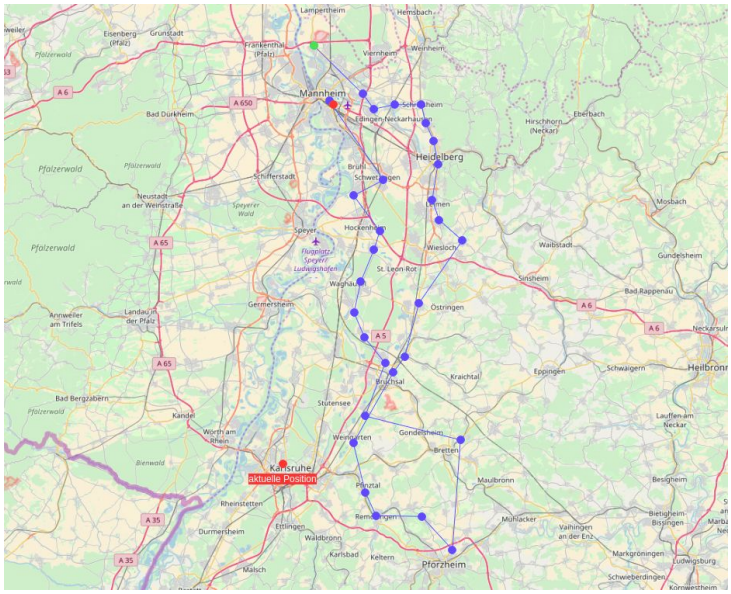

Abbildung A.4: Anzeige einer Tankroute

Neben der Karte kann ein Reiter eingeblendet werden, indem man den Navigations-Button oben links drückt (siehe Abbildung A.5). Dieser dient dazu, die errechneten Tankfüllungen pro Routen-Stopp anzuzeigen.

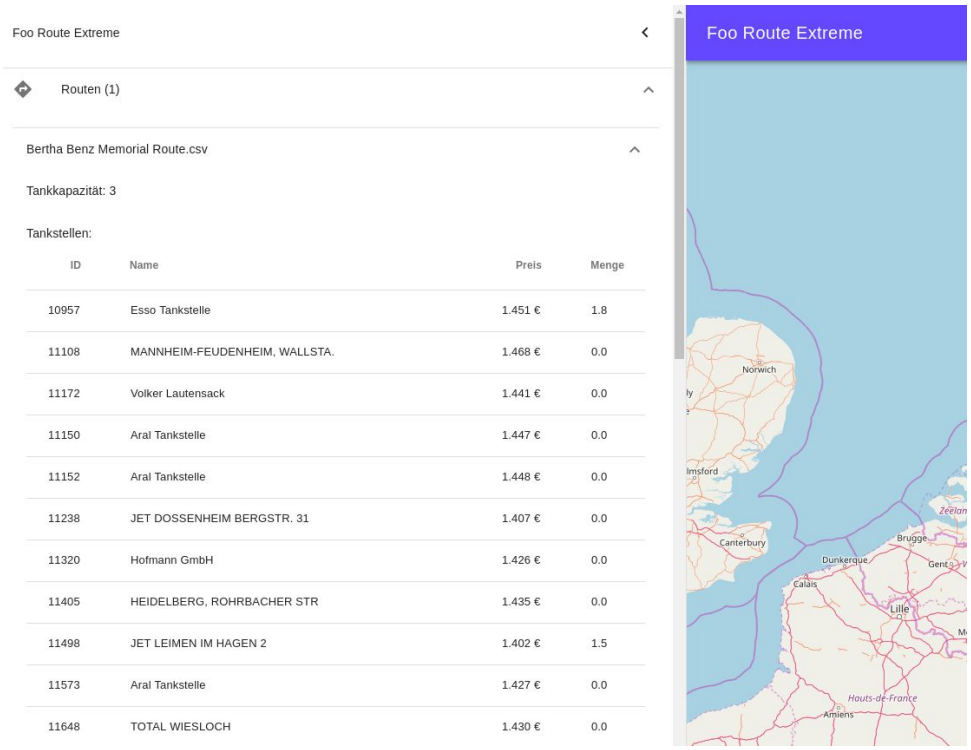

Abbildung A.5: die errechneten Tankfüllungen für die Route

A.0.2 Preisvorhersage

Wie auch bei der Routen-Berechnung werden bei der Funktion 'Preisvorhersage' alle Tankstellen in Form eines blauen Punktes auf der Karte illustriert (siehe Abbildung A.6). Der Unterschied ist jedoch, dass die jeweiligen Tankstellen nicht untereinander (wie bei einer Route) verbunden sind.

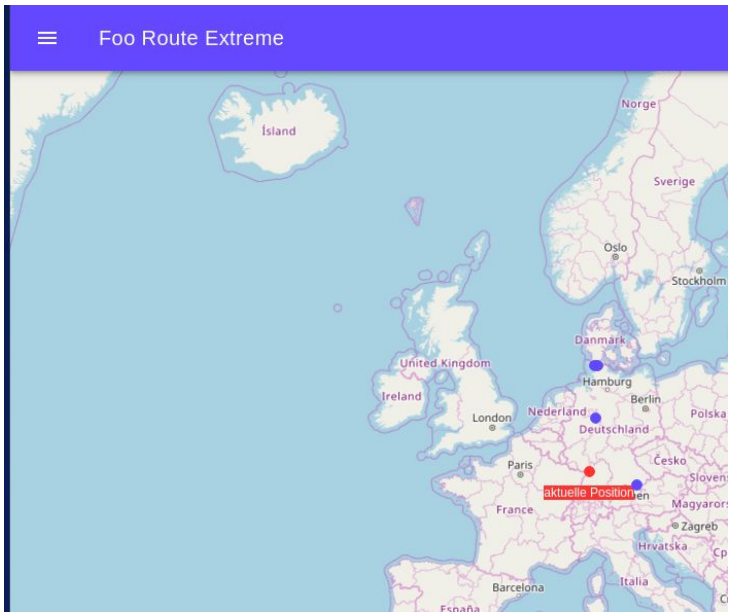

Abbildung A.6: die Tankstellen auf der Karte bei einer Preisvorhersage

Durch Klick auf die Top-Navigation, um den Reiter zu öffnen, werden die vorhergesagten Preise pro Tankstelle angezeigt (siehe Abb. A.7).

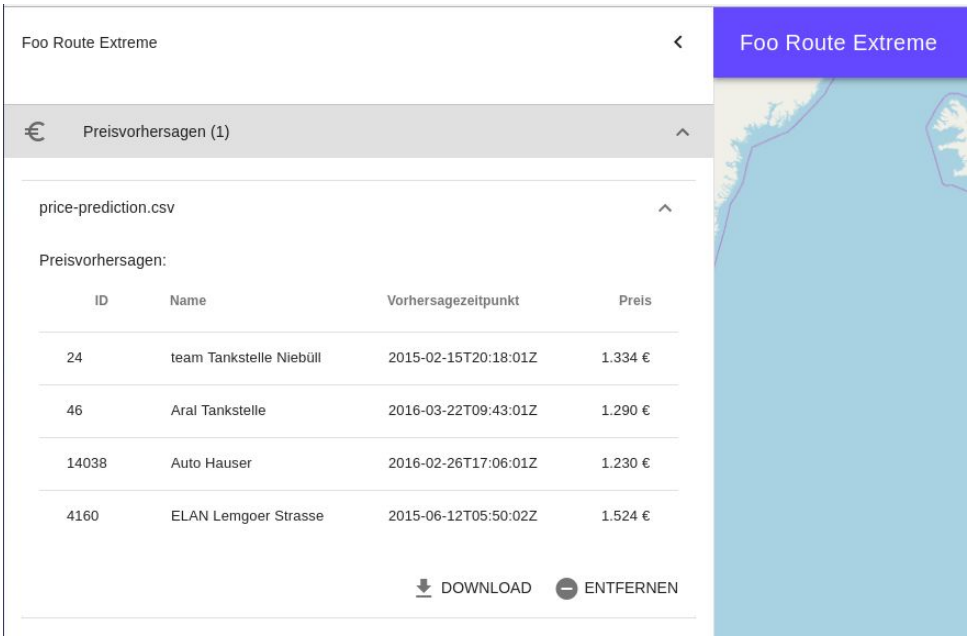

Abbildung A.7: die errechneten Preise zu den jeweiligen Daten

Zusätzlich bietet der Reiter die Möglichkeit die Routen zu löschen oder herunterzuladen.

Literatur

[1] M. Brandt, *Auto ist in Deutschland immer noch die Nummer 1*, https://de.statista.com/infografik/9162/nutzung-von-verkehrsmitteln-in-deutschland/, letzter Zugriff: 21.01.2018, 2017.

[2] S. Khuller, A. Malekian und J. Mestre, „To Fill or Not to Fill: The Gas Station Problem“, *ACM Trans. Algorithms*, Jg. 7, Nr. 3, 36:1–36:16, Juli 2011, issn: 1549-6325. doi: 10.1145/1978782.1978791. Adresse: http://doi.acm.org/10.1145/1978782.1978791.

[3] *Haversine formula*, https://rosettacode.org/wiki/Haversine_formula, letzter Zugriff: 19.01.2018.

[4] A. B. Downey, *Think Stats*. O'Reilly Media, Inc., 2011, isbn: 1449307116, 9781449307110.

[5] P. Lippe, *Deskriptive Statistik*. Stuttgart Jena: G. Fischer, 1993, isbn: 3-8252-1632-2.

[6] R. Adhikari und R. K. Agrawal, „An Introductory Study on Time Series Modeling and Forecasting“, *arXiv:1302.6613 [cs, stat]*, Feb. 2013. arXiv: 1302.6613 [cs, stat].

[7] M. Nottingham, *Problem Details for HTTP APIs*, https://tools.ietf.org/html/draft-nottingham-http-problem-06, letzter Zugriff: 19.01.2018, 2014.

[8] E. Gamma, R. Helm, R. Johnson und J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, isbn: 0-201-63361-2.

[9] dr-prodigy, *holidays-Python Package*, https://pypi.org/project/holidays/.

[10] Schulferien.org, https://www.schulferien.org/, letzter Zugriff: 18.01.2018.

[11] OpenStreetMap.org, *OpenStreetMap*, http://www.openstreetmap.org, letzter Zugriff: 18.01.2018.

[12] C. Seidenbeidel, „Das Wunder des billigen Öls“, *Frankfurter allgemeine Zeitung*, 2015, letzter Zugriff: 19.01.2018. Adresse: http://www.faz.net/aktuell/finanzen/devisen-rohstoffe/wie-lange-autofahrer-noch-so-billig-tanken-koennen-13363092.html.

[13] statista.com, *Durchschnittlicher Benzinpreis in Deutschland in den Jahren 1972 bis 2018 (Cent pro Liter Superbenzin)*, https://de.statista.com/statistik/daten/studie/776/umfrage/durchschnittspreis-fuer-superbenzin-seit-dem-jahr-1972/, letzter Zugriff: 19.01.2018, 2018.

[14] J. H. Friedman, „Greedy function approximation: A gradient boosting machine.“, *Ann. Statist.*, Jg. 29, Nr. 5, S. 1189–1232, Okt. 2001. doi: 10.1214/aos/1013203451. Adresse: https://doi.org/10.1214/aos/1013203451.

[15] C. M. Schwarzer, „Das Ladeelend hat ein Ende“, *Zeit Online*, 2017, letzter Zugriff: 20.01.2018. Adresse: `https://www.zeit.de/mobilitaet/2017-03/elektroauto-infrastruktur-bundesverkehrsministerium-foerdergeld-schnell-ladestationen`.

[16] *Energiemarkt Kommentar: Ölpreis dominiert alle Energiepreise*, `https://www.ispex.de/energiemarkt-kommentar-02-2016-oelpreis-dominiert-alle-energiepreise/`, letzter Zugriff: 20.01.2018, 2016.