

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, Tom Ganz, Carlo Götz

Abgabedatum 22. Mai 2018

Zeitraum
Kurs
Gutachter der Studienakademie

3. Studienjahr
TINF15B5
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

1	Einleitung	1
1.1	InformatiCup	1
1.2	Aufgabenstellung	1
1.3	Motivation	1
2	Theoretische Grundlagen	2
2.1	Fixed-Path-Gas-Station Problem	2
2.1.1	Implementierung	3
2.2	Zeitreihenanalyse	4
2.2.1	Gleitende Durchschnitte	4
2.2.2	Autokorrelation	4
2.2.3	Autoregressive-Moving Average (ARMA)-Modelle	4
2.2.4	Stationarität	5
2.2.5	Autoregressive Integrated Moving Average (ARIMA)-Modelle	6
2.3	Regressionsanalyse	6
2.3.1	Lineare Regression	6
2.3.2	Regression für Zeitreihenanalyse	7
3	Implementierungsdetails	9
3.1	Backend	9
3.1.1	Architektur	9
3.1.2	Persistenz	11
3.1.3	Tests	13
3.2	Frontend	14
3.2.1	Beschreibung der CSV Dateien	14
3.2.2	Design	15
3.2.3	Implementierung	15
3.2.4	Tests	17
3.2.5	Einschränkungen	18
3.3	Datenverarbeitung / Preisvorhersage	18
3.3.1	Feature Engineering	18
3.3.2	Modellauswahl	19
3.3.3	Training	21
3.4	CLI	22
4	Besonderheiten / Vorteile bei bestimmten Routen / Preisvorhersagen	23
4.1	Fehlerfälle	23
4.1.1	Nicht existierende Tankstellen	23
4.1.2	Leere Tankstrategie	23

4.1.3	Negative Tankkapazität	24
4.1.4	Tankstops sind nicht in zeitlicher Reihenfolge	24
4.1.5	Routen mit Tankstellen ohne Preisinformationen	24
4.1.6	Daten nach angefragtem Zeitpunkt verfügbar	24
4.1.7	Daten bis kurz vor angefragtem Zeitpunkt verfügbar	24
4.1.8	Routen Stops nicht erreichbar mit vollem Tank	25
4.2	Fehlversuche	25
4.2.1	Daten-Präparation	25
4.2.2	Modellerstellung	26
4.3	Evaluation	27
4.3.1	Fixed-Path-Gas-Station Algorithmus	27
4.3.2	Allgemeine Vorhersagegüte des Modells	28
5	Erweitertes Anwendungsszenario	31
5.1	Definition des Szenarios	31
5.2	Anpassung des Algorithmus	31
5.3	Neuberechnung des Routing Graphen	32
5.4	Neuberechnung des Modells zur Preisvorhersage	33
5.5	Entwicklung einer Architektur zur Skalierung der Anwendung	34
5.6	Weitere Ansatzpunkte zur Erweiterung des Systems	36
6	Ausblick	37
A	Anleitungen	38
	Installationsanleitung	38
	Bedienungsanleitung	38
	A.0.1 Routen-Berechnung	40
	A.0.2 Preisvorhersage	41

Abbildungsverzeichnis

2.1	Beispiel Ablauf	2
2.2	Anpassungsgrad der Regressionsfunktion.	7
3.1	Abhängigkeiten der einzelnen Packages des Backends	10
3.2	Start Ansicht des Frontends	16
3.3	Frontend während Verwendung	16
3.4	Relative Relevanz der einzelnen Merkmale für die Vorhersage von Benzinpreisen.	19
3.5	Entwicklung der Jahresdurchschnittspreise für einen Liter Super Benzin in den Jahren 2012 bis 2017. Die Daten stammen von statista.com [16].	20

4.1	Die tatsächlichen Preise (orange) und die der Lasso-Regression vorhergesagten (blau und von der orangefarbenen Kurve verdeckt)	26
4.2	RMSE pro Anzahl trainierter Entscheidungsbäume	30
5.1	Initiales Laden von OSM Daten in die Datenbank und Erstellung eines gewichteten Graphen zur Routenberechnung	33
5.2	Update der OSM Geometrien und Neuberechnung des gewichteten Graphen . . .	33
5.3	Horizontal skalierbare Architektur der Anwendung	35
A.1	Server-Log beim Starten des Projekts	39
A.2	grundlegende Oberfläche der Webanwendung	39
A.3	die Benutzer-Interaktionsmöglichkeiten	40
A.4	Anzeige einer Tankroute	40
A.5	die errechneten Tankfüllungen für die Route	41
A.6	die Tankstellen auf der Karte bei einer Preisvorhersage	42
A.7	die errechneten Preise zu den jeweiligen Daten	42

Tabellenverzeichnis

3.1	Im Backend verwendete Spalten der Tabelle stations	12
3.2	Auflistung der zusätzlich verwendeten Merkmale pro Benzinpreis und Tankstelle.	18
3.3	Die Hyperparameter-Werte des GBM-Modells, welche durch die Grid-Search optimiert wurden.	21
4.1	Vergleich der drei Kennzahlen MAE, RMSE und R^2	29

Abkürzungsverzeichnis

API Application Programming Interface	10
ARMA Autoregressive-Moving Average	II
ARIMA Autoregressive Integrated Moving Average	II
CLI Command Line Interface	22
CORS Cross-Origin Resource Sharing	11
CSV Comma Separated Values	14
DI Dependency Injection	9
GBM Gradient Boosting Machine	20
HTTP Hypertext Transfer Protocol	9
IOC Inversion Of Control	9
JSF Java Server Faces	9
JSON Javascript Object Notation	9
MAE Mean Absolute Error	29
OSM Open Street Map	12

POC Proof of Concept	31
RDBMS Relational Database Management System	11
RMSE Root Mean Squared Error	29
SPA Single Page Application	15
UX User Experience	31
WAL Write Ahead Log	35
XML Extensible Markup Language	33

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 und Hochschulen 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. In Abbildung 2.1 ist eine Visualisierung des Problems zu erkennen. In unserer Problemstellung ist der Pfad jedoch festgelegt und kann nicht geändert werden, somit ist das Problem des 'Shortest-Path' zu vernachlässigen. 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

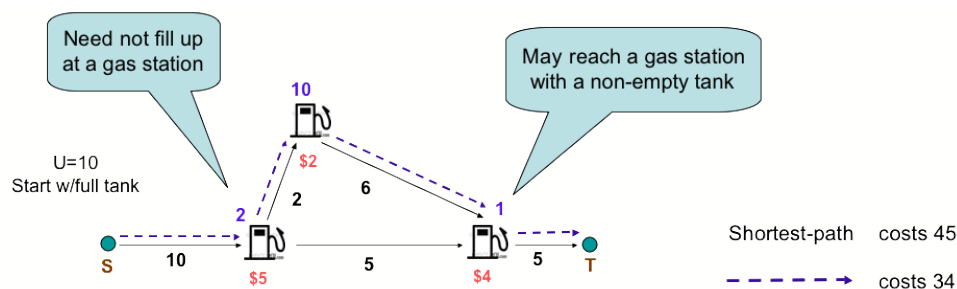


Abbildung 2.1: Visualisierung des Problems [3]

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 [4].

Die Berechnung der Haversine-Formel ist gegeben durch

$$\begin{aligned}\Delta longitude &= longitude_b - longitude_a \\ \Delta latitude &= latitude_b - latitude_a \\ a &= \left(\frac{\sin \Delta latitude}{2}\right)^2 + \cos latitude_a * \cos latitude_b * \left(\frac{\sin \Delta longitude}{2}\right)^2 \\ c &= 2 * \arcsin \min(1, \sqrt{a}) \\ d &= R * c\end{aligned}$$

Wobei R dem Erdradius entspricht.

Die Haversine-Formel ermöglicht die Berechnung der Distanz zweier Geokoordinate auf einer Kugel-Oberfläche (sogenannte Orhodrome).

2.1.1 Implementierung

Unsere Lösung für das Fixed-Path-Gas-Station Problem lässt sich in einem Algorithmus implementieren der eine Zeitkomplexität von $O(n * \log(n))$ benötigt[2]. Hierzu wird eine Priority-Queue mit den Stationen bereitgestellt die man mit einem vollen Tank von einer Station l_0 aus erreichen kann. Diese Priority-Queue wird ähnlich einem 'Moving Window' von Station zu Station in Richtung Ziel-Tankstelle 'geschoben'. Die Priority-Queue ermittelt zu jeder Tankstelle die Position abhängig ihres Preises und ihrer Nähe zum Ziel. Die nächste Tankstelle von unserer Start-Tankstelle l_0 ermitteln wir durch die Operation 'Dequeue'. Jedes mal wenn wir das Fenster um eine Tankstelle verschieben wird die Start-Tankstelle entfernt und eine näher zum Ziel hinzugefügt. Da das Entfernen und Hinzufügen in $O(\log(n))$ verrichtet werden kann[**queue**], entspricht der Aufwand das Problem zu lösen $O(n * \log(n))$. Um zu beweisen, dass die Lösung optimal ist, muss nach dem Bellman'schen Optimalitätsprinzip[5] nur bewiesen werden, dass jedes Subproblem (Zwischenroute) l_i zu l_{i+1} optimal ist. Diese ist es, falls der Tank l_i genau dem entspricht, was zum Erreichen von l_{i+1} notwendig ist, also wenn am ende der Tank leer ist.

2.2 Zeitreihenanalyse

Bei einer Zeitreihe handelt es sich um eine zeitlich geordnete Sequenz von Messungen eines Systems [6, 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 [6, 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. Genauer 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 [6, 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 [7, 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 [7, 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 [6, 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 [8, 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:

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

Die Werte ϕ_i ($i = 1, 2, \dots, p$) sind Modellparameter. Das MA(q)-Modell benutzt vergangene Fehler des Modells als erklärende Variable [8, S. 19] und ist wie folgt definiert:

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

In dieser Definition beschreibt μ den Mittelwert über q gleitende Durchschnitte und Θ_i ($i = 1, 2, \dots, q$) 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) [8, S. 19]. Die Kombination der beiden Einzelmodelle liefert dann das gewünschte ARMA(p, q)-Modell, welches eine Vorhersage nach folgender mathematischen Vorschrift berechnet:

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

2.2.4 Stationarität

Üblicherweise werden ARMA-Modelle mit dem Lag-Operator definiert [8, S. 19], welcher mathematisch als $Ly_t = y_{t-1}$ definiert ist. Somit lassen sich ARMA-Modelle mittels Lag-Polynomen wie folgt definieren:

$$AR(p) - \text{Modell} : \epsilon_t = \phi(L)y_t$$

$$MA(q) - \text{Modell} : y_t = \Theta(L)\epsilon_t$$

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

In diesem Fall gilt:

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

Ein solches Modell wird als stationär bezeichnet, genau dann wenn die Modellparameter Zeitunabhängig sind [6, S. 160]. Ein ARMA(p, q)-Modell ist stationär, wenn für alle Wurzeln aus der charakteristischen Gleichung $\phi(L) = 0$ außerhalb des Einheitskreises liegen [8, 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:

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

- p , d und q sind dabei Integer ≥ 0 und bezeichnen die Ordnung der Autoregressiven-, Integrierten- und gleitenden Durchschnitts-Komponente des Modells.
- d beschreibt das Differenzierungslevel. 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

Das Ziel der Regressionsanalyse ist die Beschreibung der Beziehung von einem Variablensatz, genannt *abhängige Variablen*, zu den sogenannten *erklärenden Variablen* [6, S. 129]. Die Regression liefert also für einen mehrdimensionalen Eingabevektor einen konkreten numerischen Wert. Der Algorithmus soll also eine Funktion $f: \mathbb{R}^n \rightarrow \mathbb{R}$ entwickeln. Für den Fall, dass jeweils genau eine abhängige und erklärende Variable für die Regression verwendet wird, spricht man von einer *einfachen Regression*, bei mehr als einer erklärenden Variable wird von *Multipler Regression* gesprochen [6, S. 129]. Entsprechend dem Grad der Regressionsfunktion wird in unterschiedliche Methoden unterschieden, welche nachfolgend erläutert werden.

2.3.1 Lineare Regression

Lineare Modelle sind die einfachsten parametrischen Methoden und sollten für jede Problemstellung zunächst in Betracht gezogen werden, da sie oft auch intrinsisch nicht-lineare Probleme einfach lösen können [9, S. 72]. Die zentrale Annahme eines linearen Modells ist dabei, dass alle Werte im Datensatz und alle anderen unbekannten Punkte auf einer Hyperebene liegen

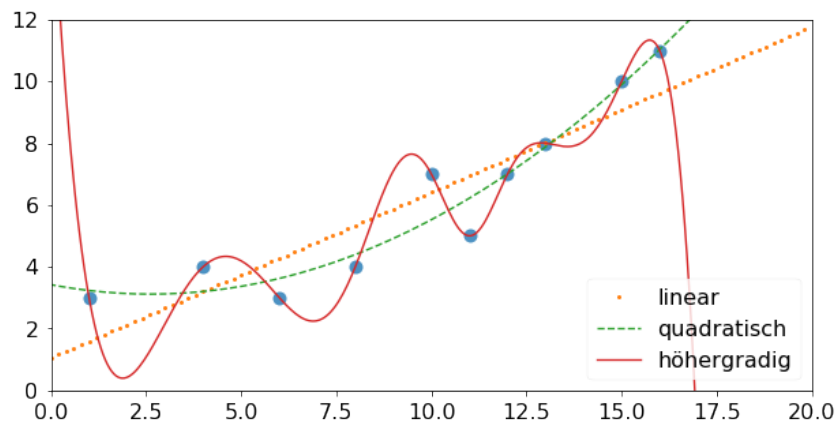


Abbildung 2.2: Die hellblauen Datenpunkte werden durch die drei Regressionsfunktionen approximiert. Hier ist zu beachten, dass die lineare Funktion (Orange gepunktet) die Daten nicht gut approximiert. Die quadratische Funktion (Grün gestrichelt) approximiert den Zusammenhang gut, die höhergradige Funktion passt sich den Datenpunkten „zu gut“ an. Man spricht hier von *Overfitting*.

und der maximale Fehler proportional zur Trainingsqualität und Adaptivität ist [9, S. 73]. Bei einem linearen Zusammenhang der untersuchten Variablen wird die Regressionsgerade über zwei Parameter, die sogenannten Korrelationskoeffizienten bestimmt:

$$\bar{y} = \alpha + \beta x \quad (2.6)$$

Der Parameter α denotiert dabei den Achsenabschnitt und β die Steigung der Regressionsgeraden.

Gesucht ist eine möglichst gute Approximation der vorhandenen Datenpunkte durch die Regressionsfunktion. In Abbildung 2.2 sind drei mögliche Regressionsfunktionen zu einem Beispieldatensatz abgebildet. Es ist zu beachten, dass die lineare Funktion hier keine gute Approximation des Zusammenhangs darstellt. Die beste Approximation wird durch die höhergradige Funktion erzielt. Dies ist jedoch nicht immer das Ziel, da mit diesem Modell sehr schlecht zukünftige Werte vorhergesagt werden können. Anschaulich wird dies, wenn man sich das rechte Ende des Funktionsgraphen anschaut. Nach dem Verlauf der roten Kurve müssten alle weiteren Werte der Messreihe plötzlich stark abfallen, obwohl eigentlich ein weiterer Anstieg zu erwarten wäre. Die beste Approximation im Kontext einer Regressionsanalyse liefert daher die quadratische Funktion.

2.3.2 Regression für Zeitreihenanalyse

Wie in [9, S. 72] erwähnt wird, stellen lineare Modelle oft einen guten Startpunkt für die Analyse vieler Probleme dar. So kann theoretisch auch eine Zeitreihenanalyse mit Hilfe eines Regressionsmodells durchgeführt werden. Jedoch ist hierbei Vorsicht geboten, denn es sprechen einige Gründe gegen die Verwendung eines Regressionsmodells bei Zeitreihenanalysen. Die folgenden drei Punkte sind [6, S. 150] entnommen:

1. Es ist in keinsten Weise garantiert, dass der Langzeittrend bei einem realen Vorgang, welcher durch die Zeitreihe dokumentiert wird, einem linearen - oder höhergradigen - Modell folgt. Bezogen auf diese Arbeit bedeutet dies, dass die Benzinpreisentwicklung stark von anderen Faktoren als der Zeit abhängt. Das gebildete Modell kann somit zu einem späteren Zeitpunkt seine Gültigkeit verlieren.
2. Die Lineare Regression gewichtet alle historischen Daten gleich stark. Die Gewichtung sollte jedoch stärker von den jüngsten Messwerten abhängen, da diese die aktuelle Entwicklung der Messreihe genauer beschreiben.
3. Eine der Annahmen Linearer Regression ist, dass die Messfehler unkorrelierte Störungen sind. In Zeitreihenanalyse ist diese Annahme oft falsch, da aufeinanderfolgende Werte oft korrelieren. Dies wurde bereits in Unterabschnitt 2.2.2 erläutert.

Die Stärke der Regressionsanalyse gegenüber den vorgestellten ARMA-Modellen liegt hingegen darin, dass zusätzliche Merkmale in den Merkmalsvektor zur Vorhersage künftiger Ereignisse hinzugezogen werden können. Dies kann die allgemeine Vorhersagegüte des Modells deutlich verbessern.

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 instanziiieren. 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.

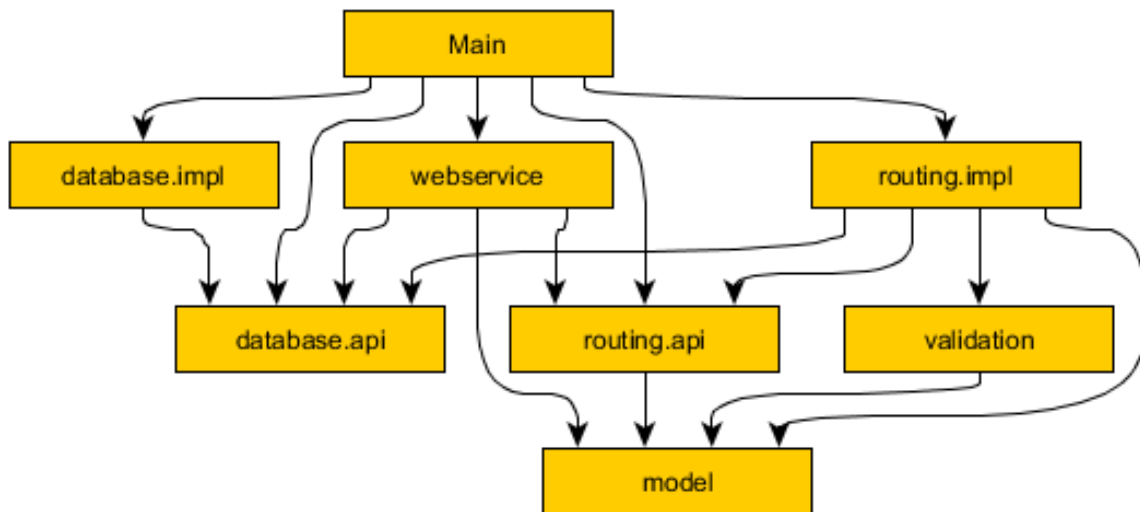


Abbildung 3.1: Abhängigkeiten der einzelnen Packages des Backends

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** verknüpft. 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 [10] 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 [11], 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
<code>id</code>	<code>smallint</code>	Id wie in Eingabedaten
<code>lat</code>	<code>double</code>	Breitengrad in WGS84
<code>lon</code>	<code>double</code>	Längengrad in WGS84
<code>station_name</code>	<code>varchar(255)</code>	Name
<code>street</code>	<code>varchar(255)</code>	Adresse
<code>house_number</code>	<code>varchar(255)</code>	Hausnummer
<code>zip_code</code>	<code>varchar(5)</code>	PLZ
<code>city</code>	<code>varchar(255)</code>	Ort
<code>brand</code>	<code>varchar(255)</code>	Marke
<code>brand_no</code>	<code>int</code>	Id der Marke
<code>bland_no</code>	<code>int</code>	Id des Bundeslands
<code>kreis</code>	<code>varchar(12)</code>	Verwaltungsschlüssel des Landkreises
<code>abahn_id</code>	<code>int</code>	Id der nächsten Autobahn
<code>bstr_id</code>	<code>int</code>	Id der nächsten Bundesstraße
<code>sstr_id</code>	<code>int</code>	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 Beschreibung der CSV Dateien

Die Anwendung verarbeitet CSV Dateien und bietet die Möglichkeit die Ergebnisse als CSV-Datei zu exportieren. Die Anwendung kann sowohl Preis- als auch Routenanfragen verarbeiten, die jeweils durch ihre eigene CSV-Datei repräsentiert werden. 3.1 zeigt eine Datei für die Anfrage von Preisen. In jeder Zeile ist eine einzelne Anfrage für eine Tankstelle gegeben. Eine Zeile besteht aus drei Informationen:

1. Zeitpunkt bis zu dem die Preise als gegeben angenommen werden dürfen.
2. Zeitpunkt für den der Preis vorhergesagt werden soll.
3. Id der Tankstelle für die der Preis vorhergesagt werden soll.

Algorithmus 3.1: CSV einer Preisanfrage

```
2015-02-10 12:18:01+01;2015-02-15 21:18:01+01;24
2016-03-22 10:42:01+01;2016-03-22 10:43:01+01;46
2016-01-27 03:06:01+01;2016-02-26 18:06:01+01;14038
2015-06-11 14:30:02+02;2015-06-12 07:50:02+02;4160
```

Die Antwort auf solch eine Anfrage wird in 3.2 gezeigt. Die einzelnen Zeilen wurden jeweils um einen vierten Wert ergänzt. Dieser Wert ist der Benzinpreis in Zehntel-Cent.

Algorithmus 3.2: CSV einer Preisantwort

```
2015-02-10 12:18:01+01;2015-02-15 21:18:01+01;24;1153
2016-03-22 10:42:01+01;2016-03-22 10:43:01+01;46;1533
2016-01-27 03:06:01+01;2016-02-26 18:06:01+01;14038;1507
2015-06-11 14:30:02+02;2015-06-12 07:50:02+02;4160;139
```

Die Datei für die Anfrage einer optimalen Tankstrategie wird in 3.3 gezeigt. Die erste Zeile enthält die Tankkapazität des Fahrzeugs in Litern. Jede weitere Zeile enthält einen Zeitpunkt und die Id einer Tankstelle. Diese Zeilen geben die erwartete Ankunftszeit an der jeweiligen Tankstelle an.

Algorithmus 3.3: CSV einer Routenanfrage

```
3
2015-08-01 08:00:00+02;10957
2015-08-01 09:55:31+02;11108
2015-08-01 10:27:25+02;11172
2015-08-01 11:02:43+02;11150
2015-08-01 11:48:16+02;11152
```

Die entsprechende Antwort wird in 3.4 gezeigt. Die Zeile mit der Tankkapazität wird nicht mehr aufgeführt. Jede Zeile wurde um zwei weitere Werte erweitert. Dabei handelt es sich um den vorhergesagten Preis in Zehntel-Cent und der Menge, die an der Tankstelle getankt wurde in Litern. Dabei handelt es sich um eine Fließkommazahl.

Algorithmus 3.4: CSV einer Routenantwort

```
2015-08-01 08:00:00+02;10957;1451;1.838860165385679
2015-08-01 09:55:31+02;11108;1468;0
2015-08-01 10:27:25+02;11172;1441;0
2015-08-01 11:02:43+02;11150;1447;0
2015-08-01 11:48:16+02;11152;1448;0
```

Die CSV-Dateien geben Zeitpunkte im Format YYYY-MM-DD HH:MM:SS+HH an. Bei Ids der Tankstellen handelt es sich um Ganzzahlen.

3.2.2 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.

3.2.3 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

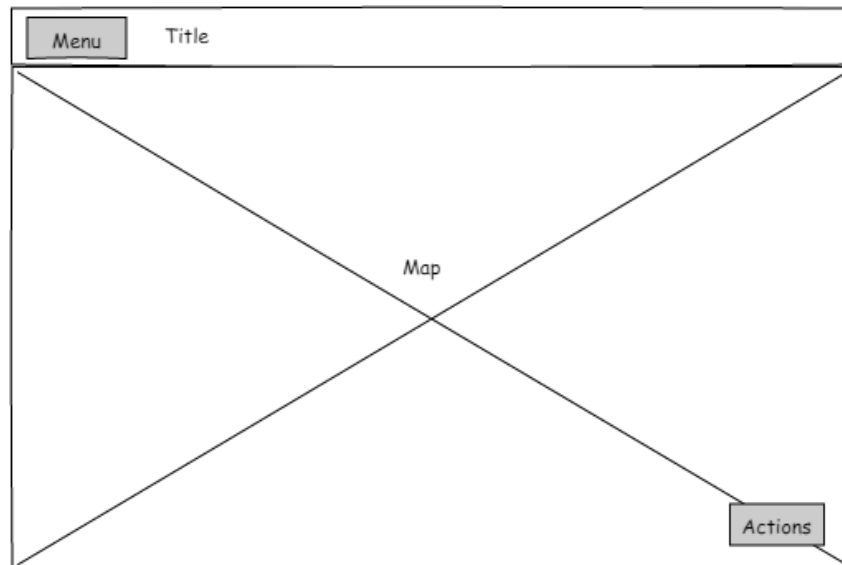


Abbildung 3.2: Start Ansicht des Frontends

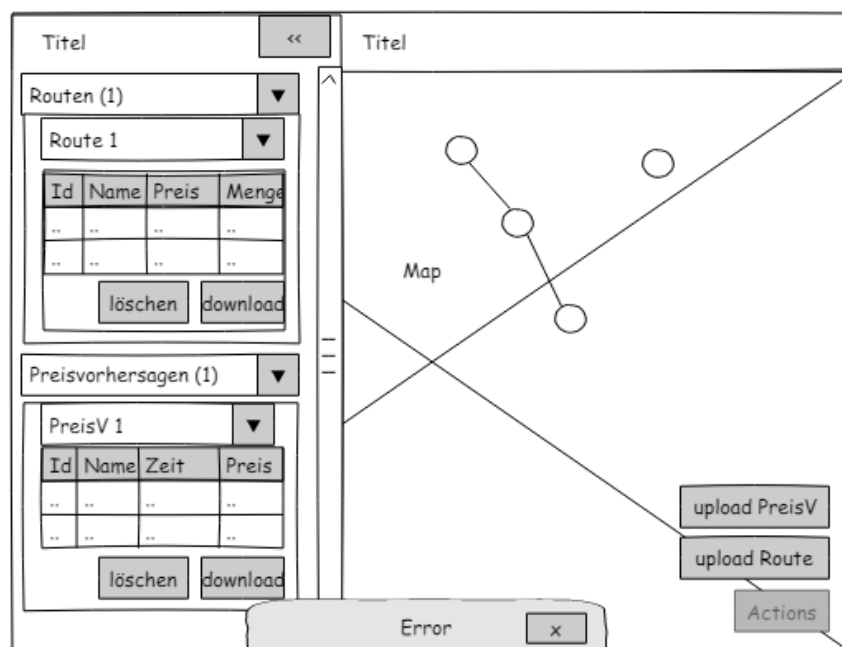


Abbildung 3.3: Frontend während Verwendung

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 **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.

3.2.4 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

Merkmal	Ursprung
Feiertage	holidays Python Package [12]
Schulferien	Schulferien.org [13]
Bundesland	} OpenStreetMap [14]
Landkreis	
nächste Hauptverkehrsstraße	

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

16. Action Preisvorhersage Downloaden aufrufen

17. Prüfen, ob CSV korrekt ist

3.2.5 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

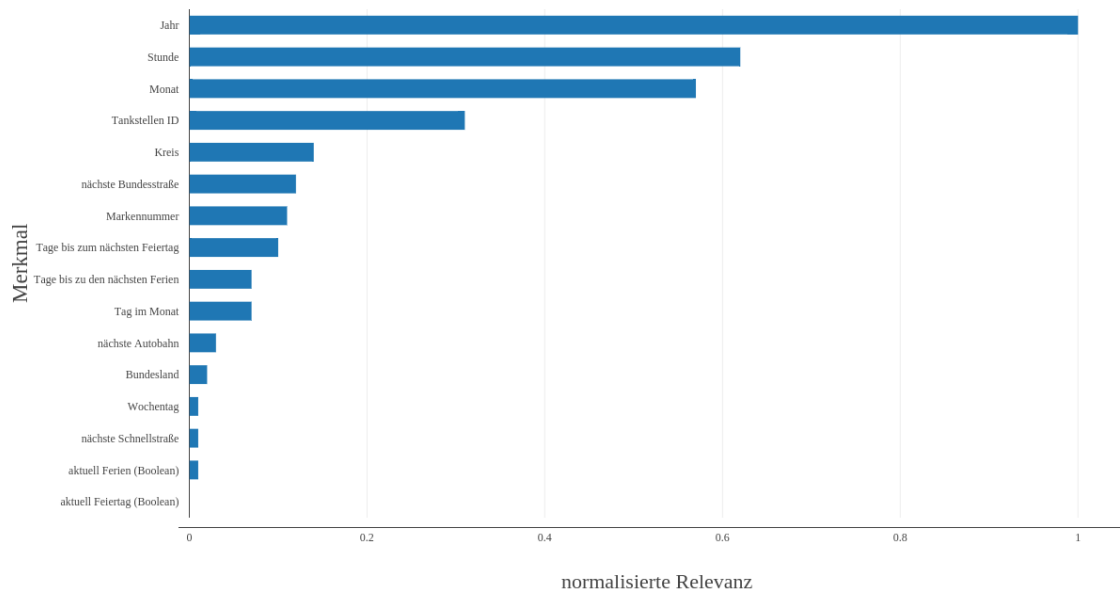


Abbildung 3.4: Relative Relevanz der einzelnen Merkmale für die Vorhersage von Benzinpreisen.

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 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 [15]. 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

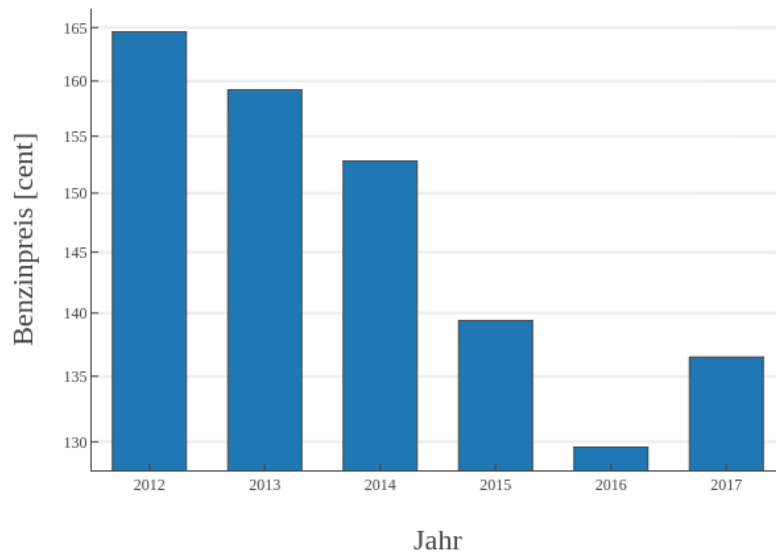


Abbildung 3.5: Entwicklung der Jahresdurchschnittspreise für einen Liter Super Benzin in den Jahren 2012 bis 2017. Die Daten stammen von statista.com [16].

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.

Gradient Boosting Machine

Die Erklärung der GBM erfolgt nach Friedman [17]. 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.

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	<code>bertha_bogus_station.csv</code>
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	<code>bertha_empty.csv</code>
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 Tankkapazität 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 nicht erreichbar.
Verhalten	Anwendung antwortet mit entsprechender ProblemResponse, kein Absturz

4.2 Fehlversuche

Unser hinreichend gutes Ergebnis mittels der 'Gradient Boosting Machine' auf dem Framework 'h2o' ist Ergebnis einer umfangreichen 'Data-Exploration': Es wurden verschiedene Modelle und Algorithmen verglichen um ein Vorhersagemodell zu entwerfen. Unsere Vorgehensweise war allgemein wie folgt:

Daten-Präparation Die Angleichung und Vorverarbeitung von den Features (auch Feature-Engineering genannt).

Modellerstellung Die Erstellung und Bewertung verschiedener Modelle und Algorithmen für unsere Domäne.

Während dieses Prozesses, wurden häufig Modelle und Algorithmen verworfen. Einige werden in diesem Kapitel dargelegt und begründet, wieso sie nicht zu unserem Problem passten.

4.2.1 Daten-Präparation

Während der Daten-Präparation zum Testen einiger Modelle, genügte es häufig ein Bruchteil des gesamten Datensatzes zum Trainieren zu verwenden und wiederum ein Bruchteil dessen, um es zu testen. Viele Modelle benötigten eine stationäre Zeitserie an Daten. Damit ist gemeint, dass jeglicher Datenpunkt in der Serie unabhängig von der Zeit ist.

Anfänglich lagen unsere Daten von unterschiedlichen Tankstellen zu unterschiedlichsten Messzeiten vor. Modelle wie 'ARIMA' benötigen eine gleichbleibende Frequenz der Daten-Zeitpunkte. Also wurden die Daten in eine einheitliche Frequenz abgetastet. Mit Hilfe von PostgreSQL konnten wir die Daten nun zu stündlichen Messpunkten vorlegen. Üblicherweise werden diese Daten von einem 'Trend' und 'Saisonalität' bereinigt.

Um die Stationarität eines Datensatzes zu bewerten wird häufig der 'Augmented Dickey-Fuller test' [18] und zur Bereinigung der Daten hierfür verschiedene Methoden wie die Daten zu logarithmieren oder mittels 'First order Difference' (Differenz der Zeitserie mit nach der um t Zeitpunkten verschobenen Zeitserie) angewendet. Mittels unseres anfänglichen bereinigten Datensatzes, war es uns nun möglich eine 'Partial Autocorrelation'[19] (ähnlich wie bei ARIMA) durchzuführen. Dadurch wird ein Korrelationskoeffizient berechnet, welcher aussagt, wie sehr eine um t Zeitpunkten verschobene Zeitreihe verglichen zu sich selbst, korreliert.

Als Ergebnis für verschiedene 'Time-Lags' schlossen wir dass die Zeitreihe nach einem 'lag' am meisten mit sich selbst korreliert, also der Benzinpreis einer Stunde vorher am meisten darüber aussagen kann, wie sich der Benzinpreis in einer Stunde entwickeln wird. Dadurch, dass wir die Benzinpreise stündlich abtasteten lag es Nahe, dass der vorhergehende Preis nicht nur ähnlich ist,

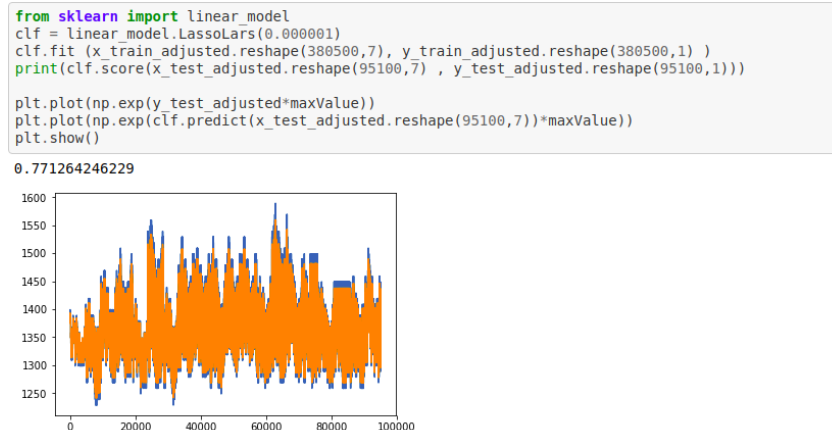


Abbildung 4.1: Die tatsächlichen Preise (orange) und die der Lasso-Regression vorhergesagten (blau und von der orangefarbenen Kurve verdeckt)

sogar sehr wahrscheinlich derselbe. Denn falls zu einem Zeitpunkt t_k kein neuer Preis vorliegt, wird der von t_{k-1} übernommen.

4.2.2 Modellerstellung

Durch unsere fälschliche Annahme durch die partielle Autokorrelation, entwarfen wir ein Modell mit Hilfe der 'Linearen Regression'. Das erste Modell wies eine vorhergesagte Zeitreihe mit einem extremen Trend und starker Abweichung der tatsächlichen Daten vor (Underfitting). Somit wurde das Modell modifiziert und zur 'Lasso-Regression' überführt.

$$\sum_i^n (y_i - \sum_j^m x_{ij} \beta_j)^2 + \lambda \sum_1^p |\beta_j|$$

Die oben genannte Formel gilt es zu minimieren. Durch die Lasso-Regression wird eine sogenannte L1-Regulation angewendet. Durch zu große Gewichtungsfaktoren β_k wird das Ergebnis insgesamt größer, da β zusätzlich als 'Strafwert' dazu addiert wird. Dadurch werden beim Trainieren automatisch die Features stärker gewichtet, welche größeren Einfluss auf das gesamte Ergebnis haben (vice versa gilt: Features die wenig oder keinen Einfluss haben werden vernachlässigt).

Das Ergebnis der Lasso-Regression war überraschend gut, jedoch war es uns nur möglich Preise eine Stunde vorzusagen (One-Step-Ahead-Forecast). Die Aufgabenstellung erwartete jedoch eine Vorhersage für bis zu einem Monat. Es wäre zwar möglich gewesen das Modell mehrmals auf sich selbst anzuwenden ($f(x_0) \circ f(x_1) \circ f(x_2) \circ \dots$ (Multi-Step-Forecast)), dadurch würde jedoch jeder Fehler, je öfter das Modell mit sich selbst angewendet wird, stärker gewichtet.

In Abbildung 4.1 ist die Bewertung der Lasso-Regression zu sehen. Die orangefarbenen Werte sind die tatsächlichen Werte der Tankstellen. Es wurden 100.000 Datenpunkte zum Testen verwendet. Auf der Y-Achse sind die Preise von 1,25 Euro bis 1,60 Euro pro Liter zu sehen. Die orangefarbene Kurve verdeckt die vorhergesagten blauen Werte, welche nur teilweise ein höheres bzw. niedrigeres Ergebnis lieferten, sich aber ansonsten deckten. Das Framework 'sklearn' bewertet das Modell mit 77 Prozent. Dieser Score setzt sich aus den summierten Differenzen der

tatsächlichen Preisen und den vorhergesagten Preisen, geteilt durch die Anzahl der Beobachtungspunkte zusammen.

Neben der Lasso-Regression wurde auch ARIMA getestet, da wir in diesem Modell aber keine zusätzlichen Features außer Zeitpunkt und Benzinpreis (zwei dimensional) einbringen konnten, wurde dies sehr schnell verworfen.

4.3 Evaluation

In diesem Abschnitt wird die allgemeine Vorhersagegüte des erstellten Modells zur Benzinpreisvorhersage diskutiert.

4.3.1 Fixed-Path-Gas-Station Algorithmus

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

4.3.2 Allgemeine Vorhersagegüte des Modells

Um die allgemeine Vorhersagegüte des erstellten Modells diskutieren zu können, müssen zunächst die einzelnen Fehlermaße erläutert werden. Dies ist mit folgender Übersicht gegeben, wobei die Definitionen an [20] angelehnt sind (n := Anzahl der Vorhersagewerte, \hat{y}_i := Vorhersagewerte, y_i := Beobachtungswerte):

Fehlermaß	Wert
MAE	2.98 cent/Liter
RMSE	4.24 cent/Liter
R^2	78.12 %

Tabelle 4.1: Vergleich der drei Kennzahlen MAE, RMSE und R^2

- **Mean Absolute Error (MAE):**

- Mathematische Definition: $MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$
- Der MAE ist ein größenabhängiges Fehlermaß, der die mittlere absolute Abweichung von der Grundwahrheit ermittelt.

- **Root Mean Squared Error (RMSE):**

- Mathematische Definition: $RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$
- Der RMSE gewichtet im Gegensatz zum MAE die Größe der einzelnen Abweichungen durch Quadrierung. Für einen gleichmäßig verteilten Fehler sind beide Fehlermaße gleich, einzelne Ausreißer werden jedoch durch den RMSE stärker gewichtet.

- **Bestimmtheitsmaß R^2 :**

- Mathematische Definition: $R^2 = \frac{\sum_{i=1}^n (\hat{y}_i - \bar{y})^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$
- Das Bestimmtheitsmaß liegt im Wertebereich $[0,1]$ und ist ein Maß für die Anpassungsgüte einer Regression. Es beschreibt, wie viel Variation in den Daten durch das verwendete Regressionsmodell erklärt werden kann.

Mit diesen Definitionen kann nun die Allgemeine Vorhersagegüte des erstellten Modells bewertet werden. In Tabelle 4.1 sind die für das erstellte Modell berechneten Werte der jeweiligen Kennzahlen eingetragen. Zunächst sind die beiden Fehlermaße MAE und RMSE zu vergleichen. Für beide Fehlermaße gilt, dass niedrigere Werte besser sind. Desweiteren beschreiben beide Maße den durchschnittlichen Vorhersagefehler des Modells. Der Unterschied liegt in der Gewichtung der einzelnen Fehler. Beim RMSE werden die Fehler vor der Mittelung quadriert. Dies ruft eine stärkere Gewichtung von Ausreißern hervor. Zu beachten ist, dass der RMSE nicht mit der Varianz der Fehlerwerte steigt, sondern vielmehr mit der Varianz der Häufigkeitsverteilung der Fehlergrößen [20]. Aus diesen Eigenschaften der Fehlermaße kann nun interpretiert werden, dass das erstellte Modell nicht für alle Tankstellen gleich gute Vorhersagen liefert. Dies ist daran zu erkennen, dass der RMSE höher als der MAE ist. Wenn für jede Vorhersage der gleiche Fehler entstünde, besäßen die beiden Fehlermaße den gleichen Wert. Eine Erklärung für die einzelnen Ausreißer ist, dass im Datensatz einige Tankstellen mit Lücken in den historischen Preisen existieren. Diese Lücken wurden durch das in Abschnitt 3.3 erläuterte Sampling der Preise auf eine stündliche Basis gefüllt. Dies ist ein Fehler, denn das Modell inhärent mitlernt. Es wäre zu untersuchen, ob eine Elimination dieser Tankstellen die Vorhersagegüte positiv beeinflusst. Jedoch bedeutet der Verzicht auf diese Tankstellen eine Verkleinerung des Datensatzes.

Der R^2 -Wert von 78.12% beschreibt eine gute Anpassung der Regression. Die gewählten erklärenden Variablen eignen sich also gut um die abhängige Variable zu beschreiben. Anders

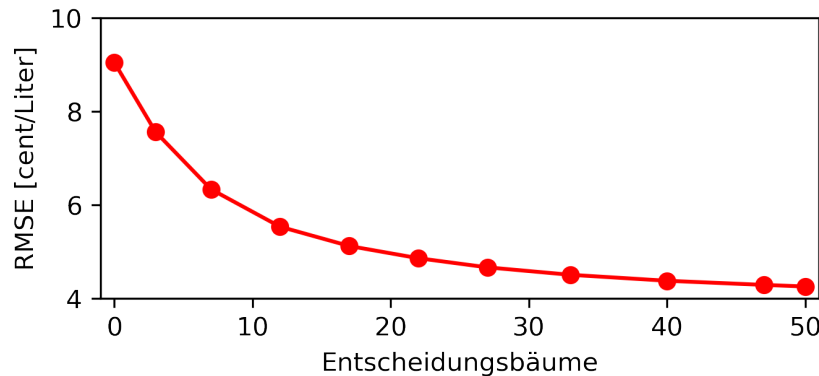


Abbildung 4.2: RMSE pro Anzahl trainierter Entscheidungsbäume

formuliert können durch das erstellte Modell 78.12% der Varianz im Benzinpreis durch die gewählten erklärenden Variablen des Merkmalvektors beschrieben werden.

In Abbildung 4.2 wird der RMSE während des Trainings der Gradient Boosting Machine aufgezeichnet. Es ist zu sehen, wie der Fehler bei fortschreitendem Training, also der Hinzunahme weiterer Entscheidungsbäume immer kleiner wird. Mit 50 Entscheidungsbäumen ist der finale RMSE aus Tabelle 4.1 erreicht.

Allgemein kann gesagt werden, dass das erstellte Modell eine gute Vorhersagegüte besitzt. Es können ungefähr 78 % der Varianz in der Ergebnisvorhersage erklärt werden. Da das erstellte Modell die Preise zu einem gegebenen Zeitpunkt direkt vorhersagen kann, also ohne direkte Abhängigkeit des in der Zeitreihe vorherigen Preises, wirkt der mittlere Fehler noch attraktiver. Das Modell kann somit nämlich auch Preise zuverlässig vorhersagen, die weiter als einen Monat in der Zukunft liegen. Für sehr weit in der Zukunft liegende Preisvorhersagen muss die letzte Aussage jedoch mit Vorsicht genossen werden, da der aktuelle Trend der Zeitreihe nicht garantiert die gleiche Steigung behält. Als Beispiel sei hier nochmals an den Preissturz im Jahr 2015 durch die Einführung des Frackings in den USA erinnert. Solche Ereignisse in der Zukunft können nur schwer vorhergesagt werden und erfordern deshalb eine regelmäßige Aktualisierung des Modells. Durch die direkte Vorhersage der Preise ergibt sich in diesem Modell ein Performance-Vorteil gegenüber schrittweisen Preisvorhersagen. Diese müssen nämlich für jede Preisvorhersage die aktuellsten Preise mit in die Berechnung einbeziehen, wohingegen das hier vorgestellte Modell direkt aus den gegebenen Metadaten einen Preis für einen beliebigen Zeitpunkt berechnen kann. Der benötigte Aufwand zur regelmäßigen Aktualisierung des Modells wird hierbei in den Hintergrund verschoben, so dass der Endnutzer des Systems keine Verzögerungen in der Preisvorhersage erwarten muss. Zusammen mit den guten Ergebnissen des mittleren Fehlers von ca. 3 Cent/Liter kann von einer sehr guten Lösung der IntelliTank-Aufgabe gesprochen werden.

5. Erweitertes Anwendungsszenario

Die im Rahmen der Studienarbeit implementierte Anwendung stellt ein Proof of Concept (POC) dar. Es wurden einige Annahmen getroffen, die die Anwendung für den Einsatz unter realen Bedingungen ungeeignet machen. So erwartet die Anwendung eine Route als Eingabe, die aus einzelnen Tankstellen, sowie der Ankunftszeit an diesen Tankstellen besteht. Auch wurde keine Rücksicht darauf genommen, dass sich die Daten, die der Preisberechnung zugrunde liegen ständig ändern. Dieses Kapitel ist ein Gedankenexperiment, um die Schritte aufzuzeigen, die im Falle einer realen Verwendung der Anwendung nötig wären. Um dies zu erreichen wird zuerst ein reales Einsatzszenario definiert und anschließend werden die erforderlichen Schritte näher erläutert.

5.1 Definition des Szenarios

Es ist recht unrealistisch, dass ein Benutzer Autoreisen plant indem er sämtliche Tankstellen entlang seiner Route ermittelt und die jeweilige Ankunftszeit an den Tankstellen berechnet. Eine bessere User Experience (UX) wäre es die Kalenderdaten des Benutzers abzufragen. Auf Basis der Termine im Kalender kann eine optimale Route berechnet werden.

Es wird die Annahme getroffen, dass die Termine im Kalender jeweils mit einer Adresse versehen sind und dass die Strecke von Termin zu Termin jeweils mit dem Auto bewältigt wird. Die Anwendung könnte so die optimale Route und Tankstrategie für einen gewissen Planungshorizont berechnen.

Dieser Anwendungsfall könnte auch von Privatpersonen auf Speditionen übertragen werden, um die Treibstoffkosten im Güterverkehr zu senken.

5.2 Anpassung des Algorithmus

Damit die Anwendung für dieses Szenario geeignet ist, müssen mehrere Teile angepasst werden. Die Anwendung verwendet bereits OSM Daten, um weitere Features für die einzelnen Tankstellen zu ermitteln. Momentan werden hierfür lediglich Daten für Autobahnen, Schnell- und Bundesstraßen, Landes- und Kreisgrenzen verwendet. Diese Daten reichen für die erweiterten Anforderungen des Szenarios nicht aus.

Um die Routen zwischen den einzelnen Terminen zu ermitteln kann *PostgreSQL/PostGIS* durch *pgRouting*¹ erweitert werden. Diese Erweiterung benötigt als Grundlage einen gewichteten Graphen. Dieser Graph kann aus OSM-Daten erstellt werden.

¹<http://www.pgrouting.org>

Um eine Route zu ermitteln benötigt *pgRouting* jeweils die Id des Start- und Endknoten des Graphen. Deshalb muss es möglich sein von OSM-Geometrien auf die entsprechende Id des Knoten zu schließen. Da der Benutzer seine Route anhand von Adressen in das System eingibt müssen diese zuerst in Geokoordinaten umgewandelt werden. Dieser Vorgang nennt sich *reverse geocoding* und ist auch auf Basis von OSM-Daten möglich. Dazu kann die Software *Nominatim*² verwendet werden. Die Geokoordinaten der Adresse genügen zur Ermittlung der Ids jedoch noch nicht. In einem weiteren Schritt muss die nächste Geometrie, die eine Repräsentierung im Graphen besitzt, ermittelt werden. Zur Beschleunigung dieser Query empfiehlt sich der Einsatz eines spatial index.

Um den Suchraum für die Route einzuschränken, empfiehlt es sich nur einen Teil des Graphen auszuwerten. Dies kann erreicht werden indem nur Knoten und Kanten verwendet werden, deren Geometrie innerhalb einer Bounding Box liegen. Eine einfache Lösung zur Ermittlung der Bounding Box wäre es die Koordinaten des Start- und Endpunktes zu ermitteln und ein Rechteck um diese zu legen und anschließend um einen Puffer zu erweitern.

Nachdem die optimale Route zwischen den Terminen ermittelt wurde geht es im nächsten Schritt darum Tankstellen entlang dieser Route zu finden. Die Route besteht aus Kanten des gewichteten Graphen und muss zuerst wieder in eine Geometrie übersetzt werden. Mit dieser Geometrie ist es möglich Tankstellen in einer gewissen Entfernung über einen Join der Tankstellen Tabelle zu ermitteln. Die gewünschte maximale Entfernung kann dabei im Join-Prädikat angegeben werden.

Im nächsten Schritt werden für die Tankstellen die entsprechenden Knoten der Routing-Datenbasis ermittelt. Es kann sein, dass diese Knoten noch nicht im Subgraphen, der für die Ermittlung der Route verwendet wurde enthalten sind. In diesem Fall muss der Subgraph erweitert werden. Mit diesen Daten kann nun die zu erwartende Ankunftszeit an den Tankstellen ermittelt werden. Über die Ankunftszeit kann anschließend der Preis vorhergesagt werden.

Der Graph mit der Route, Tankstellen und Preisen kann anschließend ausgewertet werden, um eine optimale Tankstrategie zu ermitteln. Dazu kann der Algorithmus von Lin [21] verwendet werden. Der Algorithmus weist eine Zeitkomplexität von $O(n^3)$ auf, wobei n die Anzahl an Knoten im Graphen bezeichnet. Dies kann bei langen Routen zu einer sehr hohen Laufzeit führen. Es wäre zu prüfen, ob eine Optimierung durch die Verwendung von evolutionären Algorithmen zu schnelleren Ergebnissen ähnlicher Güte führt.

Im nächsten Abschnitt wird aufgezeigt welche Daten für den angepassten Algorithmus verwendet werden und wie diese aktuell gehalten werden können

5.3 Neuberechnung des Routing Graphen

Zur Berechnung der optimalen Route werden die OSM-Daten Deutschlands benötigt. Damit *pgRouting* mit den Daten umgehen kann müssen sie in einen gewichteten Graphen umgewandelt werden. Dies kann mit der Funktion `pgr_createTopology` erledigt werden. Abbildung ?? veranschaulicht diesen Vorgang.

Die OSM-Daten können inkrementell geupdatet werden. Hierzu stellt OSM sogenannte *changesets* zur Verfügung. Diese *changesets* können mit einem Werkzeug wie *osmosis*³ in die Datenbank eingelesen werden. Der gewichtete Graph muss bei jeder Änderung der Ursprungsdaten komplett neu berechnet werden.

²<http://www.nominatim.org>

³<https://wiki.openstreetmap.org/wiki/Osmosis>

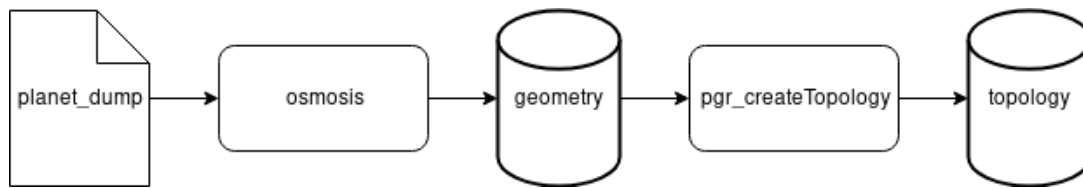


Abbildung 5.1: Initiales Laden von OSM Daten in die Datenbank und Erstellung eines gewichteten Graphen zur Routenberechnung

Die *changesets* sind zum Beispiel als Extensible Markup Language (XML)-Dateien mit einer minütlichen Auflösung erhältlich. Da eine minütliche Neuberechnung des gewichteten Graphen nicht praktikabel erscheint, können die *changesets* gesammelt und in einem täglichen oder wöchentlichen Intervall aufgespielt werden. Dieser Vorgang wird in Abbildung ?? gezeigt. Die *changesets* werden im *changeset buffer* zwischengespeichert. Täglich oder wöchentlich werden sie mit *Osmosis* in die Geometrie-Tabelle übernommen. Daraufhin wird der *changeset buffer* geleert und der gewichtete Graph wird durch *pgr_createTopology* neu erstellt. Der Parameter *clean* sorgt dafür, dass die entsprechende Tabelle davor geleert wird.

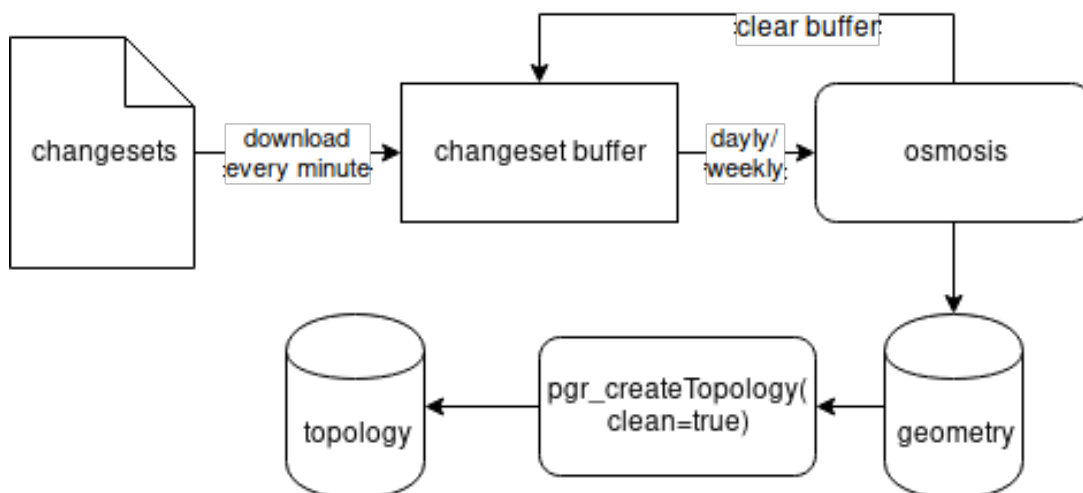


Abbildung 5.2: Update der OSM Geometrien und Neuberechnung des gewichteten Graphen

Neben den Daten für die Berechnung der Route werden noch die Preise für die Ermittlung der Tankstrategie benötigt. Der nächste Abschnitt bestimmt welche Daten dies betrifft und wie mit Änderungen an diesen Daten umgegangen wird.

5.4 Neuberechnung des Modells zur Preisvorhersage

Die historischen Benzinpreise der Tankstellen wurden mit Informationen zu Feiertagen, Schulferien Bundesland, Landkreis und den nächsten Hauptverkehrsstraßen erweitert. Mit diesen erweiterten Daten benötigte das Training der GBM über die historischen Daten in etwa 3 Stunden. Zum Training wurde ein Rechner mit einem AMD Threadripper 1950x, 64GB Arbeitsspeicher und einem SSD-Hintergrundspeicher verwendet.

Nach dem Training kann das Modell in Form einer Zip-Datei aus H2O extrahiert werden. Diese Zip-Datei wird zur Laufzeit in die Anwendung geladen und zur Vorhersage der Benzinpreise verwendet. Um immer die aktuellsten Entwicklungen auf dem Benzinmarkt zu berücksichtigen, sollte das Modell in regelmäßigen Zeitabständen neu trainiert werden. Dieser Vorgang kann losgelöst von der eigentlichen Anwendung ablaufen.

Die historischen Benzinpreise bestehen aus zwei Tabellen. Den Preisen an sich und Informationen zu den einzelnen Tankstellen. Dabei sind nicht alle Spalten der Tabellen für die Vorhersage relevant. Aus der Tabelle Preise wird das Datum, der Preis und die Id der Tankstelle benötigt. Über die Id wird die Tabelle Tankstellen mit den Preisen verbunden. Hier sind neben der Marke noch die Koordinaten der Tankstelle, also Latitude und Longitude, relevant.

Im nächsten Schritt wird das Datum der Preise in die Komponenten Jahr, Monat, Tag der Woche, Stunde und Tag des Monats zerlegt. Auf Basis der Koordinaten wird jeweils für die unterschiedlichen Arten an Hauptverkehrsstraßen ermittelt welche Hauptverkehrsstraße den geringsten Abstand zur Tankstelle hat. Dabei werden nur Hauptverkehrsstraßen innerhalb von 5 Km betrachtet. Auch auf Basis der Koordinaten wird das Bundesland und der Landkreis der Tankstelle ermittelt.

Nach der Aufbereitung der Daten kann letztendlich das Modell trainiert werden. Beim Erhalt von neuen Benzinpreisen muss nur das Datum zerlegt werden, da die Informationen zu den Tankstellen bereits vorliegen. Falls eine unbekannte Tankstelle in den Daten erscheint, werden die dazugehörigen Informationen für die Tankstelle berechnet und gespeichert.

Nach der Identifikation der sich ändernden und benötigten Daten kann eine Strategie entwickelt werden um die Anwendung für die Verwendung durch viele Benutzer zu skalieren. Diese Strategie wird im nächsten Abschnitt behandelt.

5.5 Entwicklung einer Architektur zur Skalierung der Anwendung

Bereits bei der Entwicklung der POC Version der Anwendung wurde darauf geachtet, dass die einzelnen Anfragen keine Nebeneffekte verursachen. In der Praxis bedeutet dies, dass eine Anfrage nicht den Zustand der Anwendung verändert. Das wird dadurch erreicht, dass die Anwendung nur lesend auf die Daten zugreift.

Dadurch ist es möglich die Anwendung horizontal zu skalieren. Horizontale Skalierung bedeutet, dass die maximale Anzahl gleichzeitiger Anfragen gesteigert werden kann, indem mehr Rechner dem System hinzugefügt werden.

Abbildung 5.3 zeigt eine mögliche Architektur, die diesen Sachverhalt ausnutzt, um die Anwendung für viele Anwender gleichzeitig nutzbar zu machen. Das System wird in die zwei Zonen "blue" und "green" unterteilt. Es ist jeweils nur eine der Zonen gleichzeitig aktiv. Eine Anfrage eines Benutzers zur Berechnung einer optimalen Route (**RoutingRequest**) wird von einem Load Balancer entgegen genommen. Der Load Balancer leitet die Anfrage an die momentan aktive Zone weiter.

Jede Zone besteht aus mehreren Application Servern, die jeweils die identische Version der Anwendung ausführen. Der Load Balancer entscheidet dabei an welchen der Application Server die Anfrage weitergeleitet wird und zieht dabei die momentane Auslastung der Application Server in Betracht. Zur Beantwortung der Anfrage benötigt ein Application Server die vorberechneten Daten wie zum Beispiel den gewichteten Routing Graphen. Diese Daten werden in PostgreSQL gespeichert.

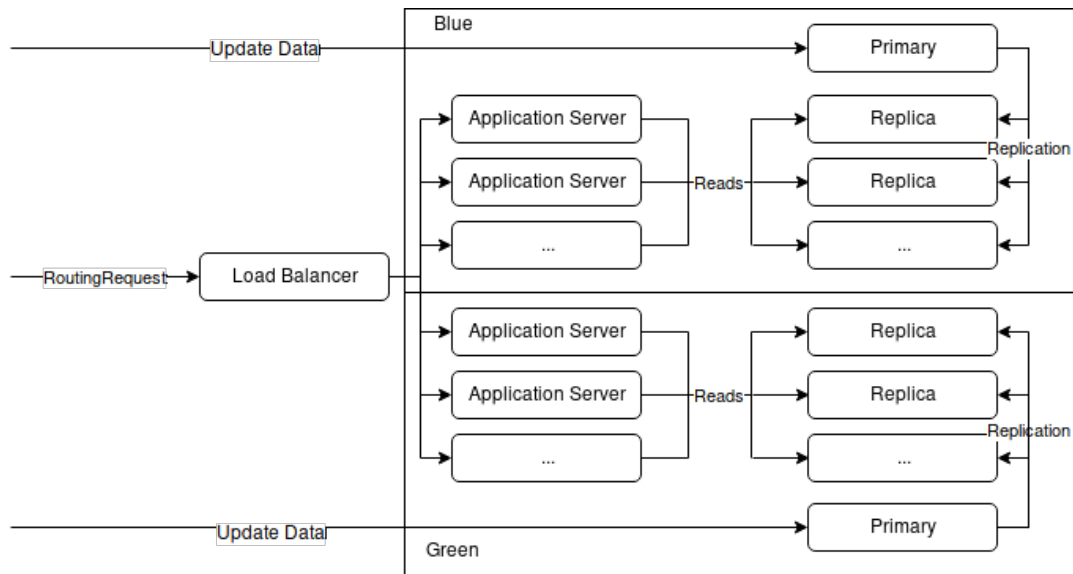


Abbildung 5.3: Horizontal skalierbare Architektur der Anwendung

Jede der Zonen beinhaltet einen PostgreSQL Cluster. Ein PostgreSQL Cluster besteht aus mehreren Rechnern, die jeweils eine Instanz der Datenbank beherbergen. Einer dieser Rechner wird als **Primary** bezeichnet. Sämtliche Änderungen an den Daten erfolgen auf diesem Rechner. Die Änderungen werden anschließend auf die anderen Rechner übertragen. Die anderen Rechner werden als **Replica** bezeichnet. Auf sie kann nur lesend zugegriffen werden.

Die Replikation der Daten kann zum Beispiel über die Replikation des Write Ahead Log (WAL) sichergestellt werden. Dieser Vorgang wird in im PostgreSQL Wiki beschrieben [22]. Das WAL beinhaltet alle Änderungen am Datenbestand. Bei einer Änderung wird die entsprechende Anweisung im WAL protokolliert. Erst bei Ausführung des Commits einer Datenbanktransaktion werden die geänderten Daten als sichtbar angesehen. Jede Änderung besteht aus einem oder mehreren Einträgen im WAL. Diese Einträge werden nicht geändert und es werden immer nur neue Einträge am Ende des Logs hinzugefügt.

Um den Datenbestand zu replizieren kann also das WAL des **Primary** Rechners an die **Replica** Rechner übertragen werden. Da das Log nur erweitert und nie geändert wird, kann dies sehr effizient umgesetzt werden.

Dieser Aufbau ermöglicht die horizontale Skalierung des Systems. Bei Einsatz entsprechender Monitoring-Werkzeuge können während des Betriebs Flaschenhälse identifiziert werden. Je nachdem, ob der Flaschenhals im Bereich der Application Server oder des PostgreSQL Clusters auftritt, können im entsprechenden Bereich mehr Rechner hinzugefügt werden. Durch die neuen Rechner kann der Flaschenhals beseitigt werden.

Die Aufteilung des Systems in die zwei Zonen hat den Vorteil, dass Änderungen am System immer in der gerade nicht aktiven Zone durchgeführt werden können. Bei Änderungen an den OSM Daten oder dem Erhalt neuer historischer Preisdaten, kann so die Neuberechnung der Daten erfolgen ohne, dass das laufende System beeinträchtigt wird. Sobald die Berechnung fertig ist, kann der Load Balancer die aktuell aktive Zone wechseln und den Benutzern stehen aktuelle Daten zur Verfügung.

Mit dem hier beschriebenen System wäre es möglich das Anwendungsszenario zu erfüllen und

die Funktionalität der Anwendung vielen simultanen Benutzern zur Verfügung zu stellen. Die Aufgabenstellung lässt jedoch noch viele weitere sinnvolle Erweiterungen zu. Diese Erweiterungen werden im Folgenden beschrieben.

5.6 Weitere Ansatzpunkte zur Erweiterung des Systems

Dieser Abschnitt beschreibt weitere Ansatzpunkte, um das System zu erweitern und den Nutzen für die Benutzer dadurch zu erweitern. Die heutige Allgegenwärtigkeit von Smartphones erlaubt kontinuierliche Rückmeldungen an das System. Der Benutzer könnte sowohl den realen Verbrauch seines Fahrzeugs, als auch die tatsächlichen Benzinpreise an das System übermitteln.

Mit diesen Daten kann die Routenplanung und die Preisvorhersage verbessert werden. Diese Daten erlauben nicht nur eine Verbesserung der allgemeinen Vorhersagegüte, sondern erlauben auch eine individuelle Anpassung an den Benutzer.

Die hier vorgestellte Architektur berücksichtigt dies nicht und wäre bei der Implementierung dieser Features erneut zu evaluieren.

6. 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 [23] wirft die Überlegung in den Raum, ob Intellitank auch hierauf anwendbar ist. Mit dieser Investition soll auch ein einheitliches Bezahlungssystem eingeführt werden. Laut einer Untersuchung der ISPEX AG [24] 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$ java -jar routingService-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 timeout for connections. (Die Methode org.postgresql.jdbc.PgConnection.getNetworkTimeout() ist noch nicht implementiert.)
[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.Slf4jLog
[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.

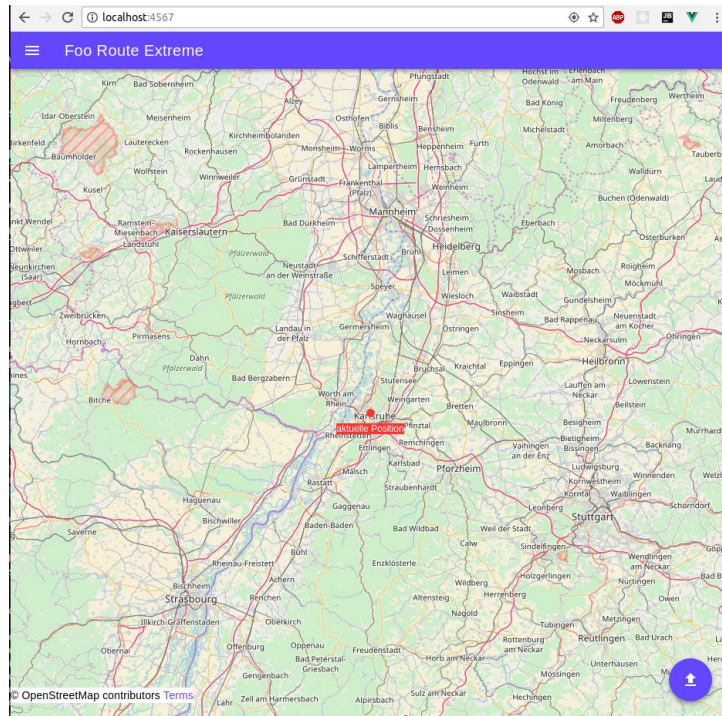


Abbildung A.2: grundlegende Oberfläche der Webanwendung

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.

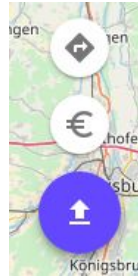


Abbildung A.3: die Benutzer-Interaktionsmöglichkeiten

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.

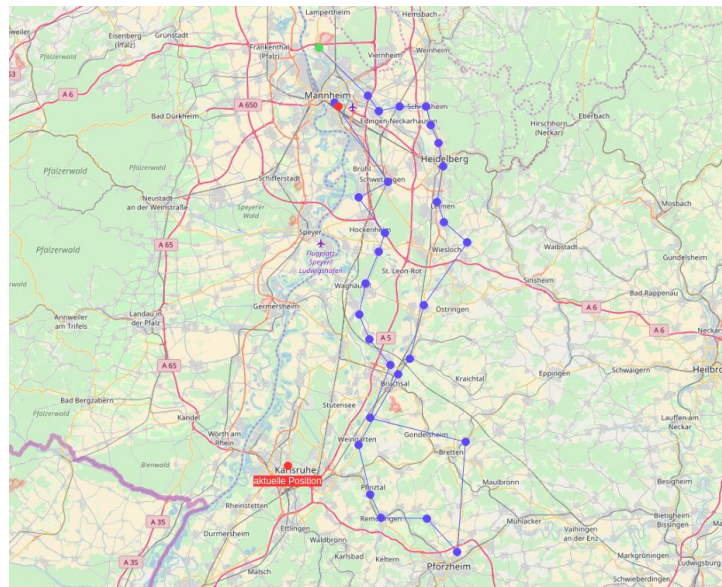


Abbildung A.4: Anzeige einer Tankroute

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.

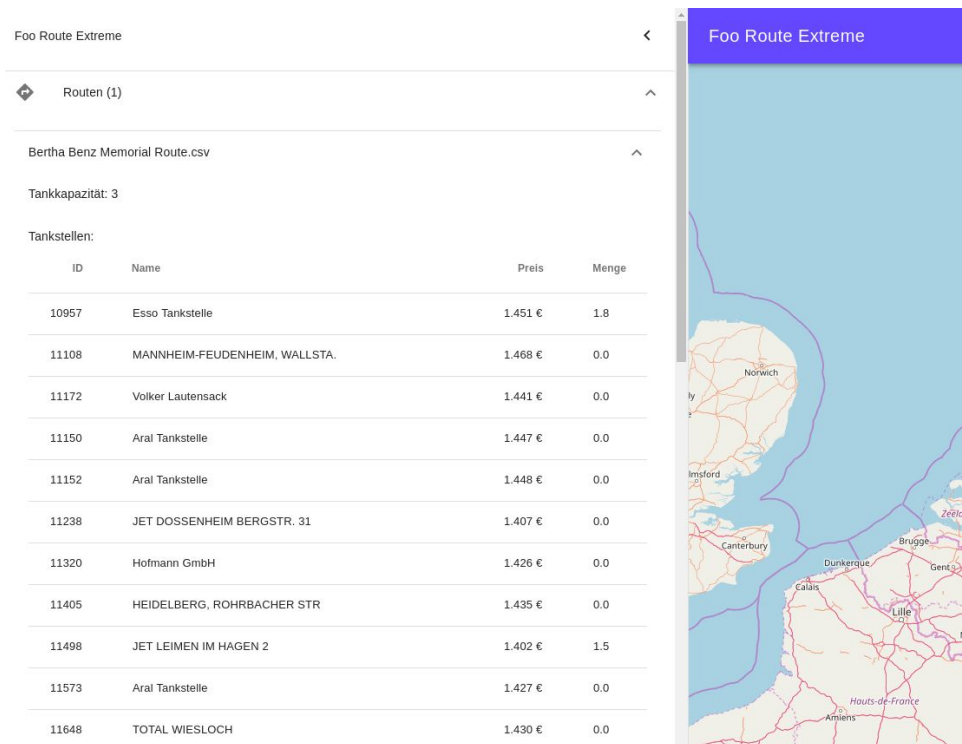


Abbildung A.5: die errechneten Tankfüllungen für die Route

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.

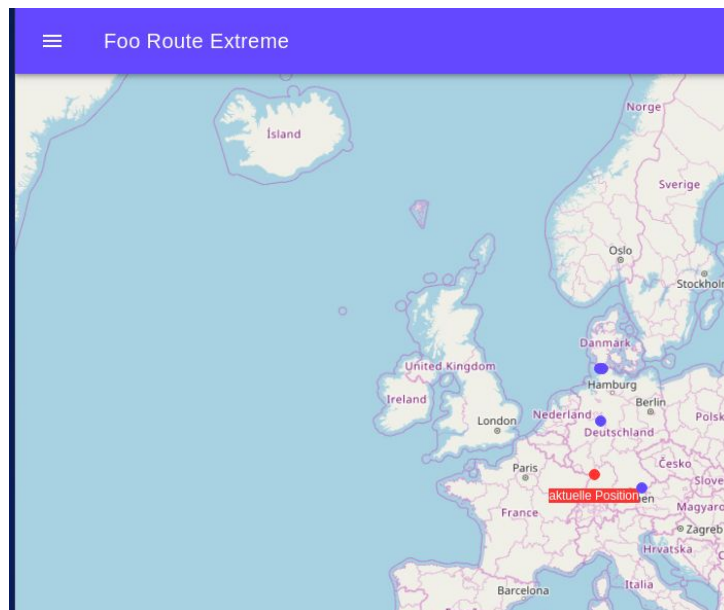


Abbildung A.6: die Tankstellen auf der Karte bei einer Preisvorhersage

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

Foo Route Extreme			
€ Preisvorhersagen (1)			
price-prediction.csv			
Preisvorhersagen:			
ID	Name	Vorhersagezeitpunkt	Preis
24	team Tankstelle Niebüll	2015-02-15T20:18:01Z	1.334 €
46	Aral Tankstelle	2016-03-22T09:43:01Z	1.290 €
14038	Auto Hauser	2016-02-26T17:06:01Z	1.230 €
4160	ELAN Lemgoer Strasse	2015-06-12T05:50:02Z	1.524 €
DOWNLOAD ENTFERNEN			

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://www.it.usyd.edu.au/~mestre/papers/gas-station.pdf>.
- [3] A. M. S. KHULLER und J. MESTRE, <http://www.cs.umd.edu/projects/gas/>, letzter Zugriff: 18.01.2018.
- [4] *Haversine formula*, https://rosettacode.org/wiki/Haversine_formula, letzter Zugriff: 19.01.2018.
- [5] ENZYKLO, *Optimalitätsprinzip von Bellman*, <http://www.enzyklo.de/Begriff/Optimalit%C3%A4tsprinzip%20von%20Bellman>.
- [6] A. B. DOWNEY, *Think Stats*. O'Reilly Media, Inc., 2011, ISBN: 1449307116, 9781449307110.
- [7] P. LIPPE, *Deskriptive Statistik*. Stuttgart Jena: G. Fischer, 1993, ISBN: 3-8252-1632-2.
- [8] 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].
- [9] G. BONACCORSO, *Machine Learning Algorithms: A Reference Guide to Popular Algorithms for Data Science and Machine Learning*. Packt Publishing, 2017, ISBN: 1785889621, 9781785889622.
- [10] M. NOTTINGHAM, *Problem Details for HTTP APIs*, <https://tools.ietf.org/html/draft-nottingham-http-problem-06>, letzter Zugriff: 19.01.2018, 2014.
- [11] 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.
- [12] DR-PRODIGY, *holidays-Python Package*, <https://pypi.org/project/holidays/>.
- [13] SCHULFERIEN.ORG, <https://www.schulferien.org/>, letzter Zugriff: 18.01.2018.
- [14] OPENSTREETMAP.ORG, *OpenStreetMap*, <http://www.openstreetmap.org>, letzter Zugriff: 18.01.2018.
- [15] 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.
- [16] 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.
 - [17] 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>.
 - [18] D. ACATAY, <http://dacatay.com/data-science/part-3-time-series-stationarity-python/>, letzter Zugriff: 18.01.2018.
 - [19] J. BROWNLEE, <https://machinelearningmastery.com/gentle-introduction-autocorrelation-partial-autocorrelation/>, letzter Zugriff: 18.01.2018.
 - [20] *MAE and RMSE - Which Metric is Better?*, letzter Zugriff: 19.05.2018. Adresse: <https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d>.
 - [21] S.-H. LIN, „Finding Optimal Refueling Policies in Transportation Networks“, in *Algorithmic Aspects in Information and Management*, R. FLEISCHER und J. XU, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 280–291, ISBN: 978-3-540-68880-8.
 - [22] *Streaming Replication*, https://wiki.postgresql.org/wiki/Streaming_Replication, letzter Zugriff: 21.05.2018, 2018.
 - [23] 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>.
 - [24] *Energiemarkt Kommentar: Ölpreis dominiert alle Energiepreise*, <https://www.ispex.de/energiemarkt-kommentar-02-2016-oelpreis-dominiert-alle-energiepreise/>, letzter Zugriff: 20.01.2018, 2016.