

TCP: TrAIn_Connection_Prediction

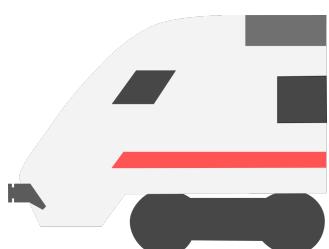
Autor: Marius De Kuthy Meurers & Theo Döllman

Webseite: <https://TrainConnectionPrediction.github.io>

GitHub: <https://github.com/TrAInConnectionPrediction/tcp>

Datum: 5. Februar 2021

Version: 3.0



Inhaltsverzeichnis

1	Einführung	1
1.1	Warum können Züge verspätet sein?	1
2	Datensammlung	1
2.1	Zuglaufinformationen	1
2.2	Streckennetz	3
3	Datenanalyse	5
4	Maschinelles Lernen	8
4.1	Vorüberlegungen	8
4.2	Verbindungsbewertung	8
4.3	Feature Importance	12
4.4	Klassifikationsgenauigkeit	13
5	Verbindungsbewertung	13
5.1	Vergleich und Evaluierung	13
6	Webseite	13
6.1	Warum eine Webseite?	13
6.2	Backend	14
6.3	Frontend	14
6.4	Kommunikation Frontend-Backend	15
7	Zusammenfassung	15

Kapitel 1 Einführung

Die Bahn bietet eine statische Reiseplanung an, die auf den Zeiten im Fahrplan basiert. Aber wie oft hat man bei einer Verbindung schon den Anschluss verpasst und musste sich einen neuen suchen?

Wir haben in unserem Projekt eine KI-basiertes System entwickelt, das basierend auf historischen Zuglaufinformationen vorhersagt, wie wahrscheinlich es bei einer Zugverbindung ist, alle Anschlusszüge zu erreichen und einem damit hilft die beste Verbindung zu finden.

Eine besondere Schwierigkeit dabei war, die benötigten Daten zu bekommen, da diese von der Bahn nicht veröffentlicht werden.

Als Vision ließe sich mit unserem Zugverbindungsverhersagesystem der öffentliche Verkehr verlässlicher machen und somit der Umstieg vom Auto auf die Bahn für mehr Menschen attraktiver gestalten. Eine solche KI unterstützt also die Verkehrswende und trägt somit zum Umweltschutz bei.

1.1 Warum können Züge verspätet sein?

Als erstes mussten wir uns überlegen, weshalb Züge überhaupt verspätet sein können. Grundsätzlich gibt es einen Hauptgrund dafür: Die Kapazität des Schienennetzes ist limitiert, also es kommt auch bei der Bahn zu Staus. Dadurch müssen Züge langsamer fahren oder ihre Aufenthaltszeit im Bahnhof verlängern. Für eine Kapazitätsverringerung gibt es wiederum einige primäre Gründe: Langsamfahrstellen, Umleitungen oder Streckensperrungen. Aber auch sekundäre Einflüsse wie defekte Triebfahrzeuge oder verlängerte Einstiegszeiten durch kaputte Zugtüren, können durch den Blockbetrieb im Bahnnetz wiederum andere Züge aufhalten - ähnlich wie bei einem Stau auf der Autobahn. Einige davon sind für die von uns geplanten Verspätungsvorhersagen geeignet, andere eher weniger. Statistisch gesehen, je weiter ein Zug laut seinem Fahrplan gefahren ist, desto verspäteter ist er (siehe Kapitel 3). Personalausfall ist eine weitere Ursache für Streckenengpässe. Es gibt aber keine Möglichkeit, im Voraus zu wissen, ob eine Person ausfallen wird. Das Wetter ist so ein Zwischending: Bei extremen Wetterbedingungen werden durch z.B. beschädigten Oberleitung Streckensperrungen vorgenommen. Aber für eine Zugverspätungsvorhersage benötigen wir dann natürlich auch eine Wettervorhersage, und die ist, insbesondere für Vorhersagen über drei Tage hinaus, von eingeschränkter Qualität.

Kapitel 2 Datensammlung

Um ein System zur Vorhersage von Zugverspätungen, wie oben beschrieben, zu entwickeln, braucht man historische Echtzeitdaten von Ankunfts- und Abfahrtszeiten der Züge über einen längeren Zeitraum. Des Weiteren braucht man genügend Daten über die Faktoren, die zu Verspätungen führen, wie Informationen über das Streckennetz.

Es gibt leider kaum öffentlich zugängliche Datensätze, die für dieses Projekt in Frage kommen. Diese Daten mussten also alle aus geeigneten Quellen von uns selber zusammengestellt werden.

2.1 Zuglaufinformationen

Die Deutsche Bahn selber veröffentlicht keine historischen Zuglaufinformationen. Das einzige, was sie öffentlich zugänglich machen, ist eine durchschnittliche Prozentzahl pünktlich kommender Züge pro Monat und Jahr, jeweils

im Nah- und Fernverkehr. Diese sind interessant, um die eigenen Daten zu überprüfen, können aber sonst kaum verwendet werden.

Nachdem wir 2019 knapp 2 Mio. Datenpunkte über die Webseite [zugfinder.de](#) erhalten haben, haben wir im Jahr 2020 eigene Datenquellen ergeschlossen: Wir machen es grundsätzlich ganz ähnlich, wie es David Kriesel in seinem Talk “Die Pünktlichkeit ist eine Zier” ([Kriesel, 2020](#)) gemacht hat, die Idee hatten wir aber schon bevor er seinen Talk gehalten hat. Das Interne Reisenden-Informationssystem¹ (IRIS) der Bahn hat eine öffentliche XML-Schnittstelle. Diese Scheint noch nicht einmal ein Rate-Limit zu haben.

Mittels IRIS bekommt man folgende Informationen:

- Fahrplan pro Stunde und Bahnhof
- Alle bekannten Fahrplanänderungen pro Bahnhof
- Nur die in den letzten 3 min neu bekannt gewordenen Fahrplanänderungen pro Bahnhof

Die Fahrpläne enthalten dabei jeweils für Ankunft und Abfahrt des Zuges unter anderem folgende Informationen:

- Den Pfad, also die schon befahrenen und noch abzufahrenden Stationen
- Das Datum und die Uhrzeit
- Das Gleis (kann auch z.B. '3a' sein, wenn das Gleis doppelt belegt wird)
- Die Line (z.B. S8)

Die Fahrplanänderungen enthalten die dazu passenden Änderungen:

- Den geänderten Pfad
- Die geänderte Uhrzeit und das geänderte Datum
- Das geänderte Gleis

Ein Python-Script lädt jede Stunde die Fahrpläne aller deutschen Bahnhöfe herunter, wandelt das XML in JSON um und speichert dieses in einer Postgresql Datenbank. Anfangs haben wir so auch jede Stunde die bekannten Änderungen heruntergeladen, jedoch mussten wir bei unseren ersten Analysen (Fig: [3.1](#)). feststellen, dass das IRIS Verspätungsdaten kürzer zwischenspeichert als wir gedacht hatten. Daher sind wir dazu übergegangen, alle 2 - 3 Minuten die Fahrplanänderungen der letzten 3 Minuten jedes Bahnhofs herunterzuladen. Bei 7606 Bahnhöfen macht das 5.476.320 Requests pro Tag. Dafür reicht der Jetson Nano², den wir bisher verwendet hatten, nicht mehr aus. Wir haben das System daher auf unseren Datenbankserver umgezogen, den wir netterweise von der Schule bekommen haben und auch dort betreiben durften.

Wir sammeln so im Monat über 15. Mio Datenpunkte. Die Datenbank ist mittlerweile über 100 GB groß. Da wir neben einfacheren Analysen auch komplexe Verbindungs berechnungen machen, scheidet SQL als einziges Tool zur Analyse aus. Unsere Alten Datensatz hatten wir mit Pandas ([pandas development team, 2020](#)) analysiert, aber dafür reicht der Arbeitsspeicher jetzt nicht mehr aus. Wir verwenden deshalb Dask ([Dask Development Team, 2020](#)). Dask ist ein ganz gutes Framework für “larger-than-memory Operations” mit der Pandas API, hat aber leider auch viele Bugs und ist nicht so intuitiv gestaltet wie man es wünschen würde.

Die Daten sind so, wie sie von den Crawlern in die Datenbank geladen werden, noch nicht verwendbar. Sie müssen erst geparsed werden. Da wir nicht jeden Tag alle Daten neu Parsen können, und jederzeit noch weitere Änderungen für schon geparseste Daten bekannt werden können, müssen wir in der Datenbank auch schon geparsete Datenpunkte

¹Für Tübingen Hbf: <https://iris.noncd.db.de/wbt/js/index.html?bf=TT&Zeilen=25>

²Ein Raspberry Pi mit NVIDIA GPU

ändern können. Für dieses sogenannte “upserting” benutzen wir Pangres³, da Pandas dieses SQL-Feature immer noch nicht unterstützt. Dass das klappt, braucht jeder Datenpunkt einen eindeutigen Index. Das IRIS gibt jedem Datenpunkt eine eindeutige ID, diese ist aber in Textform. Das ist zwar für Pangres kein Problem, aber Dask kann nur mit einem Integer als Index umgehen. Daher erstellen wir mit Cityhash⁴ einen 64-bit Hash der vom IRIS erzeugten ID. Dieser muss dann noch von unsigned in signed umgewandelt werden, da PostgresQLs größter Interger Datentyp BIGINT (64-bit signed) ist, und Cityhash aber 64-bit unsigned benutzt.

Das gesamte Crawlen wird von einem Discord⁵ Bot überwacht, der eine Warnung sendet, sobald zu wenige Daten eintreffen. Dieser hat sich auch als ganz nützlich erwiesen, z.B. als die Festplatte von unserem Datenbankserver eines Tages voll war.

2.2 Streckennetz

Im letzten Jahr hatten wir gelernt, dass für die Vorhersagen die Länge der von einem Zug bereits zurückgelegte Strecke besonders wichtig ist. Bei 300 Bahnhöfen war es noch möglich, die Abstände zwischen den Bahnhöfen manuell hinzuzufügen. Mit den neuen Daten, hätten wir aber manuell 30.000 Eingaben machen müssen. Bei optimistischen 10 Sekunden für jede Eingabe, sind das ca. 80h manuelle Arbeit, plus extra Zeit für unerwartete Besonderheiten bei der Bahn. Wir brauchen also ein routingfähiges Streckennetz, um die Abstände automatisch zu errechnen.

Auf dem OpenData-Portal⁶ der Deutschen Bahn gibt es nur ein nicht routingfähiges Streckennetz ohne Privatbahnenstrecken, die API des Trassenfinders⁷ gibt uns zwar die Strecke zwischen Betriebsstellen⁸, aber nur im DB Netze Teil des Streckennetzes und nicht alle Bahnhöfe sind klar auf Betriebsstellen zu mappen. Delfi⁹ veröffentlicht GTFS¹⁰ mit shapes.txt¹¹, aber nicht für Sachsen und Schleswig-Holstein, und eine Anfrage an das Eisenbahnbusdamsamt über das Informationsfreiheitsgesetz war erfolglos, da “die angefragten Informationen nicht vorliegen.” Nach einiger Zeit sind wir auf das Infrastrukturregister¹² von DB Netze gestoßen. Wir haben die Request der Website zum Backend analysiert und so herausgefunden, dass das Backend ein OpenSource Geoserver¹³ ist. Dieser hat natürlich eine öffentliche Dokumentation und kann einem auch nicht nur die Karte einfach so anzeigen. Einer der in der Dokumentation aufgelisteten Datentypen, die einem der Geoserver geben kann, ist kml¹⁴. So kann man mit einer Anfrage eine kml-Datei aller Strecken in Deutschland inklusive Privatbahnen herunterladen.¹⁵. Das Problem: Die Namen der Bahnhöfe im Infrastrukturregister sind anders als die, die den Fahrgästen angezeigt werden und die wir

³<https://pypi.org/project/pangres/>

⁴Non cryptographic hash library by Google <https://github.com/google/cityhash>

⁵Chatprogramm, ähnlich wie Slack, dass ursprünglich für Gamer entwickelt wurde

⁶<https://data.deutschebahn.com/>

⁷<https://trassenfinder.de/>

⁸<https://de.wikipedia.org/wiki/Betriebsstelle>

⁹Durchgängige Elektronische FahrgastInformation: https://www.opendata-oepnv.de/ht/de/organisation/delfi/startseite?tx_vrrkit_view%5Bdataset_name%5D=deutschlandweite-sollfahrplandaten-gtfs&tx_vrrkit_view%5Baction%5D=details&tx_vrrkit_view%5Bcontroller%5D=View

¹⁰General Transit Feed Specification

¹¹Die Datei in GTFS, die geographische Informationen enthält

¹²<https://geovdbn.deutschebahn.com/isr>

¹³<http://geoserver.org/>

¹⁴Keyhole Markup Language: https://en.wikipedia.org/wiki/Keyhole_Markup_Language

¹⁵ACHTUNG: DOWNLOAD 500 MB GROSS https://geovdbn.deutschebahn.com/pgv-map/geoserver.action?LAYERS=ISR%3AISR_V_GEO_TEN_KLASSIFIZIERUNG&TRANSPARENT=TRUE&FORMAT=kml&VERSION=1.1.1&TILED=false&USERDEFINEDSLD=&SERVICE=WMS&REQUEST=GetMap&VIEWPARAMS=ZEITSCHIEIBE%3AUNDEFINED%3BLANG%3ADE%3BALG_DBNETZ_STRECKE%3Aalle%20Strecken%3BAJHR%3A2021&SRS=EPSG%3A31467&BBOX=3250000,5200000,3950000,6100000&WIDTH=700&HEIGHT=900

haben. Also müssen die einen Bahnhofsnamen auf die anderen gemappt werden. Wir haben dazu zwei verschiedene Methoden benutzt:

- Der Abstand zwischen den Bahnhöfen in km
- “gestalt pattern matching”¹⁶ der Bahnhofsnamen

Leider fehlen am Ende trotzdem ca. 2000 Bahnhöfe (25%). Wir können also zwischen vielen Bahnhöfen den Abstand ausrechnen, aber bei vielen Anfragen würden wir die Streckenlänge zwischen den Bahnhöfen nicht feststellen können.

Letztendlich haben wir daher folgendes gemacht: Wir haben mit Hilfe von osmnx (Boeing, 2017) einen mathematischen Graphen des Schienennetzes von OpenStreetMaps heruntergeladen. Die Ecken des Graphen, sind dabei aber immer Weichen und ähnliches, also nicht da, wo die Bahnhöfe sind. Die eigentliche Aufgabe besteht daher darin, die Bahnhöfe als Ecken in den Graphen einzufügen. Hierfür haben wir einen Algorithmus entwickelt:

1. Finde zu jedem Bahnhof die nächste Kante
2. Finde den dem Bahnhof nächsten Punkt auf dieser Kante
3. Konstruiere einen Vektor \vec{k} vom Bahnhof zum nächsten Punkt
4. Konstruiere einen Vektor \vec{o} orthogonal zu \vec{k}
5. Erstelle mit Hilfe von \vec{k} und \vec{o} ein Kreuz mit dem Bahnhof als Mittelpunkt
6. Finde alle Schnittpunkte des Kreuzes mit den umliegenden Kanten
7. Füge bei den Schnittpunkten eine den Bahnhof als extra Ecke ein

Mit diesem Algorithmus kann man jetzt die Schnittpunkte der Gleise mit dem Bahnhof bestimmen, wie z.B. hier der Algorithmus angewendet auf den Tübinger Hauptbahnhof zeigt: **Tübingen Hbf**

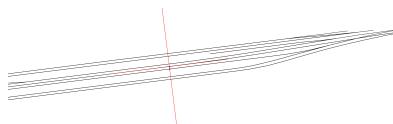


Abbildung 2.1: Tübingen Hbf

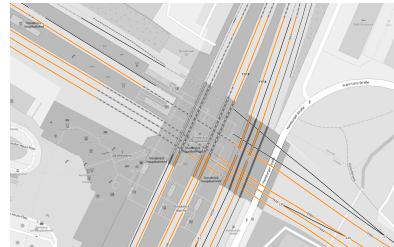


Abbildung 2.2: Osnabrück Hbf¹⁷



Abbildung 2.3: Von Tübingen Hbf nach Altingen(Württ)

Aber warum überhaupt ein Kreuz? Würde nicht eine einzelne Linie nach \vec{k} reichen? Wie bei so vielen Dingen mit der Bahn ist das mal wieder nicht ganz so einfach: Wie hier zu sehen ist, gibt es auch sogenannte Kreuzungsbahnhöfe wie z.B. Onsabrück Hbf(Siehe . Da hier alle Gleise zu dem Bahnhof zählen, würde mit einer einfachen Linie nur c.a. die Hälfte der Gleise in unserer Karte zu dem Bahnhof zählen.

Das Ergebnis lässt sich auf jeden Fall sehen. Und auch wenn wir in der Produktion nur noch ein vereinfachtes Streckennetz verwenden, könnten wir theoretisch auch wie hier die gefahrene Strecke plotten, wie in **Von Tübingen Hbf nach Altingen(Württ)**

¹⁶https://en.wikipedia.org/wiki/Gestalt_Pattern_Matching

¹⁷Screenshot von <https://www.openrailwaymap.org/>

Kapitel 3 Datenanalyse

Nach der aufwendigen Datensammlung konnten wir durch unsere Analyse den Datensatz durchleuchten und validieren, teilweise aber auch Auffälligkeiten feststellen. Wegen der besseren Anschaulichkeit haben wir alle Analysen primär in Form von Plots (Graphen) dargestellt.

Die Plots gibt es in voller Größe in unsere Repository auf GitHub:

<https://github.com/TrAInConnectionPrediction/tcp/blob/master/docs/analysis.md>

Bei allen Analysen gruppieren wir die Daten mittels group_by()¹. Normalerweise wäre es damit ganz einfach, aber bei der Größe des Datensatzes dauert dies, obwohl es von Dask für parallele Ausführung optimiert ist, ca. eine Stunde.

Als erstes haben wir die Verspätung im Verlauf der Stunde analysiert.

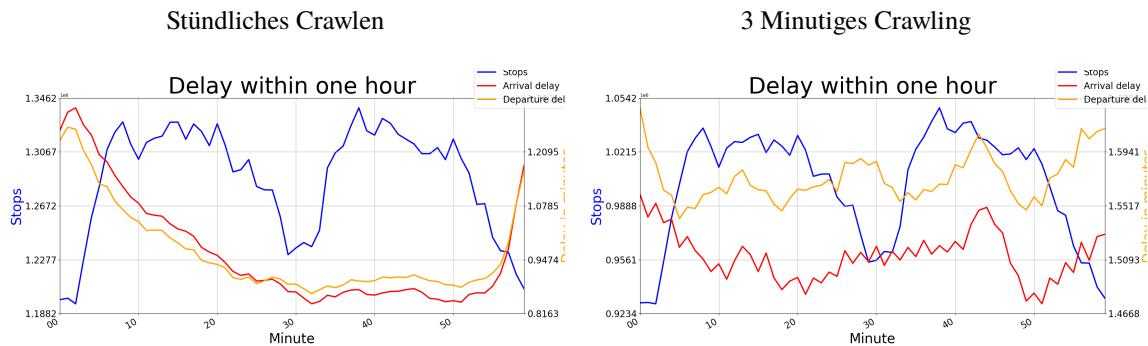


Abbildung 3.1: Stündliches Crawling reicht nicht

Die rote und die orangene Kurve ist jeweils die Verspätung bei Ankunft und Abfahrt. Es fällt auf, dass bei den stündlich geholten Daten die Verspätung um die ganze Stunde deutlich höher ist. Das ist einfach damit zu erklären, dass unser Programm immer zur vollen Stunde die aktuellen Verspätungsinformationen heruntergeladen hat. Wir dachten, dass das IRIS auch noch Änderungen von Zügen enthält, die vor einer Stunde abgefahren sind, da es nicht anders in der Dokumentation der API steht und da es David Kriesel in seinem Bahn-Mining Projekt genau gleich gemacht hat. Die Analyse zeigt aber, dass die Verspätungsdaten nicht so lange gecached werden und wir sie öfters holen müssen. Mit dem Crawling alle 2-3 Minuten haben die Verspätungskurven keine solchen Auffälligkeiten mehr.

Die blaue Kurve ist die Anzahl der Arrival- und Departure Events pro Minute in Millionen in allen unseren Daten. Hier springt ins Auge, dass anscheinend zur ganzen und halben Stunde weniger solche Events stattfinden. Das hat uns selber etwas gewundert. Die beste Vermutung von uns und von DB-Analytics (dort haben wir auch nachgefragt) ist, dass dieses Muster vom "Deutschlandtakt" kommt. Aber nicht einmal DB-Analytics konnte das sicher bestätigen.

Weiter haben wir uns die selben Datenfeatures über einen Tag und über die Woche angeschaut.

Über den Tag verteilt ist der Berufsverkehr am Morgen und Abend sowohl in der Verspätung als auch in der Anzahl der Fahrten zu erkennen. Und bei der Wochenübersicht sieht man auf einen Blick das Wochenende: Der Berufsverkehr am Morgen fehlt komplett und dadurch ist auch insgesamt weniger los.

¹Ganz ähnlich wie ein GROUP BY in Sql

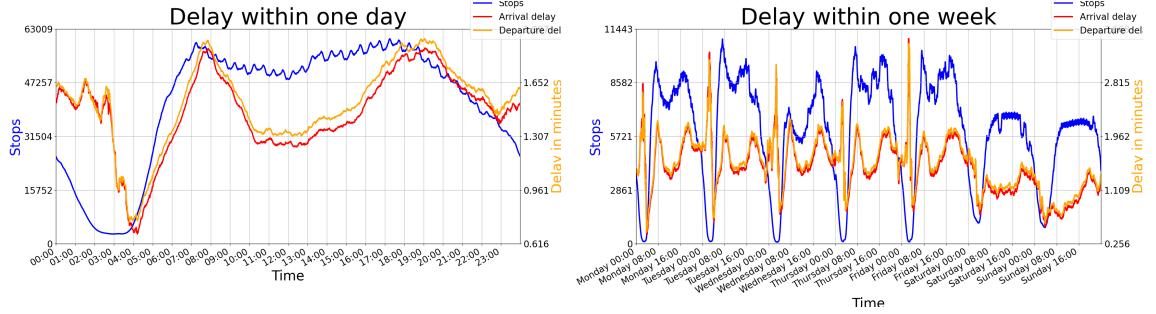


Abbildung 3.2: Berufsverkehr

Plottet man die gesamten Daten, fällt neben dem Wochenmuster vor allem auf, wo unser System nicht richtig gearbeitet hat und keine Daten gesammelt wurden, wie in Abbildung 3.4 zu sehen ist. Ende November wurde unserem Server beim Umpatchen ausversehen der Strom gezogen und dadurch die dann die Festplatte beschädigt. Im Dezember wurde unser Server leider in der Schule von der Firewall geblockt. Wir mussten daher von der Schule ins SFZ Eningen umziehen.

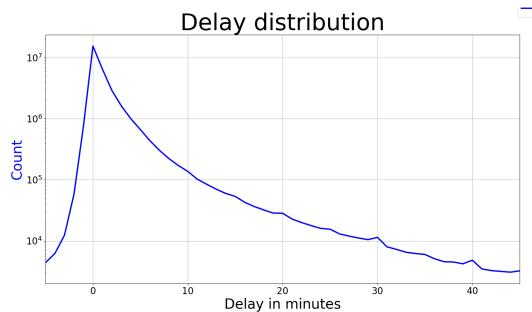


Abbildung 3.3: Verspätungsverteilung

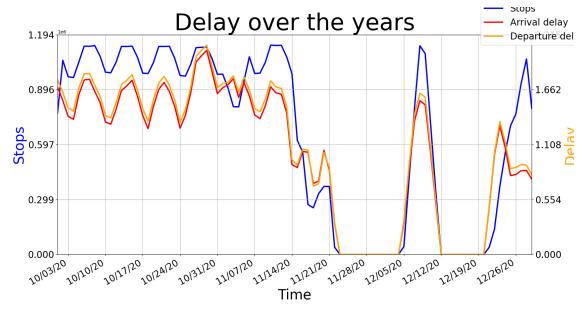


Abbildung 3.4: Datenvollständigkeit

Weitere interessante Entdeckungen kann man machen, wenn man einen Blick auf die Verspätungsverteilung in Abbildung 3.3 wirft. Es scheint besonders viele Züge zu geben, die genau 5, 10, 15, 20, 25, 30, ... Minuten Verspätung haben. Wir vermuten, dass diese Informationen teilweise tatsächlich im IRIS nicht genauer vorliegen. Auf den Bahnhofstafeln wird ohnehin nur die Verspätung in 5 Minuten-Schritten angezeigt.

Als nächstes haben wir die geographische Verteilung von Verspätungen und Zughalten visualisiert. Die Größe der Kreise entspricht der Anzahl der Halten und die Farbe der Punktlichkeit.

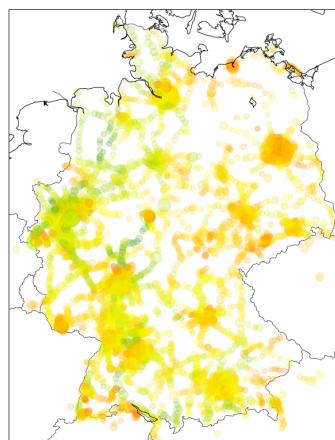


Abbildung 3.5: Geographische Verspätungsverteilung (rot: viel Verspätung; grün: wenig Verspätung, kontinuierliche Skala)

Im Osten fahren deutlich weniger Züge als im Westen, wie die geringere Anzahl an Kreisen und ihre kleinere Fläche zeigt. Wie die rötlichere Farbe der Verbindungen im Osten zeigen fahren sie dort gegenüber den grüneren Verbindungen im Westen augescheinlich auch mit mehr Verspätung. Im Ruhrgebiet gibt es einige Zugstrecken, auf denen die Züge besonders pünktlich fahren.

Weiter haben wir uns dafür interessiert, was für einen Einfluss die Zuggattung hat. Die Kreisgröße entspricht wieder der Anzahl der Halte und die Farbe ist links die Verspätung (Rot -> viel Verspätung; Grün -> wenig Verspätung) und rechts der Prozentsatz an ausgefallenen Halten.² (Rot -> viele Ausfälle; Grün -> wenige Ausfälle)

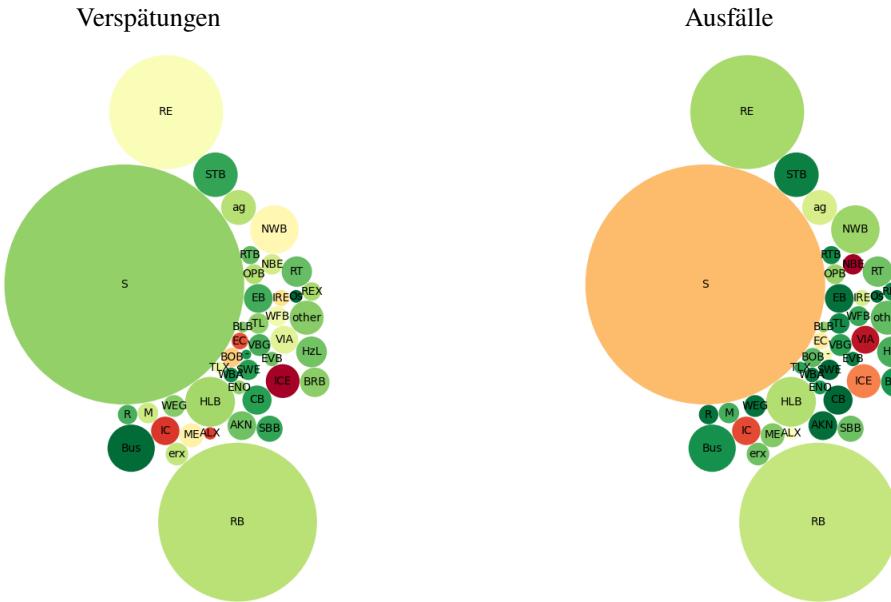


Abbildung 3.6: Pro Zugtyp

Besonders der Fernverkehr ist rot und damit verspätet. Zu Bussen bekommen wir keine Verspätungsinformationen. Diese sind daher grün und damit pünktlich. Auch Ausfälle sind in der Regel beim Fernverkehr am häufigsten, obwohl es hier auch Ausreißer wie die Nordbahn (NBE, rot) gibt, bei der uns der Kundenservice bestätigen konnte, dass es in der letzten Zeit viel ausgefallen ist, unter anderem wegen einer Güterzugentgleisung und heftigem Vandalismus in einem Fahrzeug der Linie RB 63.

Ein weiterer sehr wichtiger Faktor für Verspätungen ist die bereits gefahrene Strecke der Züge. Diese fasst viele Faktoren zusammen, die wir einzeln gar nicht erfassen können. In Abbildung 3.7 wird klar, dass die durchschnittliche Verspätung steigt, wenn ein Zug schon mehr Halte absolviert hat.

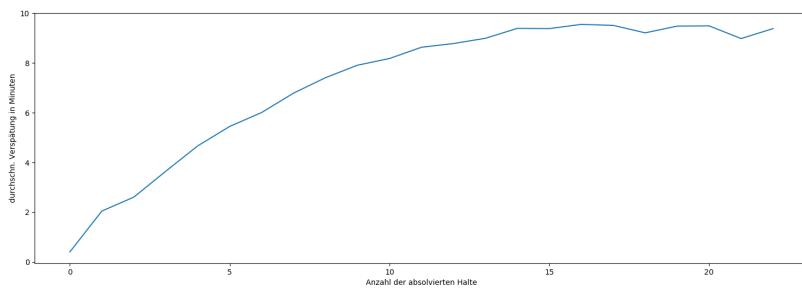


Abbildung 3.7: Verspätung und Streckenlänge

²Um einen solchen "packed bubble chart" in Python zu erstellen, gibt es übrigens keine geeigneten Tools, also mussten wir auch das selber schreiben. Das Beispiel bei matplotlib kommt von uns https://matplotlib.org/devdocs/gallery/misc/packed_bubbles.html

Kapitel 4 Maschinelles Lernen

4.1 Vorüberlegungen

Unser Ziel ist die Bewertung einer Zugverbindung basierend auf den Anschlussicherheiten. Maßgeblich dafür sind die einzelnen Umstiege. Eine Direktverbindung kann von uns nur als “gelingt immer” bewertet werden, da per Definition keine Anschlusszüge verpasst werden können. Hat eine Verbindung mehr als einen Umstieg, so wird bei jedem Umstieg einzeln die Wahrscheinlichkeit berechnet, dass dieser gelingt. Das Produkt all dieser Werte bezeichnen wir dann als **Verbindungsscore**. Der Name kommt daher, dass der Verbindungsscore zum Ranking dienen soll und nicht als Garantie, dass eine Verbindung zu z.B. 90% der Zeit funktioniert. Das zur Zeit auf der Website verwendete Verfahren prognostiziert die Verspätung der einzelnen Züge in einer Verbindung und berechnet daraus den Verbindungsscore.

4.2 Verbindungsbewertung

Grundsätzlich gibt es zwei unterschiedliche maschinelle Lernverfahren, mit denen wir Verspätungen vorhersagen können:

1. Wir verwenden eine kontinuierliche Skala für die Verspätungen und nutzen zum Vorhersagen ein Regressionsmodell.
2. Wir definieren diskrete Klassen (z.B. nicht verspätet oder verspätet, oder mehr als 5 Minuten Verspätung, mehr als 10 Minuten Verspätung, etc.) und wir sagen diese mit einem Klassifikationsmodell vorher.

Für beide Ansätze des maschinellen Lernens benötigen wir Daten, aus denen wir die Verspätung vorhersagen wollen. Wichtig ist bei diesen Daten, dass sie für unsere Website auch in Echtzeit verfügbar sind. Das ist gar nicht so einfach, da wir unsere Daten von dem IRIS bekommen, zu Routing auf unserer Website aber HAFAS von marudor verwenden. Wir könnten die Linie (z.B. S3 für eine bestimmte S-Bahnlinie) zwar zum Trainieren verwenden, können aber dann später keine Vorhersagen damit machen, weil dieses Merkmal nicht über HAFAS verfügbar ist. Aus den vielfältigen Daten, welche wir gesammelt haben, haben wir letzten Endes folgende Merkmale ausgewählt:

- **Wann?** Wochentag, Uhrzeit
- **Wo?** Längen- & Breitengrad, Aufenthaltszeit des Zugs, Bahnsteig, Bahnhof, Streckenlänge seit Startbahnhof, Wievielter Halt seit Startbahnhof
- **Zug?** Zugnummer, Zuggattung, Betreiber

Diese Daten können aber so noch nicht für einen Machine Learner verwendet werden. Einige Merkmale, wie zum Beispiel die Betreiber-ID¹, die als Text vorliegt, müssen erst kodiert werden, bevor sie ein Machine Learner verstehen kann. Dafür gibt es zwei verschiedene Methoden: One-Hot oder Label Encoding. Bei One-Hot encoding wird für jedes unique value aus einer Spalte eine neue Spalte vom Typ bool erstellt, die die Werte aus der Originalspalte abbildet. Beim Label Encoding wird einfach jedem unique value ein bestimmtes Integer zugeordnet.

Ursprüngliche Spalte	DB Regio	Go-Ahead	FlixTrain
DB Regio	1	0	0
Go-Ahead	0	1	0
FlixTrain	0	0	1

Ursprüngliche Spalte	Label Encoding
DB Regio	0
Go-Ahead	1
FlixTrain	2

¹Der Betreiber ist quasi das Unternehmen, dass den Zug betreibt. Wie z.B. DB Regio AG, Go-Ahead oder FlixTrain.

Bis auf wenige Ausnahmen liefert One-Hot encoding bessere Ergebnisse, da Label Encoding ein Problem hat: Die zugeordneten Integer haben eigentlich keine Reihenfolge. Das *FlixTrain* > *DBRegio* ist, ergibt keinen Sinn. Baum-basierte Lerner machen aber bei einem bestimmten Wert einen Split, der dann z.B. Go-Ahead und FlixTrain zusammenwürft, obwohl diese eigentlich ganz anders sind.

Für uns kommt leider trotzdem nur Label Encoding in Frage, da wir sonst in unserem Datensatz über 10000 Spalten hätten, was nicht mehr in den Arbeitsspeicher passt und wir daher nicht trainieren können. Wir verwenden Dask Categoricals um die Features zu kodieren. Wie bei so vielen Dingen mit Dask ist das leider mal wieder komplizierter als erwünscht, da beim Speichern und erneutem Laden in das Apache Parquet² Format die einzelnen Kategorien vergessen werden und manuell wieder geladen werden müssen. Übrigens ohne, dass dies irgendwo in der Dokumentation erwähnt wird.

Fast alle Experimente, die wir bisher mit Regression für die Verbindungsbewertung gemachte haben, haben sehr schlechte Ergebnisse erzielt. Auf Regression umzusteigen ergibt unserer Meinung nach nur Sinn, wenn wir vor hätten, eine Echtzeitprognose, wie zum Beispiel einen Abfahrtsmonitor mit verbesserten Verspätungsangaben, zu entwickeln. Dabei gibt es aber einige Probleme, weshalb wir dies erst mal nicht machen werden: Sagen wir einem Zug eine höhere Verspätung vorher, als der Zug tatsächlich hat, so verpassen eventuell einige Fahrgästen den Zug weil, sie zu spät zum Bahnhof kommen. Solche Fehler könnte man eventuell verhindern, wenn man Positionsdaten der Züge hätte, aber heutzutage sind noch nicht einmal alle Loks mit GPS ausgestattet.

Deswegen sind wir schon letztes Jahr auf Klassifikation umgestiegen. Dabei haben wir für Ankunft uns Abfahrt jeweils diese 3 Verspätungsklassen verwendet:

- bis zu 5 Minuten Verspätung
- bis zu 10 Minuten Verspätung
- bis zu 15 Minuten Verspätung

Um die Ergebnisse zu verbessern, musste ein besseres Verfahren entwickelt werden.

Folgende Ideen sind uns dazu gekommen:

- Mehr Klassen
- Verbesserte Klassifikation
- Direkte Ermittlung des Verbindungsscores

4.2.1 Mehr Klassen

Um die Genauigkeit der Vorhersagen zu verbessern, haben wir jetzt statt jeweils 3 Verspätungsklassen für Ankunft und Abfahrt 40 solcher Klassen definiert. Für die Berechnung des Verbindungsscores haben wir die Klassen für die Ankunft als $delay \leq [0, 1, 2, \dots, 39]$ die für die Abfahrt als $delay \geq [0, 1, 2, \dots, 39]$ definiert. Dementsprechend ändert sich auch die Berechnung des Verbindungsscores eines einzelnen Umstiegs. Bisher hatten wir je nach Umsteigezeit hardencoded, wie die Klassen zur Berechnung kombiniert werden. Jetzt gibt es nur noch 2 Fälle:

Unsere Modelle liefern die in der folgenden Tabelle aufgelisteten Werte. Z.B. ar_4 ist die Wahrscheinlichkeit, dass der Zug maximal 4 Minuten verspätet ankommt und dp_1 ist die Wahrscheinlichkeit, dass der Zug mindestens eine Minute verspätet abfährt.

²https://en.wikipedia.org/wiki/Apache_Parquet

Umsteigezeit	Verbindungsscore
≤ 39	$= ar_{u-2} * dp_0 + \sum_{i=u-1}^{39} (ar_i - ar_{i-1}) * dp_{i-u+2}$
> 39	$= 1$

Tabelle 4.1: Verbindungsscore Klassen

Minuten Verspätung	0	1	2	3	4	5	...	39
Modell Ankunft	ar_0	ar_1	ar_2	ar_3	ar_4	ar_5	...	ar_{39}
Modell Abfahrt	dp_0	dp_1	dp_2	dp_3	dp_4	dp_5	...	dp_{39}

Grundsätzlich sollte gelten: $ar_0 \leq ar_1$ und $dp_0 \geq dp_1$

Im folgenden wird ein Umstieg betrachtet, bei dem es Fahrplanmäßig 3 Minuten Umsteigezeit gibt. Wir gehen davon aus, dass man mindestens 2 Minuten Umsteigezeit braucht, um den Anschluss zu erreichen. Hat der ankommende Zug in diesem Fall eine Minute Verspätung, so ist das egal, da der Anschluss trotzdem erreicht wird. Hat der Zubringerzug genau 3 Minuten Verspätung, muss der Abfahrende Zug mindestens 2 Minuten Verspätung haben, damit der Anschluss klappt.

ar_1	$ar_2 - ar_1$	$ar_3 - ar_2$	$ar_4 - ar_3$	$ar_5 - ar_4$...	$ar_{39} - ar_{38}$
dp_0	dp_1	dp_2	dp_3	dp_4	...	dp_{38}

Die eingetragenen Wahrscheinlichkeiten in der Tabelle werden je Spalte multipliziert und dann summiert. Mathematisch ausgedrückt sieht die Formel dann wie in der Tabelle 4.1 dargestellt aus, wenn u die planmäßige Umsteigezeit ist.

4.2.2 Verbesserte Klassifikation

Eine Zeit lang haben wir einen Random Forest Classifier verwendet. Dieser hat in einer ersten Analyse mit Weka³ und einer weiteren in Python deutlich besser abgeschnitten als Neuronale Netze, Decission Trees, Support Vector Machines und Extra Trees⁴. Auch in unserem Fall, stellte sich heraus, dass XGBoost (Chen and Guestrin, 2016) besser als unsere bisherigen Classifier abschnitt, in Tabelle 4.2 sieht man die Genauigkeit der Modelle auf einem ausgeglichenem Datensatz, also in dem Datensatz sind gleich viele verspätete wie pünktliche Züge.

Modell	Genauigkeit	Konfusionsmatrix ⁵
Majority Baseline	49.9303%	274507 0 275273 0
RandomForest	65.8962%	167097 107410 80086 195187
ExtraTrees	63.8821%	145447 129060 69509 205764
XGBoost	72.0632%	189436 85071 68520 206753

Tabelle 4.2: Vergleich ML Modelle

³<https://www.cs.waikato.ac.nz/ml/weka/>

⁴kurz für Extremely Randomized Trees Klassifier

⁵Eine Konfusionsmatrix auch Wahrheitsmatrix genannt ist eine Tabelle, die häufig verwendet wird, um die Leistung eines Klassifikationsmodells auf einem Satz von Testdaten zu beschreiben, für die die wahren Werte bekannt sind

Um die Klassifikation zu verbessern, haben wir uns nach besseren Classifiern umgeschaut. Auf AI Competition Seiten, wie kaggle.com, ist XGBoost seit einiger Zeit berühmt für die guten Resultate und gewinnt viele Wettbewerbe. XGBoost ist Teil der scikit-learn ([Pedregosa et al., 2011](#)) API und daher ein “drop-in-replacement” für uns. Jedenfalls theoretisch. Wie wir es bei dem Random Forest auch gemacht haben, müssen wir erst mit Hyperparametertuning das Modell an unsere Daten anpassen. Bei einem Random Forest gibt es 2 Parameter zu tunen. Und da es nur ein paar Sekunden dauert, einen einzelnen Classifier zu trainieren, kann das Hyperparametertuning problemlos in einer Nacht mit einer GridSearch gemacht werden. XGBoost hingegen hat einen viel größeren Parameterspace und braucht auch generell deutlich länger zum Trainieren. Wir verwenden daher Optuna ([Akiba et al., 2019](#)), um die besten Parameter für unsere Modelle zu finden. Optuna ist ein Blackboxoptimizer in Python, der besonders für Hyperparametertuning entwickelt wurde. Das Framework kann daher sowohl für XGBoost, als auch für Pytorch, Tensorflow oder Scikit-Learn verwendet werden. Wir lassen von Optuna die Parameter `N_ESTIMATORS`, `MAX_DEPTH`, `GAMMA` und `LEARNING_RATE` optimieren. Obwohl wir freundlicherweise Computer Ressourcen (NVIDIA V100) vom Max Planck Institut für Intelligente Systeme in Tübingen bekommen haben, braucht es durch Cross-Validation bis zu sechs Stunden, um ein einzelnes Modell zu tunen. Und wir haben ja insgesamt 80. Daher haben wir uns erst mal 14 Modelle herausgepickt, und bei diesen jeweils nur 50 anstelle von 100 Trials berechnet. Das ist eventuell insofern problematisch, weil der Tuner erst einige zufällige Versuche braucht, um dann tatsächlich gute Schätzungen zu machen. Wie in diesem GIF https://miro.medium.com/max/640/1*qDY1wa5b4hhaMCMY6YMQvg.gif sehr gut illustriert, erstellt der Tree-structured Parzen Estimator (TPE), der hinter Optuna steckt, eine Art Heatmap, die mit jedem Trial immer mehr Details bekommt. Wir können jetzt nicht sagen, ob die 50 Trials ausreichend waren, um die besten Ergebnisse zu liefern. Wir werden auf jeden Fall so bald wie möglich die Berechnungen mit mehr Trials wiederholen.

Die Ergebnisse sind wie erwartet pro Modell sehr unterschiedlich. Ein Parameter, die `MAX_DEPTH`, welche angibt, wie tief ein Baum wächst, ist einem klaren Trend gefolgt. Die Modelle die eine geringe Verspätung vorhersagen, wie z.B. $delay \leq 0$ funktionieren mit einer hohen `MAX_DEPTH` (14) am besten und Modelle, die hohe Verspätungen vorhersagen, kommen besser mit einer `MAX_DEPTH` von 2-3 klar.

Um zu verstehen, warum sich die `MAX_DEPTH` so ändert, muss man sich die Majority Baseline der verschiedenen Klassen anschauen. Diese besagt, wie viel Prozent der Datenpunkte zu der Klasse mit den meisten Datenpunkten gehören.

Modell	ar_0	ar_1	ar_2	ar_3	ar_4	...	ar_{38}	ar_{39}
Majority Baseline	52,71%	73,47%	82,80%	88,02%	91,24%	...	99,93%	99,94%

Nur 0,06% der Züge sind mehr als 39 Minuten verspätet. Dementsprechend gibt es in unserem Datensatz nur wenige Datenpunkte von mehr als 39 Minuten verspäteten Zügen. Ein Baum, mit einer zu hohen Tiefe, würde hier enorm schnell overfitten. Im Vergleich dazu wird das Modell ar_0 quasi auf einem balanced Datensatz trainiert. Da es für beide Klassen ($delay \leq 0$ und $delay > 0$) sehr viele Datenpunkte gibt, hilft eine hohe `MAX_DEPTH` um genauer zu klassifizieren.

4.2.3 Direkte Ermittlung des Verbindungsscores

Eine weitere Möglichkeit ist, dass wir gar keine Verspätungsklassen mehr vorhersagen sondern direkt den Verbindungsscore ermitteln. Dies hat einige Vorteile, wie zum Beispiel, dass es den Code deutlich simpler macht. Es muss nur noch ein Modell trainiert werden und wir brauchen auch nur noch ein Modell, um die Vorhersagen zu machen. Wir haben dafür einen Datensatz errechnet, indem Umstiege von 2 bis 10 Minuten enthalten sind. Als Features benutzen wir die Features des ankommenden und des abfahrenden Zuges plus die planmäßige Umsteigzeit. Bei

inem ersten Testlauf konnten wir damit den Verbindungsscore um gut 1% im Vergleich zur Majority Baseline von 88% Verbessern. Bei dem Feature Importance ist nun die Umsteigezeit das wichtigste Merkmal, aber dies war zu erwarten, da bei 10 Minuten Umsteigezeit wesentlich häufiger die Verbindung erreicht wird als bei nur 2 Minuten Umsteigezeit. Es war uns aber noch nicht möglich, hier ein Hyperparameter Tuning zu machen, da wir die benötigte Rechner-Kapazität zur Zeit nicht haben. Es ist aber insgesamt ein vielversprechendes Verfahren.

4.3 Feature Importance

Um zu evaluieren, wie gut ein Modell in seinen Vorhersagen ist, ist es auch wichtig sich anzuschauen, welche Rolle die einzelnen Features spielen. Wir haben daher den Information Gain für jedes Merkmals berechnet.

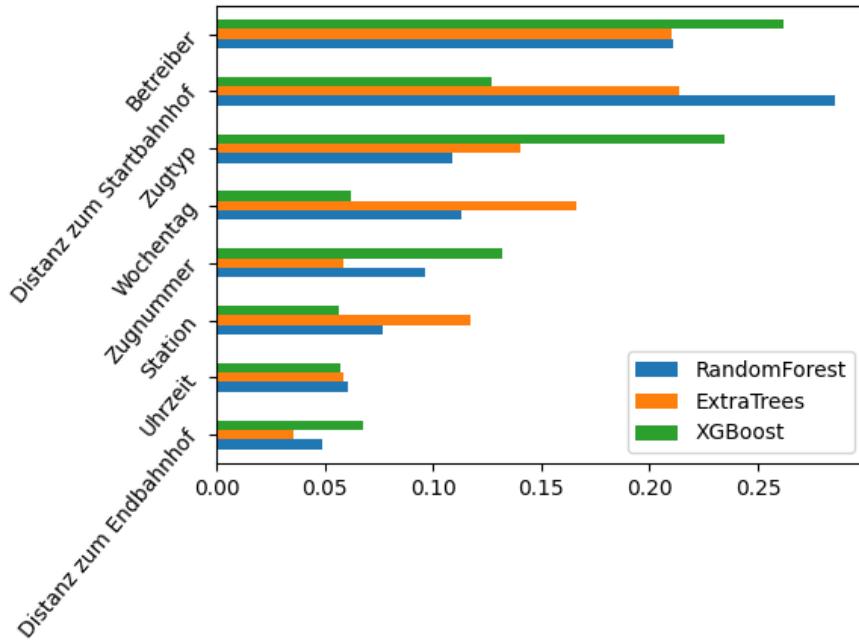


Abbildung 4.1: Feature Importance⁶

Die unsere unterschiedlichen Classifier sind sich hier uneinig. Der XGBoost-Classifier findet interessanter Weise den Betreiber am wichtigsten, was uns sehr überrascht hat. Der RandomForest hingegen bleibt, wie letztes Jahr, bei der “Distanz zum Startbahnhof” als wichtigstes Feature. Das ist auch nicht überraschend, da dieses Merkmal kumulativ viele Faktoren zusammenfasst, die wir einzeln gar nicht erfassen können, wie zum Beispiel Signalstörungen, Polizeieinsätze oder Personenschäden. Was positiv auffällt, ist, dass die gegenüber letztem Jahr neuen Features Betreiber, Zuggattung, und Zugnummer, für die Modelle informativ sind. Uns hat überrascht, dass die Angabe der Stunde am Tag, in der der Halt stattfindet, so unwichtig ist, obwohl die Verspätung über den Tag verteilt so klare Muster zeigt, wie in Abbildung 3.2 zu sehen ist. Vielleicht ist die Stunde aber auch einfach zu ungenau und es wäre besser stattdessen die Minuten seit 00:00 Uhr zu nehmen.

Ein weiteres Feature zeigt sich nicht in dieser Abbildung: Wir haben von DB Analytics gelernt, dass ein Großteil der Verspätungen von Baustellen verursacht werden. Wir haben aber keinerlei Daten dazu, wo und wann es Baustellen gibt. Was wir daher machen, ist, dass wir die Modelle immer nur auf die Daten der letzten zwei Wochen trainieren. Häufen sich im Sommer die Baustellen, so passen sich unsere Modelle automatisch auf die neue Situation an. Es hat außerdem den positiven Effekt, dass wir nicht jedes mal zum Trainieren eine V100 mit 32GB Grafikspeicher brauchen.

⁶Die hier gezeigten Modelle sind die gleiche wie in Tabelle 4.2

4.4 Klassifikationsgenauigkeit

Jedes Modell hat eine unterschiedliche Klassifikationsgenauigkeit. Da die Modelle für Ankunft und Abfahrt ungefähr gleich abschneiden, betrachten wir im folgenden nur die Modelle für die Ankunft.

Model	ar_0	ar_2	ar_5	ar_8	ar_{13}	ar_{20}	ar_{30}
Majority Baseline	53.71%	83.52%	93.56%	98.25%	98, 36%	99.13%	99.61%
Model Accuracy	78.81%	84.99%	93.57%	98.25%	98, 36%	99.13%	99.61%
Delta	25.10%	01.47%	00, 01%	00, 01%	00.00%	00.00%	00.00%

Die Ergebnisse für ar_0 sind wirklich gut. Hier können wir fast 80% der Züge richtig klassifizieren. Das ist eine riesige Verbesserung gegenüber der Majority Baseline. Dieses Modell alleine könnte verwendet werden, um besonders knappe Verbindungen zu optimieren. Die anderen Ergebnisse lassen leider etwas zu wünschen übrig. Dabei müsste man aber einige Modelle nochmal gesondert für bestimmte Zugtypen anschauen, da in den Daten die beiden sehr pünktlichen Zugtypen "S" und "RB" die Daten dominieren. Würde man hier ein extra Modell nur für den Fernverkehr trainieren, so könnte dieses bei mittleren Verspätungen deutlich besser Abschneiden.

Kapitel 5 Verbindungsbewertung

5.1 Vergleich und Evaluierung

Um zu schauen wie Aussagekräftig der Verbindungs-Score ist, haben wir eine Evaluierungsmethode entwickelt: Es werden alle Verbindungen im Datensatz gesucht, die planmäßig genau 5 Minuten Umsteigzeit haben, dargestellt im linken Plot "Planmäßige Umsteigzeit" in Abbildung 5.1. Im Plot "Tatsächliche Umsteigzeit" wird die Verspätung der Züge mit eingerechnet, sodass die tatsächliche Umsteigzeit herauskommt. Eine Umsteigzeit unter 0 Minuten bedeutet, dass der abfahrende Zug schon weg war, bevor der ankommende am Bahnhof eingetroffen ist.

Die rote Linie ist eine lineare Regression, durch alle Punkte auf dem Graphen. Da die Ableitung der Regression positiv ist, sind die Vorhersagen auf jeden Fall besser als der Durchschnittswert. Im Bereich, wo der Verbindungsscore um 0,6 ist, werden funktionieren weniger Umstiege als im Bereich $Verbindungsscore > 0,9$, obwohl dort insgesamt die meisten Verbindungen sind.

Kapitel 6 Webseite

6.1 Warum eine Webseite?

Wenn etwas entwickelt wird und dann nur in einer Schublade verschwindet, dann hat es niemandem was gebracht. Um das Projekt öffentlich zu machen, bietet sich eine für jeden zugängliche Webseite oder eine App an. Letzteres hat den Nachteil, dass man nicht für alle Plattformen den gleichen Code verwenden kann. Das macht es aufwändiger, vor allem wenn man nicht nur mobile Betriebssysteme unterstützen möchte. Eine Webseite hingegen funktioniert grundsätzlich auf allen Geräten, solange diese Internet haben, und ist bei Bedarf auch einfach durch eine Progressive Web App in die Appstores zu bringen (Der Bing Crawler bringt PWAs sogar automatisch in den Microsoft Store).

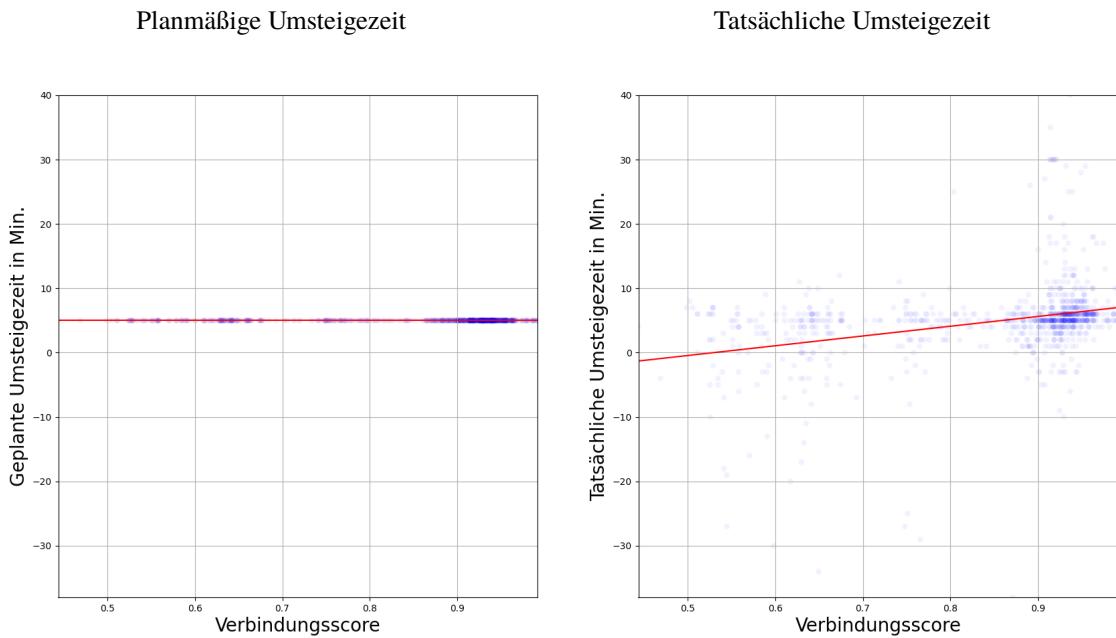


Abbildung 5.1: Bezug von Verbindungs-Score zu Umsteigezeit

6.2 Backend

Da wir unsere gesamte Codebase in Python entwickelt haben, sowie unsere Maschine Learning Modelle, war es am sinnvollsten, auch das Backend mit Hilfe von Python zu erstellen. Wir haben uns für Flask und gegen Django als Web-Framework entschieden, weil Flask etwas leichter ist und eine flachere Learning Curve hat. Als Production Server verwenden wir gunicorn. Unser Webserver ist in einem Docker Image, welches bei einem Push zum Master-Branch auf GitHub, automatisch auf unseren privaten Server deployed wird, wodurch unser Webserver immer aktuell bleibt.

Das Backend hat drei Zugangspunkte¹:

- <http://server.url/api/connect>
Wird beim laden der Webseite aufgerufen um, die Bahnhofsvorschläge zu bekommen.
- <http://server.url/api/trip>
Sucht die Verbindungen und gibt dann die bewerteten Verbindungen zurück.
- <http://server.url/api/deploy> wird von GitHub zum deployen genutzt.

6.3 Frontend

Das Ziel der Website ist es, eine benutzerfreundliche Oberfläche zu bieten, mit der es möglich ist eine Verbindung zu suchen, und dann graphisch zu sehen, wie gut diese ist. In den drei Eingabefeldern werden, ähnlich wie bei der Deutschen Bahn, Start- und Zielbahnhof sowie Datum und Uhrzeit einer gewünschten Verbindung eingegeben. Bei der Bahnhofseingabe werden alle verfügbare Bahnhöfe als Vorschläge angezeigt, die Datumseingabe kann über eine Schaltfläche gemacht werden. Die möglichen Verbindungen werden farblich gekennzeichnet angezeigt. Wird eine der Verbindungen angeklickt, werden weitere Details dazu angezeigt.

¹<http://server.url> wird hier als Beispiel URL genutzt, da sich die URL der Webseite manchmal ändert. <https://trainconnectionprediction.github.io/> leitet aber immer auf unsere Webseite um.

6.4 Kommunikation Frontend-Backend

Eine Hürde war die Kommunikation zwischen Frontend und Backend. Nachdem das Formular mit der zu suchenden Verbindung abgeschickt wird dauert es ca. 10 Sekunden, bis das Backend alle Berechnungen fertig hat. Um den Benutzer nicht warten zu lassen, sollte eine dynamische Webseite entstehen, welche nicht immer neu lädt, sondern direkt mit dem Backend kommuniziert und sich automatisch mit den Vorhersagen verändert.

Anfangs wurde dies mit Hilfe von SocketIO gelöst, doch diese Verbindung stellte sich als sehr anfällig für schlechte Internetverbindungen heraus, und es gab Probleme beim Cachen von Ergebnissen für unterschiedliche gleichzeitige Nutzer. Daher sind wir auf eine REST²-API umgestiegen, welche vergleichsweise einfach zu implementieren war.

Um ein Verbindung zu suchen, senden wir mit Hilfe von Ajax³ eine Post-Request an <http://server.url/api/trip> mit den Eingaben des Users als Parameter Server. Der Server antwortet die Anfrage mit einem JSON-String, welcher ein Array mit jeder berechneten Verbindung beinhaltet. Dieser JSON-String wird danach graphisch für den Benutzer angezeigt.

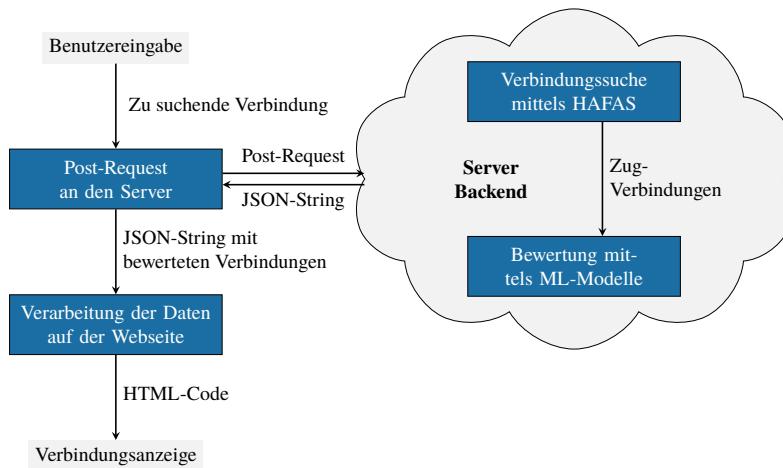


Abbildung 6.1: Funktionsweise der Webseite

Kapitel 7 Zusammenfassung

In 2 Jahren Arbeit ist ein KI-basiertes System und eine Webseite entstanden, über die jede*r die beste Verbindung für sich suchen kann. Dafür wurden Daten von verschiedenen Quellen zusammen gecrawled, sodass wir einen Datensatz bekommen haben, der nicht mal mehr in 32GB RAM passt. Dass hat für einige Schwierigkeiten gesorgt. Auf diesem Datensatz wurde dann eine Analyse gemacht, um prädiktive Features für eine Künstliche Intelligenz zu finden. Dafür wurden verschiedene Machine Learning Algorithmen getestet, um den besten davon herauszufinden. Die Prognosen des Algorithmus können nun auf einer für alle Bahnfahrer*innen verwendbaren Website angezeigt werden.

Und es geht weiter: Wir haben fest geplant, unsere eigene Routing-Engine zu bauen, um in Zukunft noch bessere Verbindungen zu finden, sobald wir genügend GTFS-Daten dafür bekommen.

²Representational State Transfer

³Kurz für “Asynchrones JavaScript + XML”

Unterstützungsleistungen

Vielen Dank an alle, die uns bei unserer Arbeit unterstützt haben.

- **Uni Tübingen / MPI für Intelligente Systeme / BWKI:** Bereitstellung von Rechenleistung mittels CPUs und GPUs.
- **DB Analytics:** Einblicke in den Bahnbetrieb sowie Antworten auf bahninterne Fragen. Außerdem gewinnt unsere Kontaktperson Jochen Völker unseren Low Latency Award ($\overline{\text{latency}} \leq 10\text{min}$) beim Beantworten von E-Mails.
- **marudor:** Marudor hat auf seiner Website marudor.de eine API, mit der man einfach auf die HAFAS API zugreifen kann. Diese ist gut dokumentiert, und hat uns z.B. das Routing deutlich erleichtert.
- **OpenStreetMaps:** Ohne sehr viele Menschen, die eine öffentliche und extrem genaue Karte des Deutschen Schienennetzes erstellt haben und immer erneuern und verbessern, hätten wir vermutlich immer noch kein routingfähiges Streckennetz.
- **SFZ Eningen:** Unsere Server stehen zur Zeit im SFZ Eningen, nachdem wir aus dem Schulnetzwerk rausgeworfen wurden. Vielen Dank an den Standortleiter Herrn Groß, der auch beim Transport der Server nach Eningen geholfen hat.



Vielen Dank an Herrn Glück, der uns bei der Anschaffung unseres eigenen “Supercomputers” sehr unterstützt hat. Finanzielle Unterstützung dafür erhielten wir von:

- Bosch
- Herrn Mörike
- Kepi-Föderkreis
- Micron

Literaturverzeichnis

- Akiba, Takuya, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama,** “Optuna: A Next-generation Hyperparameter Optimization Framework,” 2019. in KDD.
- Boeing, Geoff,** “OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks,” *Computers, Environment and Urban Systems*, 2017, 65, 126 – 139.
- Chen, Tianqi and Carlos Guestrin,** “XGBoost: A Scalable Tree Boosting System,” in “Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining” KDD ’16 ACM New York, NY, USA 2016, pp. 785–794.
- Dask Development Team,** *Dask: Library for dynamic task scheduling* 2020.
- Gerbig, Hjalmar,** “Bildnachweis: Foto “ICE in der Morgensonne” auf der Titelseite verwendet mit freundlicher Genehmigung des Autors.” www.jally.de/sg3q2.
- Kriesel, David,** “36C3 - BahnMining - Pünktlichkeit ist eine Zier,” 2020. <https://youtu.be/0rb9CfOvojk>.
- pandas development team, The,** “pandas-dev/pandas: Pandas,” February 2020.
- Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay.** “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, 2011, 12, 2825–2830.