Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

# Realisierung von umfassenden Analysetechniken in einer hybriden Datenverarbeitungsarchitektur

Simone Schmidt

| | |
|---|---|
| **Studiengang:** | Medieninformatik |
| **Prüfer/in:** | Prof. Dr.-Ing. Bernhard Mitschang |
| **Betreuer/in:** | Corinna Giebler, M. Sc. Informatik,<br>Dr. rer. nat. Christoph Stach |
| **Beginn am:** | 7. Mai 2018 |
| **Beendet am:** | 14. November 2018 |

# Kurzfassung

Es gibt heutzutage viele Bereiche, in denen große Mengen an Daten anfallen, wie zum Beispiel in der *Industrie 4.0*, bei *eHealth* und bei Überwachung und Regelung des *öffentlichen Personennahverkehrs (ÖPNVs)*. Um möglichst viele vorteilhafte Informationen aus den Daten zu gewinnen, werden umfassende Analysen benötigt, die nicht nur historische, sondern auch Echtzeitdaten berücksichtigen und die Analyseergebnisse in Echtzeit anwenden können. Es gibt hybride Architekturen, welche die Analyse beider Arten von Daten durch die Nutzung von Stream- und Batchverarbeitung unterstützen. Eine solche Architektur ist *Hybrid Processing Architecture for Big Data (BRAID)*, wobei *BRAID* zusätzlich die Zusammenarbeit zwischen Batch und Stream ermöglicht. Diese Arbeit untersucht, inwiefern *BRAID* für die Umsetzung solcher umfassender Analysen geeignet ist. Hierfür wird ein Anwendungsfall aus dem Bereich des ÖPNV entwickelt, welcher umfassende Analysen benötigt, und es werden Anforderungen abgeleitet, welche ein System erfüllen muss, um dem Anwendungsfall gerecht zu werden. Beispiele aus der Literatur werden untersucht. Dabei zeigt sich, dass die Anforderungen von bestehenden Systemen noch nicht voll erfüllt werden können. Unter Nutzung der Architektur *BRAID* wird ein System entwickelt, welches die Anforderungen erfüllt. Es werden verschiedene Machine Learning (ML)-Verfahren und Frameworks, welche für solch ein System genutzt werden können, diskutiert, untereinander verglichen und evaluiert. Das geeignetste wird jeweils für die Umsetzung ausgewählt und das System wird prototypisch implementiert. Das entwickelte System wird gegen die Anforderungen evaluiert, wobei sich zeigt, dass das System alle Anforderungen erfüllen kann. Insgesamt zeigt sich hierdurch, dass *BRAID* zur Umsetzung eines Systems für umfassende Analysen geeignet ist.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Verzeichnis der Listings

# Abkürzungsverzeichnis

**BRAID** Hybrid Processing Architecture for Big Data. 3, 15, 30, 46

**HDFS** Hadoop Distributed File System. 34, 50

**ML** Machine Learning. 3, 15, 25, 29, 47

**ÖPNV** öffentlicher Personennahverkehr. 3, 15, 49

**PMML** Predictive Model Markup Language. 50

**VFDT** Very Fast Decision Tree. 51

**VVS** Verkehrs- und Tarifverbund Stuttgart. 7, 19, 49

# 1 Einleitung

Heutzutage fallen in vielen Bereichen große Mengen an Daten an. Solche Bereiche sind zum Beispiel die *Industrie 4.0* [Gr16], *eHealth* [Sta+18] und die Überwachung und Regelung des *öffentlichen Personennahverkehrs (ÖPNVs)* [Rag+16a]. In diesen Bereichen fallen unter Anderem Echtzeitdaten an. In der *Industrie 4.0* sind das beispielsweise Sensordaten von Sensoren, die die Produktion überwachen, bei *eHealth* beispielsweise in regelmäßigen Abständen gemessene Gesundheitsdaten, wie Blutdruck und Puls, und aus dem *ÖPNV* beispielsweise die reellen Abfahrts- und Ankunftszeiten der Fahrzeuge an den Haltestellen. Diese Echtzeitdaten werden in der Regel zur späteren Verwendung gespeichert, wodurch historische Datensätze entstehen. Um aus diesen großen Datenmengen, an sowohl historischen als auch Echtzeitdaten, möglichst große Vorteile zu ziehen, müssen darauf umfassende Analysen durchgeführt werden. Der Begriff umfassende Analysen beschreibt Analysen, die nicht nur historische, sondern auch Echtzeitdaten berücksichtigen, und die Analyseergebnisse wiederum in Echtzeit anwenden können.

Analysen von historischen und Echtzeitdaten allein können jeweils schon hilfreiche Erkenntnisse bieten. Es gibt jedoch einige Fälle, in denen Wissen, welches aus historischen Daten erlernt wurde, auf Echtzeitdaten angewendet werden soll. Es kann beispielsweise vorkommen, dass ein Nutzer aufgrund einer Klassifizierung von aktuellen Daten Entscheidungen treffen möchte, die sein momentanes Handeln beeinflussen. Hierzu muss zunächst ein Klassifizierungsmodell existieren, welches auf Echtzeitdaten angewandt wird. Dieses Modell muss zuvor auf historischen Daten erlernt worden sein. Durch das Phänomen des Konzeptdrifts [Tsy04] können erlernte Modelle ungenau werden, wodurch Echtzeitdaten falsch klassifiziert werden, was wiederum zu falschen Handlungen des Nutzers führen kann. Deshalb ist es wünschenswert, dass dieser Fall erkannt wird und das Modell aktualisiert werden kann. Um dies zu erreichen, müssen Ergebnisse von Verarbeitungen historischer und Echtzeitdaten sich gegenseitig und zusätzlich die zukünftige Verarbeitung beeinflussen können.

Um sowohl historische als auch Echtzeitdaten optimal verarbeiten zu können, werden *Batch Processing* (dt. Stapelverarbeitung), für die effiziente Verarbeitung historischer Daten, sowie *Stream Processing* (dt. Datenstromverarbeitung), für die schnelle und direkte Verarbeitung von Echtzeitdaten, benötigt [CY15]. Es gibt verschiedene Architekturen, die beides ermöglichen. Eine solche Architektur, die zusätzlich gegenseitige Beeinflussung ermöglicht, ist *Hybrid Processing Architecture for Big Data (BRAID)* [Gie+18].

## 1.1 Aufgabenstellung

In dieser Arbeit soll untersucht werden, inwiefern *BRAID* für die Umsetzung von umfassenden Analysen geeignet ist.

Hierfür wird zunächst ein konkreter Anwendungsfall entworfen und Anforderungen, die sich aus diesem ergeben, werden abgeleitet. Daraufhin wird am Beispiel des zuvor definierten Anwendungsfalls ein System entworfen, welches die Architektur *BRAID* nutzt. Hierfür werden existierende Systeme und Technologien evaluiert, welche für die Umsetzung verwendet werden können und für die Implementierung geeignete ausgewählt. Das System soll Machine Learning (ML) Modelle auf historischen Daten erlernen und diese auf Echtzeitdaten anwenden können. Gleichzeitig sollen Echtzeitdaten zur Verbesserung des Modells genutzt werden können. Dies bedeutet, dass das System zum einen ein bestehendes Modell anhand von Echtzeitdaten adaptieren können soll. Zum anderen soll ein neues Modell berechnet werden, wenn das bestehende zu ungenau ist. Dieses System wird prototypisch implementiert. Abschließend wird evaluiert, inwiefern das umgesetzte System die Anforderungen erfüllt, welche sich aus dem Anwendungsfall ergeben.

## 1.2 Gliederung

Die Arbeit ist folgendermaßen aufgebaut:

**Kapitel 2 - Anwendungsszenario** Umfassende Analysen werden in vielen Bereichen benötigt. Als Diskussionsgrundlage wird in diesem Kapitel deshalb ein beispielhaftes Szenario aus dem Bereich ÖPNV vorgestellt, für welches umfassende Analysen benötigt werden. Dieses Szenario befasst sich mit der Verspätungsvorhersage in öffentlichen Verkehrsnetzen. Es werden Use-Cases für dieses Szenario erstellt. Zudem ergeben sich aus den Use-Cases gewisse Anforderungen, die ein System erfüllen sollte. Diese werden aus den Use-Cases abgeleitet und vorgestellt.

**Kapitel 3 - Verwandte Arbeiten** Dieses Kapitel befasst sich mit einigen Arbeiten, welche Analysen auf Verkehrsdaten durchgeführt haben. Es wird behandelt, inwiefern sie für das beschriebene Szenario relevant sind und evaluiert, wie weit die zuvor aufgestellten Anforderungen bereits erfüllt werden.

**Kapitel 4 - Grundlagen** Einige für die Umsetzung des Systems relevante Techniken, Konzepte und Systeme werden in diesem Kapitel vorgestellt. Hierzu gehören die Konzepte des *Batch* und *Stream Processing*, Architekturen, die gleichzeitig im Batch und im Stream Verarbeitungen durchführen können, verschiedene Systeme, die sowohl Batch-, als auch Streamverarbeitung ermöglichen, sowie geeignete ML-Techniken. Die Architekturen, Systeme und Techniken werden untereinander verglichen und für das Anwendungsszenario passende für die Implementierung ausgewählt.

**Kapitel 5 - Konzept** In diesem Kapitel wird ein Konzept für die Umsetzung des Vorhersagesystems entwickelt. Hierbei werden die Daten erläutert, welche vom System verarbeitet werden sollen. Es wird die Architektur des Systems vorgestellt und erklärt, wie die Komponenten des Systems arbeiten und interagieren. Zusätzlich wird erläutert, welche Art von ML-Algorithmen am besten für das System geeignet sind.

**Kapitel 6 - Implementierung** Dieses Kapitel erläutert Möglichkeiten für die konkrete Umsetzung des beschriebenen Systems. Es wird eine mögliche Datenquelle betrachtet sowie Empfehlungen für die Umsetzung der Architektur getroffen. Hierbei werden unter anderem Empfehlungen bezüglich der Technologien für die Umsetzung der einzelnen Komponenten getroffen und ein konkreter ML-Algorithmus für das System vorgestellt.

**Kapitel 7 - Prototyp** Das zuvor vorgestellte System wird in leicht vereinfachter Version prototypisch implementiert. Der Prototyp wird in diesem Kapitel beschrieben. Hierbei werden die verwendeten Daten, die umgesetzte Architektur mit allen verwendeten Technologien und die Auswahl des ML-Algorithmus erläutert.

**Kapitel 8 - Evaluation** In diesem Kapitel wird bewertet, inwiefern das entwickelte System die Anforderungen, welche in Kapitel 2 gestellt werden, erfüllt. Hierfür werden sowohl das System, wie es in Kapitel 5 und Kapitel 6 vorgestellt wird, als auch der implementierte Prototyp, welcher in Kapitel 7 beschrieben wird, gegen die einzelnen Anforderungen evaluiert.

**Kapitel 9 - Zusammenfassung und Ausblick** Abschließend bietet dieses Kapitel eine Zusammenfassung der Arbeit sowie einen Ausblick auf mögliche zukünftige Arbeiten.

# 2 Anwendunsszenario

Die Ergebnisse dieser Arbeit können in verschiedensten Anwendungsfällen verwendet werden, in denen umfassende Analysen benötigt werden. Ein solcher Bereich ist der ÖPNV. Analysen werden hier für viele Zwecke gebraucht, darunter die Identifikation von Verspätungsursachen [Rag+16a], Untersuchungen des Bedarfs an öffentlichen Verkehrsmitteln [Bal+04] oder die Langzeitplanung in öffentlichen Verkehrsnetzen [Rag+16a]. Auch für die Vorhersage von Verspätungen werden Analysen benötigt. Dieses Szenario soll beispielhaft als Grundlage für die Diskussion und Implementierung dienen.

Viele Menschen sind täglich auf öffentliche Verkehrsmittel angewiesen, um von A nach B zu kommen. Allerdings kommt es leider häufig zu mehr oder minder großen Verspätungen bei einzelnen Fahrten oder in ganzen Abschnitten von Verkehrsnetzen [Pet+16; Stu18]. Um sich bei wichtigen Terminen nicht zu verspäten, kann es also von Vorteil sein, wenn bei der Planung einer Fahrt mit öffentlichen Verkehrsmitteln die zu erwartende Verspätung mit einbeziehen werden kann, vor allem, wenn bei dieser Verbindung ein Anschlusszug erreicht werden muss. Sich hierfür auf eigene Erfahrungen zu verlassen, wie es viele Reisende heutzutage tun, ist allerdings unzureichend. Personen von außerhalb, die zum ersten Mal oder nur selten ein bestimmtes Verkehrsnetz nutzen, haben diese Erfahrungen meist nicht. Zudem können diese Kenntnisse veraltet sein, wenn sich beispielsweise das Netz ändert oder durch Dauerbaustellen Engstellen im Netz entstehen. Um eine Reise rechtzeitig im Voraus planen zu können, brauchen Fahrgäste also eine Möglichkeit, mehrere Stunden oder gar Tage im Voraus eine Vorhersage für die Verspätung bei einer gewünschten Fahrt zu bekommen.

Es ist also sinnvoll, Verspätungsvorhersagen für den laufenden Betrieb sowie frühzeitige Vorhersagen für die Verbesserung der Reiseplanung zu ermöglichen.

## 2.1 Konkrete Fragestellung

In diesem Zusammenhang ist in Anlehnung an eine relevante Fragestellung aus dem ÖPNV

*Wie viel Verspätung habe ich als Reisender beginnend bei einer bestimmten Haltestelle bis hin zu einer bestimmten anderen Haltestelle zu erwarten?*

das folgende Anwendungsszenario entstanden. Ziel ist es, die Verspätungen von einer bestimmten Starthaltestelle zu mehreren Zielhaltestellen vorherzusagen. Kann man dies für eine bestimmte Starthaltestelle in einem bestimmten Verkehrsnetz umsetzen, so ist es auch in beliebigen Verkehrsnetzen und für beliebige Starthaltestellen in diesen Netzen möglich. Kann man diese Frage also für eine Haltestelle beantworten, so kann man sie für beliebige Haltestellen beantworten.

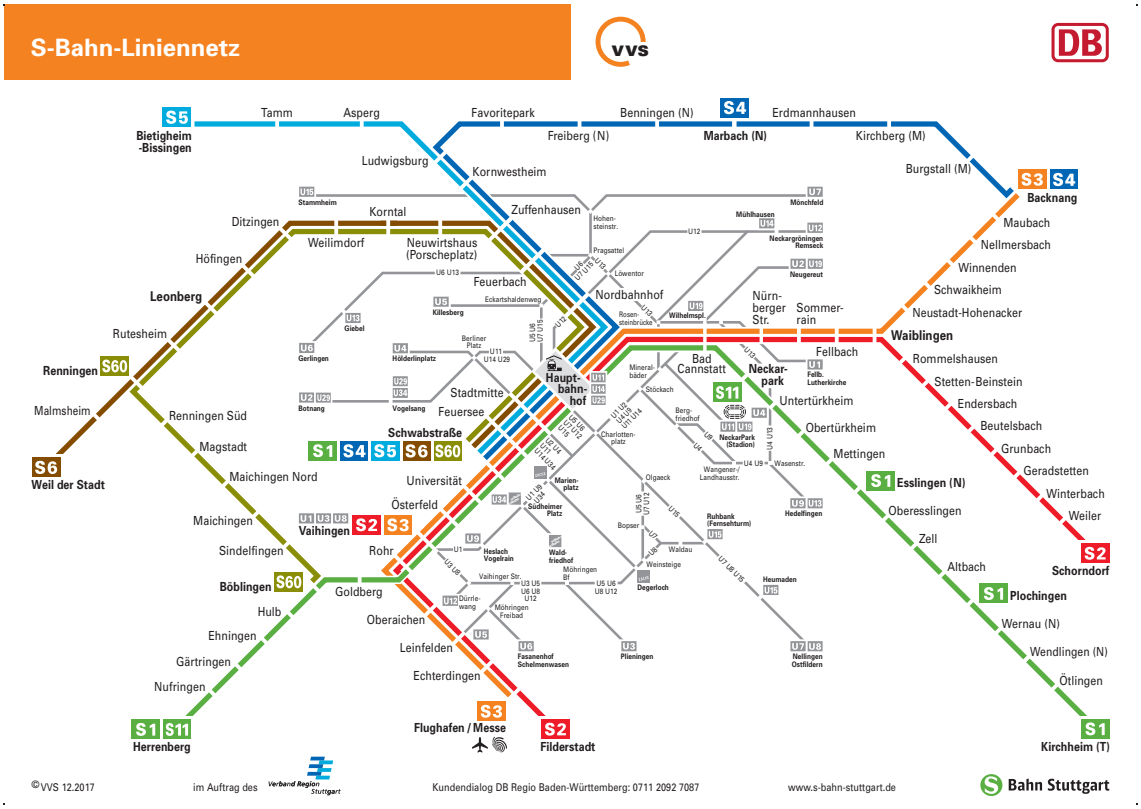



**Abbildung 2.1:** Das S-Bahnnetz im VVS [Ver18a].

Zur Veranschaulichung und Vereinfachung wird hier ein konkreter Fall aus dem Verkehrs- und Tarifverbund Stuttgart (VVS)-Netz (siehe Abbildung 2.1) betrachtet: In diesem Szenario werden Verspätungen betrachtet, die Reisende, beginnend vom Flughafen, bei einer Fahrt zu einigen wichtigen Umsteige- und Zielhaltestellen, zu erwarten haben. Der Flughafen wurde hierbei als Start gewählt, da dort einige interessante Startpunkte, wie beispielsweise die Messe Stuttgart, liegen und es gleichzeitig für ankommende Flugreisende der Umstiegspunkt für die Weiterfahrt im VVS-Netz ist. Als Ziele wurden die Haltestellen Universität (als häufiges Ziel), Schwabstraße (als Anbindungsstelle zur Weiterfahrt mit anderen S-Bahn-Linien), Stadtmitte (als häufiges Ziel), Hauptbahnhof (als Anbindungsstelle zur Weiterfahrt mit anderen S-Bahn-Linien oder anderen Verkehrsmitteln sowie als häufiges Ziel) und Bad Cannstatt (als häufiges Ziel) gewählt.

Für dieses vereinfachte S-Bahn-Netz (siehe Abbildung 2.2) sollen Vorhersagen getroffen werden, bezüglich der Verspätung, beginnend bei der Starthaltestelle bis zu einer der Zielhaltestellen.

## 2.2 Use-Cases

ÖPNV-Netze haben für in der Regel zwei Arten von Daten. Einerseits eher statische Datensätze, die sich nur selten ändern, welche das Netz, dessen Haltestellen, Linienverläufe, geplante Abfahrtszeiten und ähnliches beschreiben. Andererseits Echtzeitdaten, die die reale Situation im Netz in Form von

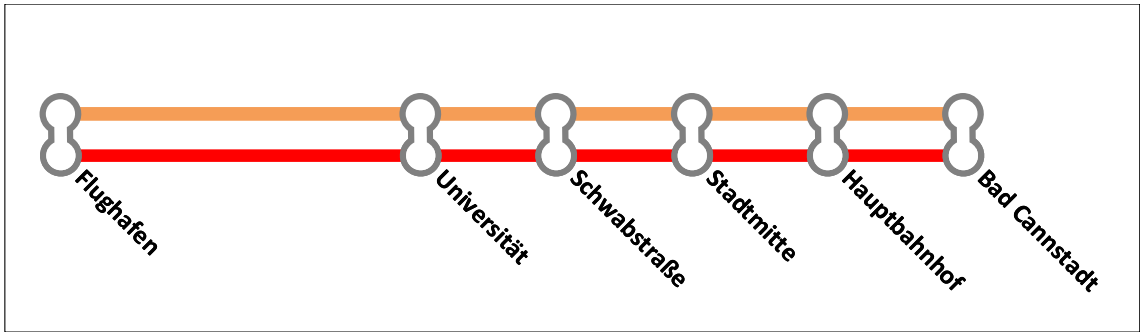



**Abbildung 2.2:** Das vereinfachte S-Bahnnetz mit sechs Haltestellen und zwei S-Bahn-Linien (S2 in rot und S3 in orange).

tatsächlichen Ankunftszeiten, Störungsmeldungen, aktuellen Fahrzeugpositionen und Ähnlichem beschreiben. Diese können auch gesammelt als historische Datensätze vorliegen [Rag+16a; Ver14; Ver18e].

Mit diesen Daten können nun verschiedene Zwecke erfüllt werden. Diese sind in den folgenden Use-Cases beschrieben:

UC1 Ein Nutzer, der in einigen Tagen mit dem Flugzeug in Stuttgart landet, will seine Weiterreise planen. Damit er weiß, welchen Anschlusszug er buchen muss, möchte er wissen, wie viel Verspätung er bis zum Hauptbahnhof zu erwarten hat.

UC2 Am Flughafen angekommen, bekommt der Nutzer mit, dass es eine Weichenstörung gab. Er möchte nun schnell wissen, wie sich die Verspätung dadurch ändert. So weiß er, ob er seinen Anschlusszug noch rechtzeitig erreicht und kann im Notfall umbuchen.

UC3 Ein Nutzer möchte vom Flughafen zu einer Veranstaltung an der Universität fahren. Er muss dort pünktlich ankommen und hat mitbekommen, dass es an diesem Tag stark stürmen soll. Nun rechnet er mit wetterbedingten Störungen und möchte wissen, wie sich dies auf die Verspätungen auswirkt.

UC4 Die Bauarbeiten für Stuttgart 21 sind im Bereich zwischen Flughafen und Bad Cannstatt endgültig abgeschlossen, wodurch die S-Bahnen in diesem Abschnitt allgemein viel pünktlicher fahren. Ein Nutzer möchte wissen, wie sich dies auf die Verspätungen auf seiner wöchentlichen Fahrt vom Flughafen zur Schwabstraße auswirkt.

## 2.3 Anforderungen an das System

Aus den soeben dargelegten Anwendungsfällen lassen sich einige Anforderungen ableiten, die ein System erfüllen muss, auf dem entsprechende Analysen durchgeführt werden sollen.

A1 (Aufgrund von UC1, UC2, UC3, UC4) Es soll ein möglichst genaues Vorhersagemodell für Verspätungen auf historischen Verspätungsdaten erlernt werden.

A2 (Aufgrund von UC1, UC2) Wenn Nutzer eine Reise planen, wollen sie nach Eingabe der erforderlichen Daten ein möglichst aktuelles Ergebnis. Wenn ein Nutzer für eine Verbindung eine Verspätungsvorhersage fordert, muss dies verarbeitet und die Klassifizierung anhand des erlernten Modells durchgeführt werden. Für die Klassifizierung sollen zudem Echtzeitdaten über den momentanen Zustand im Netz berücksichtigt werden. Unter anderem soll bei solchen Vorhersagen davon ausgegangen werden, dass der Streckenzustand den aktuellen Störungsmeldungen entspricht.

A3 (Aufgrund von UC3) Bei solch einer Vorhersage einer Verspätung soll auch von einem vom Nutzer selbst definierten Streckenzustand (zum Beispiel Unwetter, verschiedene Störungen oder Baustellen) ausgegangen werden können.

A4 (Aufgrund von UC4) Wenn sich die Gegebenheiten im Netz abrupt stark ändern, können sich dadurch auch die Verspätungen schnell ändern. Die Vorhersagequalität des Modells verschlechtert sich in solch einem Fall innerhalb kurzer Zeit stark. Ist dies der Fall, so soll dies erkannt und das Vorhersagemodell, anhand der nun relevanten neueren Daten, neu berechnet werden. Es soll also anhand der eingehenden Verspätungsdaten in Echtzeit die Qualität des Klassifizierungsmodells evaluiert werden. Wird das Modell zu ungenau, so wird veranlasst, dass das Modell neu berechnet wird.

A5 (Aufgrund von UC1, UC2, UC3, UC4) Zudem können sich die Gegebenheiten im Netz, und damit auch die auftretenden Verspätungen, langsam ändern, wodurch sich die Qualität des Modells langsam verschlechtern kann. Um trotzdem jederzeit möglichst genaue Vorhersagen treffen zu können, muss das Modell an die schwachen Veränderungen angepasst werden, sobald sie auftreten. Um dies zu ermöglichen, soll das bestehende Vorhersagemodell anhand der eintreffenden Verspätungsdaten in Echtzeit angepasst werden.

A6 (Aufgrund von UC2) Der momentane Streckenzustand im System soll anhand der eintreffenden Störungsmeldungen angepasst werden.

Um diese Anforderungen zu erfüllen, muss das System also sowohl die Verarbeitung von historischen als auch von Echtzeitdaten ermöglichen. Die Verarbeitung von historischen Daten wird benötigt, um ein möglichst genaues Vorhersagemodell zu erlangen. Die Verarbeitung von Echtzeitdaten hingegen wird für alles gebraucht, das in Echtzeit geschehen soll, also zur Evaluation des Modells, zur Klassifizierung der Anfragen und zur Verarbeitung der eintreffenden Störungsmeldungen. Zudem müssen die Ergebnisse aus der Verarbeitung der historischen Daten für die Verarbeitung von Echtzeitdaten verwendet werden können, da beispielsweise die Qualität des erlernten Modells anhand der Echtzeitdaten evaluiert werden muss. Gleichzeitig muss es möglich sein, dass die Ergebnisse der Verarbeitung von Echtzeitdaten die Verarbeitung der historischen Daten beeinflussen, da wenn die ermittelte Qualität des Modells zu gering wird, das Modell nur unter Verwendung neuerer historischer Daten neu berechnet werden soll. Somit müssen sich die beiden Verarbeitungsarten im System gegenseitig beeinflussen können.

Im Folgenden wird zunächst in Kapitel 3 betrachtet, wie in anderen Arbeiten Analysen auf Mobilitätsdaten durchgeführt wurden. Anhand dessen wird evaluiert, inwiefern andere Systeme die aufgestellten Anforderungen bereits erfüllen und welche Funktionalitäten fehlen, um sie voll zu erfüllen.

Die Nutzung einer hybriden Verarbeitungsarchitektur ist bei den gegebenen Anforderungen an das System von Vorteil. In Abschnitt 4.2 werden verschiedene hybride Verarbeitungsarchitekturen vorgestellt. Sie ermöglichen hierbei, dass parallel effizient auf großen Mengen historischer Daten gearbeitet wird, sowie eintreffende Daten in Echtzeit verarbeitet werden. Somit könnte effizient ein Vorhersagemodell für Verspätungen erlernt werden und gleichzeitig könnten sich beispielsweise Nutzer schnell Verspätungen vorhersagen lassen.

# 3 Verwandte Arbeiten

Analysen von Verkehrsdaten wurden in der Literatur bereits häufig vorgenommen. Viele davon haben sich bereits mit Verspätungen in öffentlichen Verkehrsnetzen beschäftigt. Im Folgenden wird bewertet, inwiefern bisherige Arbeiten die gestellten Anforderungen aus Abschnitt 2.3 bereits erfüllen. Die Ergebnisse sind zusammenfassend in Tabelle 3.1 zu sehen.

Berlingerio et al. [Ber+15] haben sich beispielsweise mit einem System für Routenplanung in Rom beschäftigt. Sie haben ein System erstellt, welches Verbindungsvorschläge für das multimodale Verkehrsnetz Roms erstellt. Dieses System bezieht Ungewissheiten bezüglich Ankunftszeiten mit ein, indem sie zusätzlich zu den statischen Daten, die das Netz beschreiben, Echtzeitdaten über den urbanen Transport miteinbeziehen. Zudem haben sie historische GPS-Daten von Carpoolingfahrzeugen per *Mobility Profiling*, einem Data Mining Prozess, ausgewertet. Anhand dieser Auswertung haben sie alternative Routen über Carpooling erstellt. Dieses System haben sie einmal unter Einbeziehung der alternativen Busrouten und einmal ohne diese bereitgestellt, um Reiserouten für die Nutzer zu ermitteln. Nach einer Testphase haben sie das System ausgewertet und sie kamen zu dem Ergebnis, dass durch Einbeziehen der Carpoolingfahrzeuge mehr Anfragen beantwortbar waren und die Reisezeit im Durchschnitt geringer war. In diesem System wurde zwar versucht, Ankunftszeiten vorherzusagen, allerdings ohne ein zuvor erlerntes Machine Learning (ML)-Modell zu verwenden. Dies lag unter anderem daran, dass der Fokus auf der Routenplanung und nicht auf der Verspätungsvorhersage lag. Deshalb ist Anforderung A2 zum Teil erfüllt, da Ergebnisse aus einem erlernten Modell gemeinsam mit Echtzeitdaten verwendet wurden. Zudem gab es eine Kombination von Echtzeitdaten, statischen Daten, die das Netz beschreiben, und auf historischen Daten zuvor erlernten Informationen in Form der neuen Busrouten. Allerdings wurde dieses erlernte Modell nicht mehr anhand der Echtzeitdaten getestet oder verbessert. Da also weder ein Vorhersagemodell für Verspätungen erlernt, noch ein bestehendes Modell getestet oder angepasst wird, sind weder A1 noch A4 oder A5 erfüllt. Auch konnten Nutzer keinen Streckenzustand für ihre Anfragen selbst definieren und Störungsmeldungen wurden nicht miteinbezogen, weshalb weder A3 noch A6 erfüllt sind.

Raghothama et al. [Rag+16b] hingegen haben sich mit der Analyse von Verspätungen in zwei Verkehrsnetzen beschäftigt. Zum einen haben sie anhand des Systems aus [Ber+15] historische und Echtzeitverspätungsdaten betrachtet. Das Augenmerk lag hier jedoch eher auf der Erkennung und Berechnung von historischen Verspätungen. Hier wurden Fälle betrachtet, in denen nicht klar ist, zu welcher Fahrt eine Ankunftszeit an einer Haltestelle gehört. Anhand der berechneten Verspätungen wurden simple Analysen, wie die durchschnittliche Verspätung an einer Haltestelle oder für eine Fahrt im Tagesverlauf, durchgeführt. Mit diesen Analysen kann keine der Anforderungen erfüllt werden. Zum anderen haben sie sich mit einer Analyse von Verspätungsursachen für das Stockholmer Verkehrsnetz beschäftigt. Hier lagen wiederum statische Daten über die verschiedenen Verkehrsnetze vor sowie dynamische beziehungsweise Echtzeitdaten über Verspätungen, Störungen und Ähnliches innerhalb dieser Verkehrsnetze, welche sie über einen Zeitraum als historische Daten gesammelt haben. Anhand dieser historischen Daten wurden verschiedene Analysen durchgeführt. Es wurde

eine Untersuchung auf signifikante räumliche Korrelationen (anhand eines Moran's I Tests) und eine Hotspotanalyse (mittels KNN, siehe Abschnitt 4.4 und Kerneldichteinterpolation) durchgeführt. Zudem wurden Faktoren, die einen Einfluss auf Verspätungen haben mittels einer Regression (siehe Abschnitt 4.4) identifiziert (genauer letztendlich mittels negativer binomialer Regression). Da über die Regression ein Modell vorliegt, anhand dessen man auch Verspätungen vorhersagen kann, ist Anforderung A1 erfüllt. A6 ist teilweise erfüllt, da bei der Sammlung der historischen Daten eintreffende Störungsmeldungen erfasst und auf alle nachfolgenden Daten bezogen werden mussten. Die restlichen Anforderungen hingegen sind nicht erfüllt.

Im Gegensatz dazu war für Jiandong et al. [Jia+08] die Vorhersage von Verspätungen zur Erstellung eines automatischen Frühwarnsystems das Ziel. Genauer genommen sollten Verspätungsklassen für alle Flüge eines Flughafens anhand zahlreicher Bedingungen, darunter auch das Wetter, vorhergesagt werden. Dafür haben Jiandong et al. zunächst mittels Clustering Verspätungsklassen festgelegt, da die vorliegenden Verspätungsklassen der Flughäfen unbrauchbar waren. Genutzt wurden hierfür historische Daten eines Chinesischen Flughafens. Danach wurden verschiedene Arten von ML-Modellen (Entscheidungsbaum, Naive Bayes, neuronales Netz und Ridor Regel Modell) anhand der historischen Daten mit den zuvor bestimmten Klassen erlernt. Als bestes hat sich ein Entscheidungsbaum (siehe Abschnitt 4.4) herausgestellt, welcher für das Alarmsystem empfohlen wurde. Da auch hier nur ein Vorhersagemodell auf historischen Daten erlernt wird, ist wieder nur Anforderung A1 erfüllt.

Jeong und Rilett [JR04] hingegen haben sich mit der Vorhersage Busankunftszeiten beschäftigt. Sie haben historische Daten aus Houston, Texas verwendet. Diese Daten hatten zwei Teile. Zum einen sind dies Daten zu Fahrtzeiten zwischen den Haltestellen und Aufenthaltsdauer an den Haltestellen, welche als tatsächliche historische Daten genutzt wurden. Zum anderen sind es Daten zur Fahrzeugposition der Busse, über die die Fahrplaneinhaltung ermittelt werden kann, welche zur Simulierung von Echtzeitdaten genutzt wurden. Anhand dieser Daten wurden verschiedene Arten von ML-Modellen (ein spezielles auf historischen Daten basierendes Modell mit rekursiver Formel, ein multiples lineares Regressionsmodell und ein künstliches neuronales Netz) trainiert und getestet. Beim Test sollten die Ankunftszeiten anhand der Fahrzeugpositionen vorhergesagt werden. Am besten hat das neuronale Netz (siehe Abschnitt 4.4) funktioniert, vermutlich, da es diesem als einzigem Modell möglich war, anhand der Echtzeitdaten weiterzulernen, nachdem es auf den historischen Daten trainiert wurde. In ihrem System haben Jeong und Rilett sowohl ein Vorhersagemodell für Ankunftszeiten (und damit eben auch für Verspätungen) auf historischen Daten erlernt als auch dieses Modell in ihrem Test auf einige simulierte Echtzeitdaten angewandt. Damit ist A1 erfüllt und A2 nur teilweise erfüllt, da der aktuelle Streckenzustand nicht berücksichtigt wird. Zudem war es mit dem neuronalen Netz möglich, dass das Modell anhand von Echtzeitdaten weiter angepasst wird, wodurch A5 ebenfalls erfüllt ist. Allerdings wird beim Testen des Modells anhand der Echtzeitdaten nicht evaluiert, ob das Modell noch verwendbar ist und wenn das Modell zu schlecht wird, auch nicht aktiv eine Anpassung des Modells veranlasst, weshalb A4 nicht erfüllt ist.

Für Yaghini et al. [Yag+12] ist das Ziel die Erstellung eines möglichst genauen und schnellen Vorhersagemodells. Die Verspätungen der Passagierzüge im Iranischen Bahnnetz sollen vorhergesagt werden. Ihnen lagen historische Daten zu Verspätungen in ganzen Minuten vor, wobei Start, Ziel, Schienenabschnitt und Datum angegeben waren. Diese Daten haben sie in zehn Verspätungsklassen einteilen lassen, anhand derer später trainiert und getestet wurde. Für die Vorhersage haben sie eine Reihe von neuronalen Netzen entwickelt, welche untereinander und mit Entscheidungsbäumen

und multinomialer logistischer Regression verglichen wurden. Bei den neuronalen Netzen wurden drei Definitionsarten der Eingabeparameter und drei Methoden, die optimale Netzarchitektur zu finden, verglichen. Die entwickelten neuronalen Netze waren wesentlich besser als die alternativen Methoden. Bis auf eine Eingabedefinition waren fast alle Varianten der neuronalen Netze sehr schnell und präzise in der Vorhersage. Da hier jedoch wieder nur Vorhersagemodelle auf historischen Daten erlernt wurden, ist nur A1 erfüllt.

Sun et al. [Sun+16] beschäftigen sich mit der Vorhersage von Verspätungen von Bussen. Für sie ist jedoch das Ziel, den Mangel an Daten, welcher aufgrund kleiner Nahverkehrsnetze und seltener Fahrten in kleinen bis mittelgroßen Städten besteht, auszugleichen. Sie nutzen in ihrem System historische und Echtzeitdaten aus Nashville. In den historischen Daten liegen tatsächliche Abfahrts- und Ankunftszeiten sowie Aufenthaltszeiten von Bussen an den einzelnen Haltestellen vor und damit auch die Verspätungen. Als Echtzeitdaten erhalten sie die Fahrzeugpositionen der Busse mit Zeitpunkt der Messung. Sie unterscheiden in den Vorhersagen zwischen Langzeitvorhersagen und Kurzzeitvorhersagen. Langzeitvorhersagen sind alle Vorhersagen, die nicht für den aktuellen Tag sind oder die jenseits einer gewissen Schwelle zu weit in der Zukunft liegen. Für diese Vorhersagen nutzten sie ein Vorhersagemodell, das allein auf historischen Daten beruht. Es handelt sich um ein KNN-Modell (siehe Abschnitt 4.4), welches immer neu berechnet wird, wenn ein neuer Satz historischer Daten hinzugefügt wird. Für die Kurzzeitvorhersage wird anhand der letzten Positionsdaten des entsprechenden Busses eine bisherige Verspätung ermittelt. Auf diese wird für jeden Streckenabschnitt, der noch gefahren werden muss, eine vermutete zusätzliche Verspätung addiert. Diese wird anhand der anderen Fahrzeuge abgeleitet, die in einem gewissen Zeitraum diesen Streckenabschnitt durchfahren haben, wobei auch Fahrzeuge anderer Linien berücksichtigt werden. Durch diese Berücksichtigung anderer Linien wird der Mangel an Daten ausgeglichen. Anhand dieses kombinierten Vorhersagemodells haben sie bessere Vorhersageergebnisse erhalten, als das dortige Busunternehmen selbst. In ihrem System haben Sun et al. ein Vorhersagemodell auf historischen Daten erlernt, wodurch A1 erfüllt ist. Zudem nutzen sie Echtzeitdaten, um Anfragen zu klassifizieren und damit Verspätungen vorherzusagen, wodurch A2 teilweise erfüllt ist.

Eine weitere Arbeit befasst sich nicht direkt mit der Vorhersage von Verspätungen oder Ankunftszeiten, sondern mit der Verlässlichkeit von Busnetzen [Sun+17]. Verlässlichkeit bedeutet in diesem Zusammenhang, wie sehr die Werte schwanken und vom vorgegebenen Wert abweichen. Sun et al. [Sun+17] interessieren sich für die Verlässlichkeit bezüglich des Intervalls in dem die Busse fahren, bezüglich der Pünktlichkeit, also den realen Ankunfts- und Abfahrtszeiten im Vergleich zum Fahrplan, und bezüglich des Auslastungsgrades der Busse, also wie voll sie sind. Es gibt bereits Gleichungen zur Bestimmung eines aktuellen Verlässlichkeitswertes. Diese nutzen sie, um für historische Daten aus der Stadt Dalian (China) die Verlässlichkeit zu ermitteln. Die historischen Daten enthalten für jede Station und jeden Bus Positionen und Ankunftszeiten, welche über GPS ermittelt wurden, sowie die Zahl der zu- und aussteigenden Gäste, welche über Infrarotsensoren im Bus ermittelt wurden. Da Sun et al. an einer Vorhersage, nicht nur an der Bestimmung eines aktuellen Wertes interessiert sind, nutzen sie die berechneten Verlässlichkeitswerte, um einen Random Forest (ein ML-Verfahren welches auf der Nutzung mehrerer Entscheidungsbäume, siehe Abschnitt 4.4, beruht) und verschiedene Vergleichsmodelle (ein neuronales Netz und eine Support Vector Machine) zu trainieren und zu testen. Sie kamen zu dem Schluss, dass der Random Forest die besten Vorhersagen erbringen konnte. Da Sun et al. nur ein historisches Vorhersagemodell trainiert haben und dieses auch nur indirekte Hinweise auf Verspätungen liefern kann, wird nur Anforderung A1 teilweise erfüllt.

| | A1 | A2 | A3 | A4 | A5 | A6 |
|---|---|---|---|---|---|---|
| **Berlingerio et al. [Ber+15]** | | (x) | | | | |
| **Raghothama et al. [Rag+16b]** | x | | | | | (x) |
| **Jiandong et al. [Jia+08]** | x | | | | | |
| **Jeong und Rilett [JR04]** | x | (x) | | | x | |
| **Yaghini et al. [Yag+12]** | x | | | | | |
| **Sun et al. [Sun+16]** | x | (x) | | | | |
| **Sun et al. [Sun+17]** | (x) | | | | | |
| **Heppe und Liebig [HL17]** | x | x | | | | |

**Tabelle 3.1:** Evaluation von Systemen aus der Literatur, welche Mobilitätsdaten analysieren, gegen die Anforderungen aus Abschnitt 2.3.

Heppe und Liebig [HL17] legen wie Berlingerio et al. [Ber+15] das Augenmerk eher auf die Routenplanung, als auf Verspätungen. Im Gegensatz zu Berlingerio et al. ist für Heppe und Liebig jedoch die Berücksichtigung von Echtzeit-Verspätungsvorhersagen bei der Routenplanung der entscheidende Punkt. Sie wollen unter Berücksichtigung des Straßennetzes, den Fahrplänen des öffentlichen Nahverkehrs und Echtzeit-GPS-Daten der eingesetzten Fahrzeuge Vorhersagen für Verspätungen treffen. Unter Berücksichtigung der Verspätungsvorhersagen aller möglichen Routen soll dann eine optimale aktuelle Route empfohlen werden. Zur Vorhersage nutzen sie eine Art probabilistisches graphisches Modell, ein Spatio-Temporal-Random-Field (STRF). Dieses nutzt zeitliche und räumliche Abhängigkeiten der Verbindungen, welche als Graph dargestellt werden und wird auf historischen Daten trainiert. Über Echtzeit-GPS-Daten werden anhand der Fahrzeugpositionen die momentanen Verspätungen ermittelt. Diese werden wiederum genutzt, um anhand des STRF Verspätungen vorherzusagen. Dieses System testen sie anhand von Daten aus Warschau (Polen) und vergleichen es mit einem anderen Ansatz zu Verspätungsvorhersage, wobei sie die Verspätungen besser vorhersagen. Dieses System erlernt ein Vorhersagemodell und wendet es mit Echtzeitdaten auf Anfragen an, wodurch A1 und A2 erfüllt sind.

Wie in Tabelle 3.1 zu sehen, kann keines der bisher betrachteten Systeme alle Anforderungen erfüllen. Somit ist es erforderlich, dass ein neues System erschaffen wird, welches alle Anforderungen erfüllen kann. Hierfür ist es vor allem wichtig, dass sowohl historische als auch Echtzeitverkehrsdaten verarbeitet werden können und die Ergebnisse, beispielsweise in Form von historischen Modellen und Echtzeitzeitevaluation dieser Modelle, sich gegenseitig beeinflussen können. In Kapitel 4 werden dazu benötigte Verarbeitungsarten erklärt sowie Architekturen, die diese ermöglichen. Zudem ist durch Betrachtung dieser Fälle aus der Literatur klar geworden, dass unterschiedliche Autoren sehr unterschiedliche Meinungen haben, welches das beste ML-Verfahren, im Umgang mit Verkehrsdaten ist. Deshalb werden mögliche ML-Verfahren für die Umsetzung des Anwendungsfalls in Abschnitt 4.4 noch einmal genauer betrachtet.

# 4 Grundlagen

In diesem Kapitel werden einige Techniken, Konzepte und Systeme vorgestellt, welche zur Planung und Implementierung des Verspätungsvorhersagesystems benötigt werden. Zunächst werden in Abschnitt 4.1 zwei grundlegende Verarbeitungsarten für große Datenmengen beschrieben. Daraufhin werden in Abschnitt 4.2 verschiedene Architekturen vorgestellt, welche sowohl die Verarbeitung historischer Daten als auch Echtzeitdatenverarbeitung von Stromdaten unterstützen. In Abschnitt 4.3 werden verschiedene Systeme vorgestellt, die genutzt werden können, um die Verwendung beider Verarbeitungsarten aus Abschnitt 4.1 zu ermöglichen. Diese werden daraufhin untereinander verglichen und es wird ein geeignetes System für die Umsetzung des Anwendungsszenarios ausgewählt. Danach werden in Abschnitt 4.4 verschiedene Machine Learning (ML) Algorithmen vorgestellt, welche für das Szenario verwendbar sind, und es wird diskutiert, welche davon am besten geeignet sind.

## 4.1 Verarbeitungsarten

Es gibt verschiedene Ansätze, um große Mengen an Daten zu verarbeiten [CY15]. Die zwei grundlegend verschiedenen Verarbeitungsarten sind die Batchverarbeitung und die Streamverarbeitung.

Wie in Abschnitt 2.3 dargelegt, muss ein System für den gegebenen Anwendungsfall sowohl historische als auch Echtzeitdaten verarbeiten können. Die beiden Verarbeitungsarten sind jeweils für eine der Datenarten gut geeignet.

### Batch Processing

Die Verarbeitung als Batch ist für statische Daten geeignet, die persistent gespeichert sind [CY15; Has17]. Diese sind vorher meist über einen längeren Zeitraum gesammelt worden. Somit ist die Batchverarbeitung gut für historische Daten geeignet.

Bei der Verarbeitung wird der gesamte vorliegende Datensatz auf einmal verarbeitet. Meist werden die Daten in Teilmengen, Batches genannt, aufgesplittet und verteilt verarbeitet [CY15]. Hat die Verarbeitung begonnen, kann sie nicht unterbrochen werden. Es können weder neue Daten hinzugefügt werden noch kann die Verarbeitungslogik angepasst werden. Dies bedeutet, dass alle Daten, die miteinbezogen werden sollen, bereits vollständig im Speicher vorliegen müssen.

Aufgrund der großen Datenmengen, die auf einmal verarbeitet werden, dauert die Verarbeitung meist lange [CY15; Has17]. Zugleich ermöglicht die große Menge an Daten, die gleichzeitig und in Relation zueinander betrachtet werden können, sehr genaue Ergebnisse.

### Stream Processing

Streamverarbeitung hingegen ist für unbeschränkte Datenmengen gedacht, bei denen die Daten nach und nach eintreffen [CY15; Has17]. Damit ist sie gut für Echtzeitdaten geeignet.

Hierbei werden einzelne Datenobjekte oder kleine Datenmengen verarbeitet, sobald sie zur Verfügung stehen. Hierdurch erhält man sehr schnell Ergebnisse, weshalb diese Verarbeitungsmethode genutzt werden kann, wenn Echtzeitverarbeitung benötigt wird [CY15; Has17].

Die einzelnen Datenobjekte werden jedoch getrennt voneinander betrachtet, was die Erkennung von Zusammenhängen erschwert. Gleichzeitig ist es allerdings möglich, die Verarbeitungslogik jederzeit anzupassen, woraufhin alle nachfolgenden Daten mit der neuen Logik verarbeitet werden.

### Hybrid Processing

Es gibt jedoch Fälle, in denen sowohl eine schnelle Verarbeitung als auch große Genauigkeit der Ergebnisse gewünscht sind [CY15]. Batch- und Streamverarbeitung können jeweils nur eines von beiden erfüllen. In solchen Fällen gibt es die Möglichkeit, die beiden Verarbeitungsarten zu kombinieren. In Abschnitt 4.2 werden Architekturen vorgestellt, die solch eine hybride Verarbeitung ermöglichen.

## 4.2 Architekturen

Die *Lambda-Architektur* [MW15], die *Kappa-Architektur* [Kre14] und *Hybrid Processing Architecture for Big Data (BRAID)* [Gie+18] sind Architekturen, die sowohl die Verarbeitung historischer Daten als auch Echtzeitverarbeitung unterstützen.

### Lambda-Architektur

Die Lambda-Architektur [MW15], zu sehen in Abbildung 4.1, besteht aus separaten Zweigen für die Batch- und die Streamverarbeitung. Die Daten erreichen die Architektur als Strom und werden jeweils an beide Zweige weitergeleitet.

Im *Batch Layer* werden sie zunächst an ein *Master Dataset* angehängt. Die Daten aus dem *Master Dataset* werden regelmäßig via Batch Processing verarbeitet. Dies stellt eine Vorverarbeitung dar, in der die Daten für den Nutzer vorbereitet werden. Daraus entstandene Ergebnisse werden als sogenannte *Batch Views* abgespeichert, auf denen über das *Serving Layer* Anfragen ausgeführt werden können.

Da im *Batch Layer* nur Daten berücksichtigt werden können, welche zu Beginn jeder Berechnung vorliegen und die Berechnungen nur in bestimmten Abständen durchgeführt werden, werden in den *Batch Views* die neuesten Daten meist nicht berücksichtigt. Um dies auszugleichen werden über das *Speed Layer* die neuesten Daten via Streamverarbeitung in Echtzeit verarbeitet. Diese Ergebnisse stehen in den *Realtime Views* zur Verfügung.

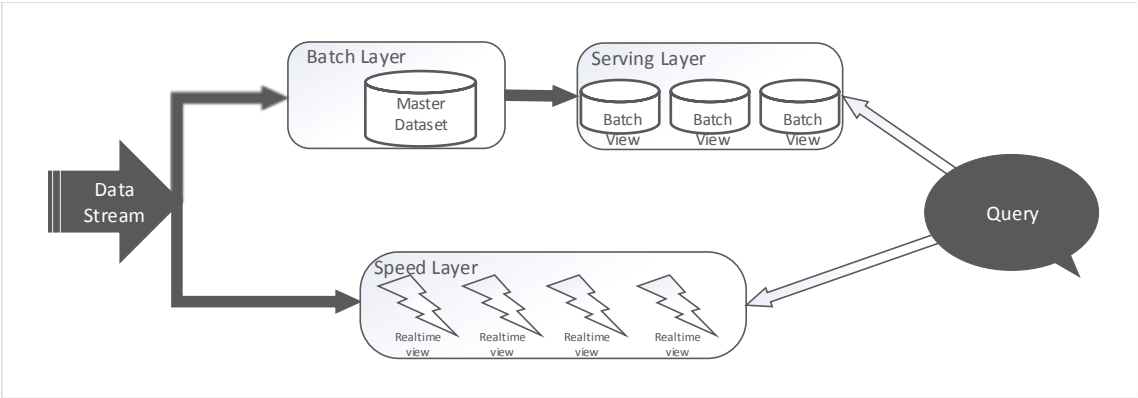



**Abbildung 4.1:** Die Lambda-Architektur (nach [MW15]).

Um Anfragen der Nutzer zu beantworten, werden nun die Informationen aus den *Batch Views* und den *Realtime Views* kombiniert, sodass alle Daten miteinbezogen werden, auch die neuesten.

In dieser Architektur sind die Batch- und die Streamverarbeitung in zwei strikt voneinander getrennten Zweigen zu finden, welche unterschiedliche Verarbeitungslogik bieten. Hierbei ist das *Batch Layer* sehr gut skalierbar, während die Logik im *Speed Layer* schnell angepasst werden kann. Dies kann jedoch dazu führen, dass die *Batch Views* und die *Realtime Views* unterschiedliche Datenschemata verwenden. Dadurch wird die Kombination von Daten aus beiden Layern und damit auch insgesamt der Zugriff auf Ergebnisdaten komplex [Gie+18].

Zudem stellen die beiden Zweige komplett voneinander getrennte Systeme dar, die nicht miteinander in Kontakt stehen. Diese müssen getrennt voneinander implementiert und auch getrennt voneinander gewartet werden. Dies führt zu einem hohen Aufwand, um die gesamte Architektur umzusetzen, zu betreiben und zu warten [Kre14].

Durch die Aufsplittung in zwei Systeme ist diese Architektur für den Anwendungsfall ungeeignet. Für die Erfüllung der Anforderungen muss es möglich sein, dass die Teile des Systems, die historische und Echtzeitdaten verarbeiten, immer wieder Daten austauschen. Nur so kann ein auf historischen Daten erlerntes Modell für die Klassifizierung von Echtzeitdaten verwendet werden. Auch der Test und die Anpassung des Modells anhand der Echtzeitdaten oder im Notfall sogar dessen Neuberechnung auf historischen Daten ist nur so möglich. Bei der Verwendung von zwei komplett voneinander getrennten Systemen ist dies nur schwer möglich.

### Kappa-Architektur

In der Kappa-Architektur [Kre14] wird versucht diese Probleme der Lambda-Architektur zu lösen, indem nur ein System genutzt wird. Wie in Abbildung 4.2 zu sehen, werden sowohl historische als auch Echtzeitdaten in das *Stream Processing System* geleitet und es gibt keine unterschiedlichen Pfade.

Die eintreffenden Daten aus dem Strom werden ebenfalls zunächst in einem Speicher abgelegt, dann werden sie direkt verarbeitet. Damit liegen immer Verarbeitungsergebnisse mit den neuesten Daten vor. Diese werden in ein *Serving Layer* geschrieben, auf welchem Anfragen ausgeführt werden können.

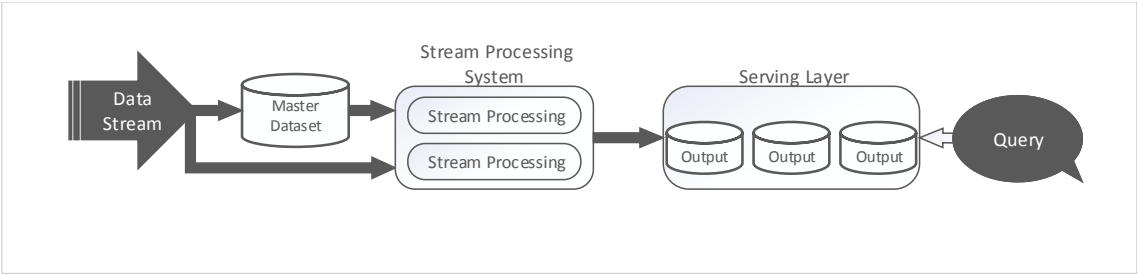



**Abbildung 4.2:** Die Kappa-Architektur (nach [Kre14]).

Sollen nun historische Daten erneut verarbeitet werden, weil sich beispielsweise die Verarbeitungslogik ändert und die alten Ergebnisse nicht mehr der aktuellen Logik entsprechen, wird ein paralleler Stream Processing Job gestartet. Dieser verarbeitet die abgespeicherten historischen Daten anhand der neuen Logik und schreibt die Ergebnisse ebenfalls ins *Serving Layer*. Dies geht so lange, bis er auf demselben Stand ist, wie der ursprüngliche Job. Daraufhin wird der alte Job, welcher noch mit der alten Logik arbeitet, beendet. Die historischen Daten werden also ähnlich zur Batchverarbeitung vollständig, mit einer während der Ausführung unveränderlichen Logik, verarbeitet.

Mit der Kappa-Architektur braucht man also nur ein System, welches dann parallel historische und Echtzeitdaten verarbeitet. Somit muss nur ein System gewartet werden, welches schnell anpassbar ist. Anstatt historische Daten, wie in der Lambda-Architektur, ständig zu verarbeiten, wird dies hier nur getan, wenn es nötig wird. Allerdings führt die Verarbeitungsweise in der Kappa-Architektur zu Verzögerungen der Anwendung der neuen Logik auf neue Daten, da mit der Menge der gespeicherten historischen Daten auch die Zeit ansteigt, bis der neue Stream Processing Job aufgeholt hat. Zudem kann die parallele Verarbeitung der historischen Daten zu den Echtzeitdaten zu Problemen bezüglich Rechenkapazitäten und Speicherbedarf führen, wenn mit großen Mengen historischer Daten gearbeitet werden soll [Win+16].

Dies macht die Kappa-Architektur ungeeignet für den Anwendungsfall, da es möglich sein soll, ein Modell auf großen Mengen historischer Daten zu erlernen, damit dieses möglichst genau ist.

### BRAID

Um die Probleme der Lambda- und der Kappa-Architektur zu umgehen und um zusätzlich den Austausch von Zwischenergebnissen zwischen historischer und Echtzeitverarbeitung zu ermöglichen, haben Giebler et al. [Gie+18] die Architektur *BRAID* vorgeschlagen. Diese zeichnet sich durch zwei Verarbeitungszweige, einen für *Stream* und einen für *Batch Processing*, sowie einen gemeinsam genutzten Speicher aus. Sie ist in Abbildung 4.3 zu sehen.

Auch hier werden alle eintreffenden Daten in einem *Master Dataset* gespeichert. Dieses befindet sich jedoch in einem gemeinsamen Speicher, auf den sowohl Stream als auch Batch Processing Zugriff haben. Zusätzlich werden die neuesten Daten, welche für die Echtzeitverarbeitung gedacht sind, vorübergehend in einem *Buffer* gespeichert. Aus dem gemeinsamen Speicher werden die Daten in einen der Verarbeitungszweige geleitet.

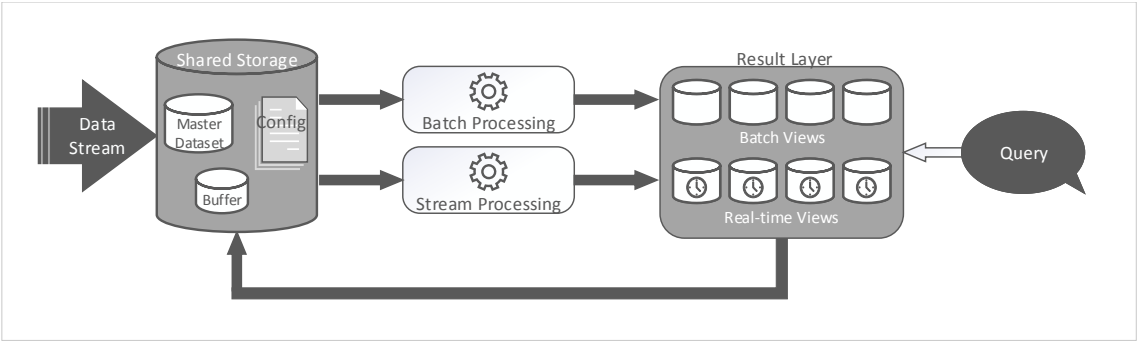



**Abbildung 4.3:** Die neue Architektur BRAID (nach [Gie+18]).

Das Besondere ist, dass Teile der Ergebnisse, welche nach der Verarbeitung in einem gemeinsamen *Result Layer* landen, in den gemeinsamen Speicher zurückgeführt werden können. Von dort aus können sie für zukünftige Verarbeitungen genutzt werden.

Der Batchverarbeitungszweig ist ähnlich zu *Batch* und *Serving Layer* in der Lambda-Architektur. Die Daten aus dem *Master Dataset* werden regelmäßig via Batch Processing verarbeitet. Hierbei können neben den historischen Daten auch Konfigurationen und Zwischenergebnisse aus dem gemeinsamen Speicher miteinbezogen werden. Die Ergebnisse werden nach der Verarbeitung in *Batch Views* im *Result Layer* gespeichert. Von dort aus können Teile zurück in den gemeinsamen Speicher geführt werden.

Im gemeinsamen Speicher ist zusätzlich, in Form von Konfigurationsdateien, die Verarbeitungslogik für beide Verarbeitungszweige gespeichert. Somit kann zum einen in beiden Zweigen dieselbe Verarbeitungslogik genutzt werden. Zum anderen kann die Verarbeitungslogik während der Laufzeit angepasst werden, da beide Zweige sowohl Lese- als auch Schreibzugriff auf den gemeinsamen Speicher haben.

Der Streamverarbeitungszweig ist ähnlich dem *Speed Layer* der Lambda-Architektur. Die Daten werden hier via Stream Processing verarbeitet. Der Buffer wird für die Echtzeitverarbeitung genutzt, da er schnellere Zugriffszeiten erlaubt. Wie bei der Batchverarbeitung können wieder Konfigurationen und Zwischenergebnisse aus dem gemeinsamen Speicher miteinbezogen werden. Ergebnisse der Verarbeitungen werden im *Result Layer* in sogenannte *Real-time Views* gespeichert und können zusätzlich genutzt werden, um Konfigurationen im gemeinsamen Speicher anzupassen sowie Zwischenergebnisse dort hineinzuschreiben. Außerdem kann ein Verhalten ähnlich der Kappa-Architektur erreicht werden, indem Daten aus dem Master Dataset über diesen Zweig verarbeitet werden.

Da alle Ergebnisse, sowohl die des Batchzweigs als auch die des Streamzweigs in einem gemeinsamen *Result Layer* landen, können Nutzer Anfragen direkt ans *Result Layer* senden, ohne darüber nachdenken zu müssen, aus welchem Zweig die Daten kommen. Unterschiedliche Datenschemata stellen hierbei ebenfalls kein Problem dar, da in vielen Speichersystemen Daten gemeinsam mit Metadaten gespeichert werden können, welche die Struktur beschreiben. Das *Result Layer* kann demzufolge die Daten selbstständig zusammenführen.

Durch diesen Aufbau ermöglicht BRAID verschiedene Verarbeitungsmodi. Es können sowohl pure Stream- als auch pure Batchverarbeitung betrieben werden. Außerdem können sowohl die Lambda- als auch die Kappa-Architektur emuliert werden. Zusätzlich gibt es einen selbstregulierenden

Modus, in dem sich die beiden Zweige durch ihre Ergebnisse beeinflussen können. Beispielsweise kann ein ML-Modell im Batch erlernt und getestet werden, welches dann sowohl im *Result Layer* landet als auch zurück in den gemeinsamen Speicher wandert. Daraufhin kann es im Stream zur Klassifizierung von eintreffenden Echtzeitdaten genutzt und eventuell durch diese sogar wiederum verfeinert werden. In diesem Modus ist die Architektur gut für den vorliegenden Anwendungsfall geeignet. Deshalb wird bei der Umsetzung des Anwendungsfalls BRAID als Grundlage für die Architektur genutzt.

## 4.3 Systeme für Batch- und Stream Processing

Für die endgültige Umsetzung wird allerdings nicht nur eine passende Architektur, sondern auch Systeme, die die benötigten Verarbeitungsarten beherrschen, gebraucht. Zwar ist es möglich, die beiden Verarbeitungszweige von BRAID mit unterschiedlichen Systemen umzusetzen, für einen Prototypen empfiehlt sich jedoch ein System, welches beide Verarbeitungszweige umsetzen kann. Giebler et al. [Gie+18] empfehlen hierfür drei verschiedene Systeme: Flink, Samza und Spark. Auf diese wird im Folgenden eingegangen.

### Flink

Flink[1] ist ein Open-Source-Framework der Apache Software Foundation zur Datenstromverarbeitung und arbeitet mit kontinuierlichen Datenströmen. Es bietet Möglichkeiten zur verteilten Ausführung.

Flink arbeitet unter der Prämisse, dass alle Daten als Datenstrom aufgefasst werden können. Die Unterscheidung zwischen tatsächlichen Datenströmen und festen Datenmengen geschieht, indem ersteres als unendlicher, letzteres als finiter Datenstrom betrachtet wird. Indem große Mengen an Daten aus Datenbanksystemen als Datenstrom verarbeitet werden, werden geringe Verarbeitungszeiten erreicht. Eine verteilte Ausführung wird erreicht, indem die Datenströme und die darauf arbeitenden sogenannten Operatoren partitioniert werden.

Implementiert wird bei der Nutzung von Flink entweder mit Java, oder mit Scala. Das Framework bietet in der Streamverarbeitung verschiedene Datenquellen, wie Kafka[2] oder TCP Sockets. Es kann in der Batchverarbeitung auch mit verschiedenen Datenquellen arbeiten, darunter SQL-basierte Datenbanksysteme oder verteilte Dateisysteme wie Hadoop Distributed File System (HDFS)[3]. Zusätzlich werden über APIs und Libraries verschiedene zusätzliche Verarbeitungsmöglichkeiten geboten, wie beispielsweise ML-Algorithmen über *FlinkML*. Allerdings bietet *FlinkML* nur eine geringe Zahl an ML-Algorithmen, einen für jede vertretene Kategorie. Die vertretenen Kategorien sind aus dem Bereich *Supervised Learning* (dt. überwachtes Lernen) eine Klassifizierung und eine Regression, aus dem Bereich *Unsupervised Learning* (dt. unüberwachtes Lernen) Clustering sowie Alternating Least Squares für Faktorisierungsprobleme.

[1] siehe: https://flink.apache.org/
[2] siehe: http://kafka.apache.org/
[3] siehe: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

### Samza

Auch Samza[4] ist ein Open-Source-Framework der Apache Software Foundation. Es ist ein Framework für verteiltes Stream Processing.

Samza arbeitet direkt auf Datenströmen. Der Strom wird in Partitionen unterteilt, welche verteilt verarbeitet werden, wodurch Samza gut skalierbar ist. Zum Austausch von Nachrichten zwischen *Jobs*, den Verarbeitungsschritten in Samza, wird Kafka verwendet. Um die Jobs zu verteilen, wird der Ressourcenmanager Apache Hadoop YARN [Vav+13] verwendet. Zudem erlaubt Samza über dieselbe Stream-Processing Engine eine Art Batchverarbeitung, bei der eine Streamverarbeitungslogik auf einen statischen Datensatz angewandt wird.

Samza wird mit Java implementiert und kann zusätzlich zu den Streamdatenquellen sowohl auf HDFS als auch mit SQL Datenquellen genutzt werden. Allerdings hat Samza keine API oder Library für die Nutzung von ML-Algorithmen, was dieses Framework für den gegebenen Anwendungsfall ungeeignet macht. Im Folgenden wird Samza darum nicht mehr betrachtet.

### Spark

Spark[5] ist ebenfalls ein Open-Source-Framework der Apache Software Foundation. Im Gegensatz zu Samza und Flink ist es jedoch ein Framework für Batchverarbeitung. Es unterstützt verteilte Verarbeitung und ist skalierbar und fehlertolerant.

Spark arbeitet mit Batchverarbeitung, wobei Datensätze für die verteilte Verarbeitung unterteilt werden. Auf diesen verteilten Datensätzen können für den gesamten Datensatz unterschiedliche Verarbeitungsschritte durchgeführt werden. Alternativ ist über die *Spark Streaming* API die Möglichkeit geboten, Datenströme zu verarbeiten. Die Ströme werden hierzu in sogenannte Micro-Batches unterteilt und als Reihe von kurzen Batch Jobs mit geringer Latenz verarbeitet.

Für die Implementierung können Scala, Java, Python und R verwendet werden. Spark war ursprünglich für die Nutzung mit HDFS gedacht, über eine API sind aber auch SQL-basierte Systeme kompatibel. Zudem können im Zusammenhang mit der Streaming API Streamressourcen, wie TCP Sockets oder Kafka, genutzt werden. Wie Flink bietet auch Spark über APIs und Libraries verschiedene zusätzliche Verarbeitungsmöglichkeiten an, darunter auch *MLlib* für ML-Algorithmen, welches eine Vielzahl an Algorithmen bereitstellt.

## 4.4 Machine Learning für multiclass Klassifizierung

In dem Anwendungsfall aus Kapitel 2 liegen historische Daten vor, für die die Verspätung bereits bekannt ist. Für eine Vorhersage von Verspätungen, wie sie für den Anwendungsfall gebraucht werden, kann also *Supervised Learning* (dt. überwachtes Lernen) betrieben werden [RN12]. Dies bedeutet, dass der ML-Algorithmus für die historischen Daten betrachtet, inwiefern die beobachtete

[4] siehe: http://samza.apache.org/
[5] siehe: https://spark.apache.org/

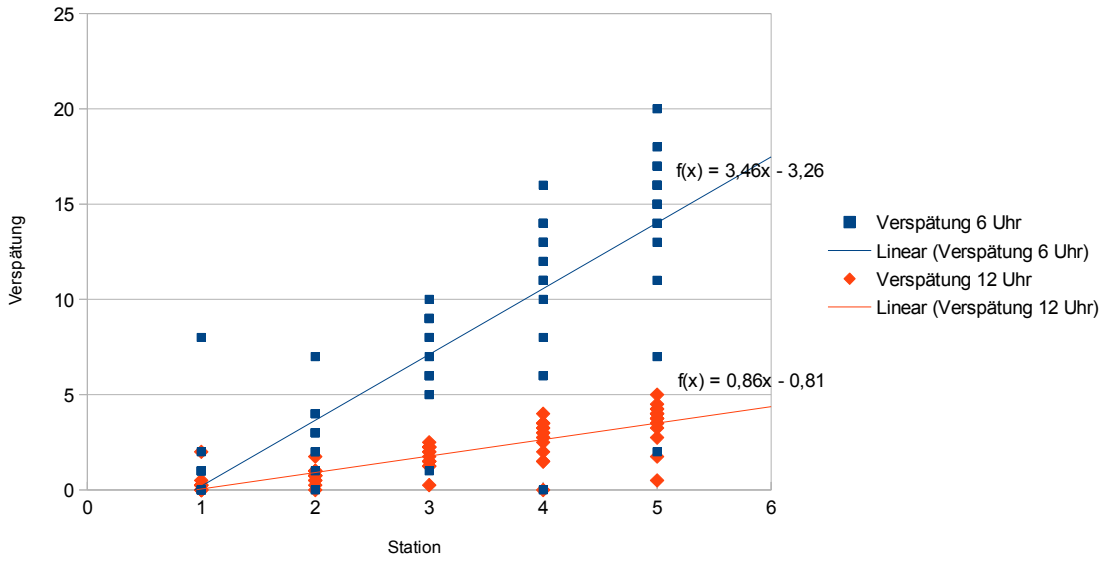


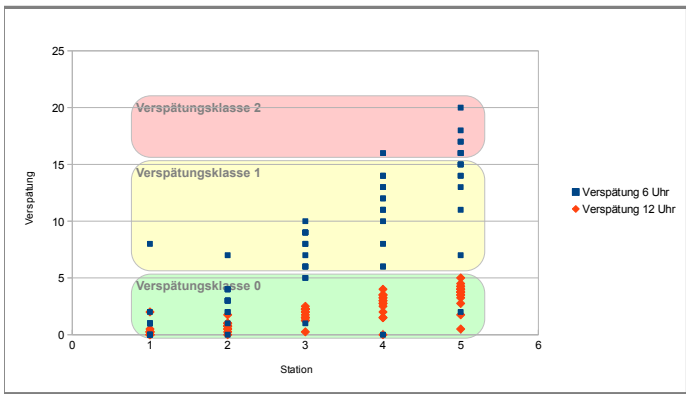

**Abbildung 4.4:** Ein beispielhaftes Regressionsmodell, welches für den Anwendungsfall entstehen könnte.

Verspätung mit den Parametern zusammenhängt, die den Kontext der Fahrt beschreiben. Dabei wird eine Abbildung von Parametern auf Verspätung abgeleitet, die diesen Zusammenhang möglichst genau beschreibt. Es kommen zwei Kategorien von ML-Algorithmen in Frage:

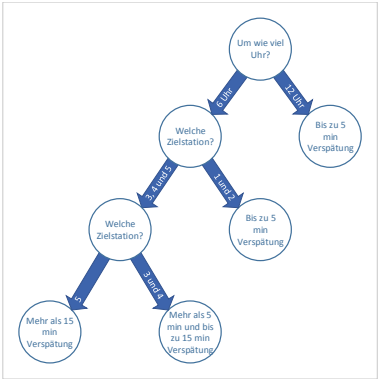
**Regression** Hier liegt die Verspätung in den historischen Daten in Form von kontinuierlichen Zahlenwerten vor [RN12]. Es soll eine mathematische Funktion ermittelt werden, die den Zusammenhang zwischen Eingabeparametern und dem Verspätungswert beschreibt. Modelle könnten beispielsweise wie in Abbildung 4.4 aussehen. Die Punkte stehen hierbei jeweils für die historischen Daten und die gleichfarbigen Linien sind die abgeleiteten linearen Regressionsmodelle. Anhand dieser Funktion werden abhängig von Eingabeparametern kontinuierliche Zahlenwerte für die vermutete Verspätung berechnet.

**Klassifizierung** Hier gibt es eine beschränkte Zahl an möglichen Verspätungsklassen, in welche die historischen Daten, die zum Erlernen des Modells verwendet werden, bereits eingeteilt sind. Es soll ein Modell ermittelt werden, welches Zusammenhänge zwischen den Eingabeparametern und der Einteilung in diese Klassen beschreibt. Bei der Klassifizierung werden die Eingabeparameter einer der Verspätungsklassen zugeordnet [RN12]. Ein Modell könnte zum Beispiel wie in Abbildung 4.5 aussehen. In Abbildung 4.5a sind die historischen Daten zu sehen, die dem Modell zugrunde liegen, wobei die verschiedenen Klassen jeweils mit farbigen Kästen hervorgehoben sind. Abbildung 4.5b zeigt einen Entscheidungsbaum, der aus diesen Daten erlernt werden könnte. Entscheidungsbäume sind eine Art von Klassifizierungsalgorithmen, auf die später genauer eingegangen wird.

In der Praxis gibt es viele ML-Algorithmen, die sowohl Regression als auch Klassifizierung beherrschen [Gun+98; LW+02]. Da für die Auswahl des Algorithmus die Unterscheidung zwischen Regression und Klassifizierung nicht so wichtig ist, wird einfachheitshalber für diesen Anwendungs-




**(a)** Beispielhafte Verspätungsdaten, bei denen die Verspätungsklassen markiert wurden.




**(b)** Entscheidungsbaum, der ausgehend von Abbildung 4.5a generiert werden könnte

**Abbildung 4.5:** Ein beispielhaftes Klassifizierungsmodell, welches für den Anwendungsfall entstehen könnte.

fall eine Klassifizierung gewählt. Es ist wünschenswert, dass unterschiedlich große Verspätungen vorhergesagt werden können und nicht nur, ob eine Verspätung auftritt oder nicht. Für die Klassifizierung muss deshalb ein *multiclass* (dt. Mehrklassen-) Klassifizierungsalgorithmus verwendet werden, also ein Algorithmus, der mit mehr als zwei möglichen Klassen arbeiten kann. Es gibt verschiedene Möglichkeiten, wie multiclass Klassifizierung betrieben werden kann [Aly05]. Als erstes gibt es Algorithmen, die entweder von Natur aus bereits mit mehreren Klassen umgehen können oder bei denen der Algorithmus auf natürliche Weise für mehrere Klassen erweitert werden kann. Die nächste Möglichkeit ist, das Mehrklassenproblem in mehrere binäre Klassifizierungsprobleme aufzuspalten und diese jeweils mit einer binären Klassifizierungsmethode zu lösen. Als letzte Möglichkeit kann eine hierarchische Klassifizierung verwendet werden, bei welcher das Problem als Baum aus binären Klassifizierungen gelöst wird. Im Folgenden werden diese Möglichkeiten genauer betrachtet.

### Erweiterbare Algorithmen

Es gibt einige Klassifizierungsalgorithmen, welche entweder von Natur aus bereits mit mehreren Klassen umgehen können oder bei denen der Algorithmus auf natürliche Weise für mehrere Klassen erweitert werden kann.

**Künstliche Neuronale Netze** Künstliche Neuronale Netze sind Klassifizierungsmodelle, welche den neuronalen Verknüpfungen im Gehirn nachempfunden sind. Eine Art von neuronalen Netzen, welche gut für Klassifizierung genutzt werden und einfach für mehrere Klassen erweitert werden können, sind mehrschichtige Feedforward-Netze. Wie in Abbildung 4.6 zu sehen, sind sie gerichtete Graphen aus Knoten, in neuronalen Netzen, auch Neuronen genannt, welche in mehreren Schichten angeordnet sind [RN12].

Knoten aus einer Schicht sind über gewichtete Verknüpfungen mit Knoten aus der darauffolgenden verbunden. Es gibt eine Reihe an Eingabeknoten, welche die Merkmale erhalten, anhand derer klassifiziert werden soll. Die Daten werden von dort aus durch eine oder mehrere sogenannte versteckte Knotenschichten geleitet, in denen verarbeitet wird. Die Knoten erhalten

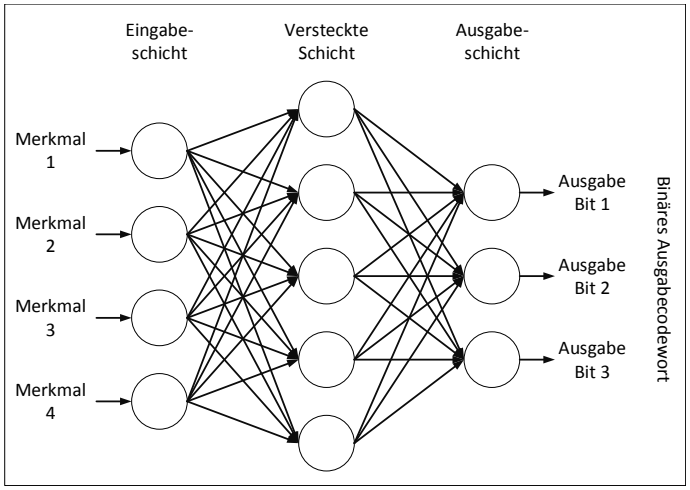



**Abbildung 4.6:** Schematischer Aufbau eines mehrschichtigen neuronalen Feedforward-Netzes.

die gewichteten Eingaben aller ihrer Vorgänger, aus welchen sie anhand einer Aktivierungsfunktion errechnen, ob der Knoten „feuert“, also eine 1 an die nachfolgenden Knoten ausgibt, oder nicht und eine 0 ausgibt [RN12]. Am Ende stehen ein oder mehrere Ausgabeknoten, die ein binäres Ergebnis ausgeben. Indem man mehrere Ausgabeknoten verwendet, kann man Neuronale Netzwerke für eine Klassifizierung mit mehreren Klassen verwenden. Hierzu werden die binären Ausgaben aller Knoten zu einem Codewort zusammengesetzt. Es gibt zwei Varianten diese Codewörter den Klassen zuzuordnen:

*one-per-class coding*: Es gibt einen Ausgabeknoten pro Klasse. Wird eine Klasse gewählt, wird an deren Knoten eine 1, an allen anderen Knoten eine 0 ausgegeben.

*distributed-output coding*: Jeder Klasse wird zu Beginn ein unterschiedliches binäres Codewort zugeordnet, wobei alle Codewörter dieselbe Länge haben. Die Länge der Codewörter entspricht der Anzahl an Ausgabeknoten, die das Neuronale Netzwerk haben muss. Bei der Klassifizierung wird die Klasse gewählt, deren zugeordnetes Codewort dem berechneten Codewort, welches von den Ausgabeknoten ausgegeben wurde, am ähnlichsten ist. Für die Ähnlichkeit der Codewörter können verschiedene Metriken genutzt werden. Bei der simpelsten wird die Anzahl der Bits betrachtet, welche verändert werden müssen, damit die Codewörter identisch sind. Ist zum Beispiel das Codewort 00000 der Klasse 1 und 10100 der Klasse 2 zugeordnet, so wird beim Ausgabecodewort 10110 Klasse 2 gewählt.

**Entscheidungsbäume** Eine weitere Möglichkeit für die multiclass-Klassifizierung bieten Entscheidungsbäume. Ein Beispiel hierfür war bereits in Abbildung 4.5b zu sehen. Dies sind Bäume, bei denen jeder Knoten einer Aufsplittung anhand eines Merkmals entspricht. Den Blattknoten ist jeweils eine Klasse zugeordnet. Bei der Klassifizierung wird der Baum anhand der Werte der Merkmale traversiert. An jedem Knoten wird anhand eines der Merkmale eine Entscheidung für einen der ausgehenden Pfade getroffen, bis ein Blattknoten erreicht wird. Die Klasse, die dem erreichten Knoten zugeordnet ist, wird ausgewählt. Mehrklassige Klassifizierung ist hier möglich, indem für jede Klasse mindestens ein Blattknoten existiert.

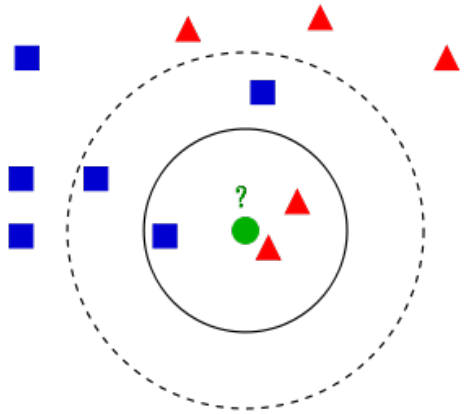



**Abbildung 4.7:** Beispiel einer Klassifizierung via KNN. Beim durchgezogenen innerern Kreis wird 3NN, beim gestrichelten äußeren Kreis 5NN verwendet [Fab10].

**k-nächste Nachbarn (KNN)** Dieser Algorithmus betrachtet Daten als Punkte in einem Koordinatensystem. Er beruht auf der Berechnung der Distanz eines Beispiels zu jedem anderen Trainingsbeispiel. Es werden die k Nachbarn mit der geringsten Distanz zum aktuell betrachteten Datenpunkt ausgesucht. Wird der Algorithmus zur Klassifizierung verwendet, wird die Klasse ausgewählt, die in den nächsten Nachbarn am häufigsten vertreten ist. In Abbildung 4.7 ist dies beispielhaft zu sehen. Der Punkt soll klassifiziert werden. Dies wird einmal, bei dem durchgezogenen inneren Kreis, mit 3NN aufgezeigt, hier würden die Dreiecke als Klasse gewählt werden. Dann wird bei dem gestrichelten äußeren Kreis 5NN aufgezeigt, wobei hier die Quadrate überwiegen und als Klasse gewählt werden. Da in den Trainingsbeispielen beliebig viele Klassen vorkommen können, ist dieser Algorithmus für multiclass-Klassifizierung geeignet.

**Naive Bayes** Naive Bayes arbeitet mit dem Ansatz einer Maximum-a-posteriori-Hypothese. Dies bedeutet, dass Eingabeparameter gesucht werden, durch die die Ausgabe maximiert wird. Betrachtet werden die Wahrscheinlichkeiten, dass eine der Klassen eintritt, wenn die gegebenen Merkmale vorliegen. Hier werden die beobachteten Wahrscheinlichkeiten für die einzelnen Klassen und für die vorkommenden Merkmale unter der Bedingung, dass diese Klasse eintritt verwendet. Um die Berechnung zu vereinfachen, wird in diesem Ansatz angenommen, dass die Merkmale voneinander unabhängig sind, wenn die Klasse gegeben ist, unabhängig davon, ob dies zutrifft. Dadurch muss für alle Klassen nur das Produkt der Wahrscheinlichkeit der Klasse und einzeln aller vorkommenden Merkmale unter der Bedingung, dass diese Klasse eintritt, berechnet werden. Mit Klasse $c$ und Merkmalen $x_1, ..., x_n$ wird also folgendes berechnet: $P(C = c)P(x_1|C = c)...P(x_n|C = c)$. Es wird die Klasse gewählt, für die dieser Wert am höchsten ist. Dies kann mit einer beliebigen Gesamtzahl an Klassen berechnet werden, womit der Algorithmus für multiclass-Klassifizierung geeignet ist.

**Support Vector Machine** Wie bei kNN werden die historischen Daten als Punkte in einem Koordinatensystem betrachtet. In der Grundform der Support Vector Machine wird versucht, eine Grenze zu finden, die den maximalen Abstand zwischen zwei vorhandenen Klassen bietet. In anderen Worten ist es ein Optimierungsproblem, bei dem versucht wird, eine Hyperebene zu finden, die den Abstand zu den Datenpunkten beider Klassen, welche der Hyperebene am nächsten liegen maximiert. In Abbildung 4.8 ist ein Beispiel hierfür zu

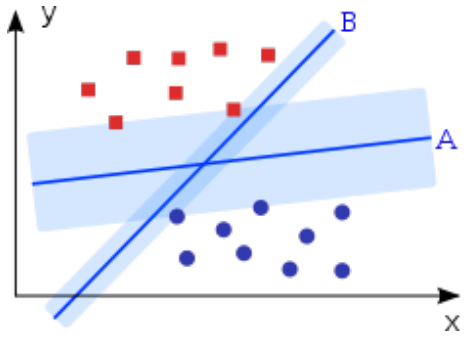



**Abbildung 4.8:** Beispiel einer Support Vector Machine, Linie A ist die Hyperebene, die als Modell gewählt ist [Fab10].

sehen. Linie B trennt zwar beide Klassen, bietet allerdings einen geringen Abstand zwischen beiden Klassen. Linie A hingegen maximiert die Distanz zu beiden Klassen und wird als Modell gewählt. Bei der Klassifizierung wird betrachtet, auf welcher Seite der Hyperebene der Datenpunkt sich befindet und die entsprechende Klasse gewählt. All dies funktioniert zunächst nur für zwei Klassen. Es wurden verschiedene Erweiterungen des Algorithmus vorgeschlagen, welche die Anwendung auf multiclass-Klassifizierungsprobleme ermöglichen. Bei diesen werden zusätzliche Parameter und Nebenbedingungen eingeführt, um die Separation mehrerer Klassen zu ermöglichen.

### Dekomposition in binäre Klassifizierung

Ein multiclass-Klassifizierungsproblem kann auch gelöst werden, indem nicht der Algorithmus erweitert, sondern das Problem aufgespalten wird. Hierzu wird das Problem in mehrere binäre Klassifizierungsprobleme aufgespalten, welche nacheinander von einem binären Klassifizierungsmodell gelöst werden können. Es gibt mehrere Ansätze, wie diese Dekomposition geschehen kann.

**One-versus-all (OVA)** Dies ist der einfachste Ansatz, er wird auch One-versus-rest (OVR) genannt. Für jede Klasse des ursprünglichen Problems wird ein neues binäres Problem erstellt mit der Entscheidung, ob ein Beispiel zu dieser Klasse oder zu einer der anderen Klassen gehört. Die binären Klassifizierungsmodelle müssen jedoch zusätzlich zur Entscheidung auch noch angeben, wie sicher diese Entscheidung ist, sodass bei der Klassifizierung die Klasse gewählt wird, welche mit der größten Sicherheit zutreffend war.

**All-versus-all (AVA)** In diesem Ansatz, auch One-versus-one (OVO) genannt, wird jede Klasse mit jeder anderen Klasse verglichen. Es wird für jedes mögliche Klassenpaar ein binäres Problem erstellt mit der Entscheidung, ob ein Beispiel zur ersten Klasse oder zur zweiten Klasse gehört. Nachdem jedes Klassifizierungsmodell seine Entscheidung getroffen hat, wird letztendlich die Klasse genommen, welche am häufigsten gewählt wurde.

**Error-Correcting Output-Coding (ECOC)** In diesem Ansatz wird jede Klasse mit einem unterschiedlichen binären Codewort derselben Länge codiert. Dann werden binäre Klassifizierungsmodelle in der Anzahl der Länge der Codewörter benötigt. Nun werden die Codewörter in einer Matrix so angeordnet, dass die Klassen mit zugeordnetem Codewort jeweils die Zeilen ergeben und jedes Klassifizierungsmodell einer Spalte mit einem anderen Bit aller Codewörter

| | **Modell 1** | **Modell 2** | **Modell 3** | **Modell 4** | **Modell 5** |
|---|---|---|---|---|---|
| **Klasse 0** | 0 | 0 | 0 | 0 | 0 |
| **Klasse 1** | 0 | 1 | 1 | 0 | 1 |
| **Klasse 2** | 1 | 1 | 0 | 0 | 0 |
| **Klasse 3** | 1 | 1 | 1 | 1 | 0 |

**Tabelle 4.1:** Beispiel für eine ECOC-Matrix.

zugeordnet ist. Ein Beispiel hierfür ist in Tabelle 4.1 zu sehen. Die Klassifizierungsmodelle werden anhand der Spalten der Matrix trainiert. Bei der Klassifizierung werden die binären Entscheidungen der Modelle der Matrix entsprechend zu einem Codewort zusammengefügt. Die Klasse, deren Codewort dem berechneten Codewort am ähnlichsten ist, wird ausgewählt.

**Generalized Coding** Dieser Ansatz stellt eine Generalisierung des ECOC dar, bei der die Matrix die zusätzliche Möglichkeit bietet, dass eine Klasse von einem Klassifizierungsmodell ignoriert wird. Analog zu ECOC wird bei einer Klassifizierung die Klasse gewählt, deren Codewort dem berechneten am ähnlichsten ist.

### Hierarchische Klassifizierung

Bei dieser Methode werden die Klassen hierarchisch in einem Baum angeordnet, wobei der Wurzelknoten alle Klassen und jeder Kindknoten jeweils nur eine Teilmenge der Klassen seines Elternknotens enthält. Dies setzt sich bis zu den Blättern fort, die jeweils eine Klasse enthalten. Bei der Klassifizierung trifft an jedem Knoten ein binäres Klassifizierungsmodell die Entscheidung für einen der Kindknoten, bis ein Blattknoten und damit die Entscheidung für eine endgültige einzelne Klasse erreicht ist.

### Die Methoden im Vergleich

In der Literatur gibt es bereits einige Vergleiche hinsichtlich der Genauigkeit der Vorhersagen von Klassifizierungsmethoden, um die Auswahl eines Algorithmus für multiclass-Klassifizierung zu erleichtern.

Li et al. [Li+04] und Statnikov et al. [Sta+04] haben jeweils verschiedene Algorithmen für multiclass-Klassifizierung anhand von Anwendungsfällen aus dem Bereich der Biologie bezüglich Daten zur Genexpression verglichen. Hierbei haben Li et al. folgende Algorithmen getestet: Support Vector Machines (mit den Dekompositionsmethoden OVA, AVA und ECOC), Naive Bayes, KNN und eine Variante von Entscheidungsbäumen. Statnikov et al. hingegen haben Support Vector Machines (in zwei Varianten einer natürlichen Erweiterung des Algorithmus und mit den Dekompositionsmethoden OVA, AVA und einer AVA-Abwandlung namens DAGSVM, bei der die Klassifizierung etwas optimiert wurde), KNN und zwei Varianten von neuronalen Netzen getestet. Beide kamen zu dem Schluss, dass Support Vector Machines am Besten abgeschnitten haben. Li et al. konnten keinen Unterschied zwischen den Dekompositionsmethoden ausmachen und fanden, dass KNN ebenfalls recht gut abschneidet. Im Gegensatz dazu kamen Statnikov et al.

zu dem Schluss, dass die Dekompositionsmethode OVA und die beiden natürlichen Erweiterung der Support Vector Machine untereinander nicht signifikant unterschiedlich waren, während alle anderen Methoden signifikant schlechter abgeschnitten haben.

Li et al. [Li+03] haben glsml-Algorithmen anhand von Anwendungsfällen aus dem Bereich der Musikgenreklassifikation verglichen. Sie haben unter anderem Support Vector Machines (OVA, AVA, und einer Variante der natürlichen Erweiterung des Algorithmus) und KNN getestet. Die Ergebnisse zeigen, dass Support Vector Machines immer am besten abgeschnitten haben, wobei die beste Dekompositionsmethode problemabhängig ist.

Mehra und Gupta [MG13] hingegen haben anhand offener Datensätze aus verschiedenen Umfeldern getestet, welche häufig für den Test von Klassifizierungmethoden verwendet werden. Diese Datensätze sind *iris*, *wine*, *glass* und *vowel*. Getestet wurden ein (mehrschichtiges Feedforward) neuronales Netz, ein Entscheidungsbaum, KNN, Naive Bayes und Support Vector Machines (in einer Variante der natürlichen Erweiterung des Algorithmus und mit den Dekompositionsmethoden OVA, AVA und der AVA-Abwandlung DAGSVM). Zudem wurde mit einem zuvor nicht verwendeten und dem *vowel*-Datensatz hierarchische Klassifizierung getestet. Sie kamen ebenfalls zu dem Ergebnis, dass Support Vector Machines am Besten abgeschnitten haben, wobei es vom Datensatz abhängig war, welche Variante am besten war und auch KNN und Naive Bayes teilweise sehr gut abgeschnitten haben.

Jiandong et al. [Jia+08], welche sich primär mit der Vorhersage von Verspätungen bei Flügen beschäftigt haben, haben unter anderem Naive Bayes, Entscheidungsbäume und neuronale Netze ausprobiert. Sie kamen mit dem Entscheidungsbaum auf die größte Genauigkeit.

Es lässt sich also, wie in Tabelle 4.2 zu sehen, eine allgemeine Tendenz zu Support Vector Machines als beste Lösung bezüglich Genauigkeit der Berechnungen erkennen. Hierbei war die optimale Variante des Algorithmus jedoch stark vom jeweiligen Problem abhängig. Da allerdings nicht immer alle Methoden getestet wurden und die Anwendungsfälle meist recht speziell und dem Anwendungsfall aus Kapitel 2 recht unähnlich waren, lässt sich noch keine endgültige Präferenz für den gegebenen Anwendungsfall treffen. Daher sind Tests verschiedener Methoden anhand dieses Anwendungsfalls sinnvoll. Diese folgen in Abschnitt 6.2.

| **Art der Daten** | Genexpression [Li+04] [Sta+04] | Musikgenre [Li+03] | offene Datensätze (*iris*, *wine*, *glass*, *vowel*) [MG13] | Flugverspätung [Jia+08] |
|---|---|---|---|---|
| **neuronales Netz** | x | | x | x |
| **Entscheidungsbaum** | x | | x | x |
| **KNN** | x | x | x | |
| **Naive Bayes** | x | | x | x |
| **Support Vector Machine natürlich erweitert** | x | x | x | |
| **OVA Support Vector Machine** | x | x | x | |
| **AVA Support Vector Machine** | x | x | x | |
| **ECOC Support Vector Machine** | x | | | |
| **Hierarchische Klassifizierung** | | | (x) | |
| **am Besten** | Support Vector Machine (Art problemabhängig) | Support Vector Machine (Art problemabhängig) | Support Vector Machine (Art problemabhängig) | Entscheidungsbaum |

**Tabelle 4.2:** Ergebnisse der Vergleiche von Algorithmen für multiclass-Klassifizierung.

# 5 Konzept

Im Folgenden wird betrachtet, wie ein System aussehen muss, welches den Anwendungsfall und die zugehörigen Anforderungen erfüllen kann. Dazu werden zunächst in Abschnitt 5.1 die Daten erläutert, welche vom System verarbeitet werden. Danach wird in Abschnitt 5.2 die eigens hierfür entworfene Architektur vorgestellt, woraufhin in Abschnitt 5.3 auf Verfahren zum Erlernen des Vorhersagemodells eingegangen wird.

## 5.1 Nahverkehrsdaten

Nahverkehrsdaten bestehen in der Regel aus zwei Teilen [Ber+15; HL17; Rag+16b; Ver14; Ver18c; Ver18e]. Der erste Teil sind statische Daten, welche das Verkehrsnetz, sowie die Sollwerte für Ankunfts- und Abfahrtszeiten der Fahrzeuge und ähnliches beschreiben. Hier werden zum Beispiel die Haltestellen beschrieben, unter anderem wo sie sich befinden und wie sie heißen. Außerdem werden beispielsweise die Linien beschrieben, also aufeinanderfolgende Haltestellen, an denen die Fahrzeuge entlangfahren. Zudem werden beispielsweise einzelne Fahrten entlang dieser Linien beschrieben, also wann ein Fahrzeug, wenn es einmal diese Linie abfährt, an jeder einzelnen Haltestelle ankommen soll, wie lange es sich dort aufhalten soll und wann es wieder abfahren soll. Diese stellen den Kontext dar, anhand dessen die andere Art von Informationen interpretiert werden kann.

Diese zweite Art von Daten sind dynamische Daten, die den Istzustand des Netzes beschreiben. Es sind Echtzeitdaten, die während des Betriebes des Netzes entstehen und zum einen als historische Daten gesammelt werden können, zum anderen in Echtzeit genutzt werden können. Sie beschreiben beispielsweise wann einzelne Fahrzeuge tatsächlich an bestimmten Haltestellen ankommen und abfahren, ob Störungen der Infrastruktur des Netzes oder einzelner Fahrzeuge vorliegen, und weiteres.

Werden die dynamischen mit den statischen Daten kombiniert, können die Verspätungen einer Fahrt an beliebigen Haltestellen und deren Kontext ermittelt werden. Zur Ermittlung der Verspätungen müssen die tatsächlichen Ankunftszeiten mit den fahrplanmäßigen Ankunftszeiten verglichen werden. Der Kontext der Verspätung beschreibt, in welchem Zustand die Strecke sich befindet, wenn die Verspätung eintritt, also beispielsweise ob gerade Streckenstörungen vorliegen. Durch den laufenden Betrieb kommen stetig Echtzeitdaten nach, welche den aktuellen Zustand des Netzes und alle aktuellen Verspätungen beschreiben. Werden diese Echtzeitdaten zu Verspätungen gesammelt, entsteht ein stetig wachsender historischer Datensatz, der Verspätungen und deren Kontext beschreibt. Für das gewünschte Vorhersagesystem kann also davon ausgegangen werden, dass ein stetig wachsender historischer Datensatz vorliegt und stetig Echtzeitdaten nachkommen.

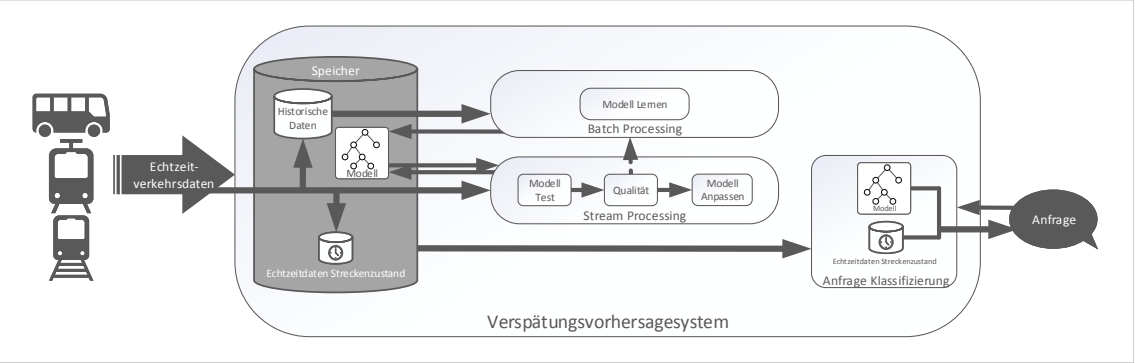



**Abbildung 5.1:** Konzeptioneller Entwurf der benötigten Architektur, entworfen anhand von *Hybrid Processing Architecture for Big Data (BRAID)*.

Wie in Kapitel 2 beschrieben, ist nun das Ziel, Verspätungen vorherzusagen. Hierbei sollen die historischen Daten genutzt werden, um ein Vorhersagemodell für Verspätungen zu erlernen. Dieses Modell soll anhand der neu eintreffenden Echtzeitdaten bezüglich Verspätungen auf seine Qualität überprüft und angepasst werden können. Mit diesem aktuellen Modell sollen Vorhersagen für Verspätungen von Fahrten getroffen werden können, wobei die Echtzeitdaten zum aktuellen Streckenzustand berücksichtigt werden sollen. Um die Verspätung vorherzusagen, muss definiert werden, welche Fahrt betrachtet werden soll. Dies ist über Angabe der Start- und Zielhaltestellen, der Linie der Fahrt und des Zeitpunkts der Fahrt möglich.

## 5.2 Architektur

Im Folgenden wird eine Architektur entwickelt, die es ermöglichen soll, diese Funktionalitäten auf den Daten zu erreichen.

Die Architektur braucht sowohl Batch-, als auch Streamverarbeitung. Die Batchverarbeitung wird benötigt, um möglichst effizient ein möglichst genaues Vorhersagemodell zu erlangen [CY15]. Die Streamverarbeitung hingegen wird für alles gebraucht, das in Echtzeit geschehen soll. Das heißt, sie wird zur Evaluation und Verbesserung des Modells anhand der Echtzeitdaten zu Verspätungen und zur Verarbeitung der eintreffenden Informationen zum Streckenzustand benötigt, wobei der Streckenzustand hier beispielsweise über Störungsmeldungen wie einer Weichenstörung angegeben sein kann.

Als Grundlage für die Architektur wurde deshalb *BRAID* gewählt, da hier die Batch- und Streamverarbeitung Daten austauschen und sich gegenseitig beeinflussen können.

Die entwickelte Architektur, zu sehen in Abbildung 5.1, erhält Echtzeitverkehrsdaten. Darin enthaltene Daten bezüglich Verspätungen und ihrem Kontext werden zum einen in einen permanenten Speicher mit historischen Verkehrsdaten abgelegt, zum anderen an den Streamverarbeitungszweig weitergegeben. Daten zum aktuellen Streckenzustand hingegen werden nur vorübergehend zwischengespeichert und mit neuen Daten überschrieben, sobald sie nicht mehr aktuell sind.

Die historischen Daten werden vom Batchverarbeitungszweig genutzt, um ein möglichst genaues Vorhersagemodell für Verspätungen zu erlernen. Dieses wird im gemeinsamen Speicher abgelegt. Der Streamverarbeitungszweig wiederum nimmt das Vorhersagemodell aus dem Speicher. Er nutzt die Echtzeitverspätungsdaten, um einerseits die Genauigkeit des Modells zu evaluieren und um es andererseits zu verbessern. Ist das Modell zu ungenau, veranlasst der Streamverarbeitungszweig, dass der Batchverarbeitungszweig ein neues Modell erlernt, welches nur neuere historische Verspätungsdaten berücksichtigt. Sobald dieses Modell fertig ist, wird das bestehende Modell im Speicher mit dem neuen überschrieben und der Streamverarbeitungszweig nutzt ab diesem Zeitpunkt das neue Modell.

Werden Anfragen an das System gestellt, werden das momentane Vorhersagemodell und, sofern die Anfrage nicht für eine Fahrt in weiterer Zukunft ist, die aktuellen Daten zum Streckenzustand aus dem Speicher geholt. Diese werden zusammen verwendet, um eine Verspätungsvorhersage zu treffen. Bezieht sich die Anfrage auf eine Fahrt in weiter entfernter Zukunft, so ist der aktuelle Streckenzustand nicht relevant und es wird nur anhand des Modells eine Vorhersage getroffen. In diesem Fall ist es möglich, dass in der Anfrage, analog zum Use-Case UC3, ein Streckenzustand definiert ist, welcher für die Vorhersage berücksichtigt wird.

*BRAID* nutzt normalerweise ein Result Layer, in dem Ergebnisse der beiden Verarbeitungszweige für die Beantwortung von Anfragen zusammengeführt werden. Der Batch- und der Streamverarbeitungszweig liefern ihre Ergebnisse an das Result Layer und für die zukünftige Verarbeitung relevante Teile wandern in den gemeinsamen Speicher zurück. Für diesen Anwendungsfall werden diese Ergebnisse jedoch nur indirekt zur Beantwortung der Anfragen genutzt. Gleichzeitig werden die gesamten Ergebnisse des Batchverarbeitungszweigs, also das Vorhersagemodell, für zukünftige Verarbeitungen des Streamzweiges benötigt. Deshalb wandert dieses direkt in den gemeinsamen Speicher. Bei den Ergebnissen der Streamverarbeitung ist es ähnlich. Die Evaluation des Modells wird nur benötigt, um eine Neuberechnung des Modells durch den Batchzweig auszulösen, falls diese nötig ist. Das verbesserte Modell hingegen wird für die weitere Streamverarbeitung benötigt, weshalb dieses direkt in den Speicher wandert. Die beiden Verarbeitungszweige haben ihre Ergebnisse hierdurch bereits kombiniert. Bei der Beantwortung von Anfragen werden die relevanten Ergebnisse und Informationen, mit anderen Worten das momentane Vorhersagemodell und die aktuellen Daten zum Streckenzustand, aus dem Speicher geholt und zur Beantwortung der Anfrage verwendet.

Zur Umsetzung der einzelnen Teile der Architektur bei der Implementierung können verschiedene Technologien verwendet werden. Zur Speicherung der historischen Daten können zum Beispiel verschiedene Datenbanksysteme oder verteilte Dateisysteme verwendet werden. Für die Implementierung des Batch- und des Streamverarbeitungszweigs können beispielsweise jeweils die Frameworks Flink oder Spark verwendet werden. Diese sind jedoch austauschbar und die Funktionalität des Gesamtsystems ist nicht an die Verwendung einer bestimmten Technologie gebunden.

## 5.3 Machine Learning Algorithmus

Für das System muss ein Vorhersagemodell für Verspätungen erstellt werden. Dies kann man mittels Machine Learning (ML) erreichen. In Abschnitt 4.4 wurden bereits verschiedene ML-Algorithmen vorgestellt, die mittels Batchverarbeitung erlernt werden können und die für einen Fall, bei dem man wie hier mehrere Klassen klassifizieren können muss, geeignet sind. Dies ermöglicht aber zunächst nur, dass das Klassifizierungsmodell im Batch berechnet und wenn es zu ungenau wird neu berechnet

werden kann. Dies ist optimal für den Fall geeignet, dass das alte Modell durch abrupte Änderungen der Gegebenheiten im Verkehrsnetz zu ungenau wird, aufgrund derer die älteren historischen Daten nicht mehr repräsentativ sind. Damit das Modell immer möglichst genau ist, ist es jedoch wünschenswert, dass das Modell zusätzlich durch neu eintreffende Echtzeitdaten weiter verfeinert werden kann. Somit kann das Modell zum einen durch die Nutzung einer größeren Datenmenge genauere Vorhersagen ermöglichen. Zum anderen kann es im Falle einer langsamen Veränderung der Gegebenheiten laufend mit angepasst werden. Dies wird durch sogenanntes inkrementelles oder auch online Lernen ermöglicht [DH00; SB17; SF86; Utg89].

Im Gegensatz zum nicht-inkrementellen Lernen werden beim inkrementellen Lernen nicht alle Daten gemeinsam betrachtet und ein für die vorliegenden Daten optimales Modell abgeleitet, welches danach nicht mehr angepasst werden kann [DH00; SB17; SF86; Utg89]. Stattdessen werden Datenobjekte einzeln betrachtet, sobald sie eintreffen. Jedes Objekt wird einzeln verarbeitet und genutzt, um das bestehende Modell abzuändern und zu verbessern. Hierbei können alle Daten gleich gewichtet werden. Es können aber auch, um Speicher zu sparen, alte Daten weniger stark gewichtet werden, als neue, damit alte Daten irgendwann vergessen werden können, was häufiger der Fall ist. Da auf diese Weise nicht die gesamten Daten gemeinsam betrachtet werden, benötigen inkrementelle Algorithmen meist größere Mengen an Daten, um denselben Grad an Genauigkeit zu erreichen, wie nicht-inkrementelle Algorithmen. Häufig wird deshalb zuerst ein Modell auf einem bestehenden historischen Datensatz erlernt und durch die eintreffenden neuen Daten nur verbessert, wie es in der vorgeschlagenen Architektur der Fall ist.

Es gibt inkrementelle Ansätze für verschiedene Algorithmen, darunter Entscheidungsbäume [DH00; SR14; Utg89], neuronale Netze [FT09; Pol+01] und Support Vector Machines [DC03]. Diese sind, wie in Abschnitt 4.4 bereits erläutert, ebenfalls für multiclass Klassifizierung geeignet. Somit könnte einer dieser Algorithmen, in einer gleichzeitig für mehrere Klassen und inkrementelles Lernen geeigneten Version, gut für den Anwendungsfall genutzt werden. Näheres zur Auswahl und Verwendung eines inkrementellen Algorithmus in der vorgeschlagenen Architektur ist in Abschnitt 6.3 zu finden.

# 6 Implementierung

Nachdem klar ist, wie solch ein Verspätungsvorhersagesystem aufgebaut sein sollte, wird nun erläutert, wie eine Umsetzung dieses Systems konkret aussehen kann. Wie bereits in Kapitel 5 werden hierfür zunächst in Abschnitt 6.1 die Daten erläutert, welche das System verarbeitet. In Abschnitt 6.2 wird eine konkrete Architektur vorgeschlagen, die die Erfüllung aller Anforderungen aus Kapitel 2 ermöglichen soll. Schließlich wird in Abschnitt 6.3 ein konkreter ML-Algorithmus vorgestellt, welcher für das System gut geeignet ist.

## 6.1 Nahverkehrsdaten

Seit Mai 2018 stellen einige deutsche Verkehrsverbünde, darunter auch der Verkehrs- und Tarifverbund Stuttgart (VVS), Daten zum öffentlicher Personennahverkehr (ÖPNV) in einem neuen *Open Data* Portal öffentlich zur Verfügung [Ver18b]. Dieses kann eine Datenquelle für die Sammlung und Auswertung historischer Verkehrsdaten darstellen. Zusätzlich können hierüber Störungsmeldungen empfangen sowie neue Verkehrsdaten abgefragt werden.

Das ÖPNV *Open Data* Portal hat zwei Teile, aus denen Daten bezogen werden können (im Falle des VVS zu finden unter [Ver18e]). Zum einen eher statische Datensätze, die sich nur selten ändern. Diese beschreiben das Netz, dessen Haltestellen, Linienverläufe, geplante Abfahrtszeiten und ähnliches [Ver18d]. Zum anderen eine Schnittstelle, über die verschiedene Informationen abgefragt werden können, wie Verbindungsauskünfte, Abfahrtstafeln an einzelnen Haltestellen oder Tarifauskünfte. Über diese kann man sich unter anderem über Störungen benachrichtigen lassen kann [Ver14; Ver18c]. Die Schnittstelle kann viele verschiedene Beschreibungen von Störungen liefern (mit Ursachen, die Umwelteinflüsse wie Wetterereignisse, verschiedene personelle Gegebenheiten, durch Betriebsmittel bedingte Ereignisse und andere Gründe einschließen, sowie verschiedenen Schweregraden der Störung) [Ver14, S. 59,114,116] [CEN11, S. 76–86]. Über die Informationen, die man bei der Schnittstelle anfragen kann, besteht auch die Möglichkeit, die tatsächliche Ankunftszeit an einer Haltestelle herauszufinden [Ver14, S. 76,126]. Es liegt also eine feste Beschreibung des Netzes und der geplanten Fahrten vor und über regelmäßige Anfragen können für alle stattgefundenen Fahrten die Verspätungen ermittelt werden. Zudem können, wann immer Störungen im Netz eintreten, Störungsmeldungen eintreffen.

In Form der Antworten auf die regelmäßigen Anfragen und der Störungsmeldungen liegen also als Stream stetig nachkommende Daten vor. Indem die eintreffenden Daten gespeichert werden, entsteht auch ein historischer Datensatz zu Verspätungen und ihrem Kontext.

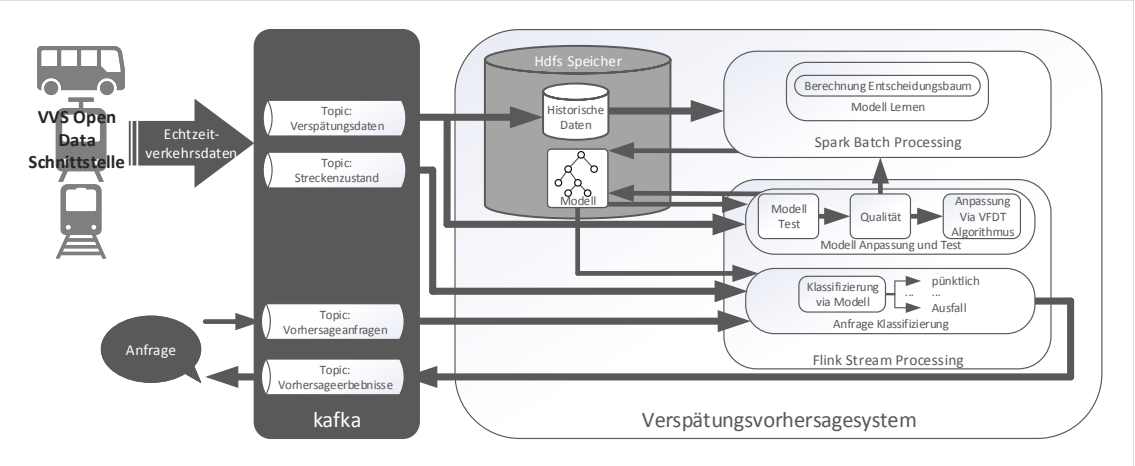



**Abbildung 6.1:** Architektur des Verspätungsvorhersagesystems.

## 6.2 Architektur

Wie bereits erläutert, könnten für die einzelnen Komponenten der Architektur verschiedene Technologien und Systeme verwendet werden. Im Folgenden wird beschrieben, wie das Vorhersagesystem konkret umgesetzt werden kann. Dies ist in Abbildung 6.1 noch einmal anschaulich dargestellt.

Für den Anwendungsfall sollen große Datenmengen verarbeitet werden können. Zudem ist es wünschenswert, dass das System skaliert werden kann, damit es auf größere Verkehrsnetze und damit größere Datenmengen angewandt werden kann. Eine Möglichkeit für den Speicher stellt deshalb das frei verfügbare Dateisystem Hadoop Distributed File System (HDFS) dar, da dieses auch verteilt und auf großen Datenmengen gut funktioniert sowie skalierbar ist.

Für die Implementierung des Batch- und des Streamverarbeitungszweigs können, wie in Abschnitt 4.3 bereits angesprochen, jeweils die Frameworks Flink oder Spark verwendet werden. Aufgrund seiner Herangehensweise an jegliche Verarbeitung als ein Fall einer Streamverarbeitung ist Flink besser für die Streamverarbeitung geeignet. Spark hingegen ist besser für die Batchverarbeitung geeignet, da es grundsätzlich Batchverarbeitung betreibt und die Streamverarbeitung nur über Nutzung von Micro-Batches löst. Beide unterstützen eine verteilte Verarbeitung und sind mit HDFS kompatibel, was diese Kombination ohne Middleware möglich macht. In beiden Fällen ist sowohl eine Implementierung in Scala als auch in Java möglich. Somit sind im Endeffekt beispielsweise beide Zweige in Scala implementiert.

Damit beide Zweige an dem Vorhersagemodell arbeiten können, muss das Modell in einer für beide Zweige, unabhängig vom Framework, nutzbaren Weise abgespeichert sein. Hierfür kann beispielsweise die Predictive Model Markup Language (PMML) [Gro+99] verwendet werden. Dies ist eine XML-ähnliche Sprache, die zur Definition und Speicherung von Vorhersagemodellen geeignet ist. Sie ermöglicht den Austausch von Modellen zwischen verschiedenen Systemen und ist gut für die Verwendung im verteilten Lernen geeignet.

Die Komponente, die zur Beantwortung der Anfragen zuständig ist, kann ebenfalls auf verschiedene Weisen umgesetzt werden. Eine davon ist, sie wie die Evaluation und Verbesserung des Vorhersagemodells als Streamverarbeitung umzusetzen. Somit kann dasselbe Framework wie für die Evaluation

des Modells verwendet werden. Es ist damit automatisch mit dem restlichen System kompatibel und diese Variante hat den Vorteil, dass die Anfragen als Echtzeitdaten verarbeitet werden und Antworten somit schnell verfügbar sind.

Zudem müssen die Echtzeitdaten zunächst vom Erzeuger, also von der Schnittstelle des VVS, ins System gelangen. Eine Möglichkeit hierfür stellt Apache Kafka dar. Kafka ist ein System, welches verteilt und skalierbar die Speicherung und Weiterleitung von Datenströmen ermöglicht. Die Datenerzeuger können Daten unter beliebigen sogenannten Topics in Kafka schreiben. Kafka speichert diese Daten (standardmäßig für sieben Tage, mit anderen Einstellungen können Daten auch unbegrenzt gespeichert werden). Zudem können sich sogenannte Consumer für ein oder mehrere Topics registrieren, woraufhin sie Daten zu diesem Topic von Kafka weitergeleitet bekommen, sobald diese in Kafka verfügbar sind. Die Consumer können zudem angeben, ab welchem Zeitpunkt sie die Daten direkt nach der Registrierung erhalten wollen. Hier gibt es unter anderem die Möglichkeiten, alle vom Zeitpunkt der Registrierung an neu eintreffenden Daten zu erhalten oder alle gespeicherten Daten zu erhalten. Kafka wäre gut für das System geeignet. Zum einen kann Kafka alle Echtzeitverkehrsdatendaten weiterleiten. Zum anderen kann es die Echtzeitdaten zum aktuellen Streckenzustand für einen definierten Zeitraum speichern, sodass bei der Klassifizierung immer alle relevanten Daten direkt von Kafka geholt werden können, ohne dass ein eigener temporären Speicher erstellen werden muss. Außerdem können Anfragen für Verspätungsvorhersagen ebenfalls mittels Kafka über ein eigenes Topic als Datenstrom an das System übermittelt werden. Die Ergebnisse können wiederum in ein weiteres Topic in Kafka geschrieben werden und von dort aus an die Nutzer weitergeleitet werden. Dies ermöglicht zudem, dass eine Nutzerschnittstelle entworfen werden kann, welche vom Vorhersagesystem völlig unabhängig ist.

Möchte ein Nutzer wie in Use-Case UC2 eine Vorhersage für eine aktuelle Fahrt, so wird die Anfrage in Kafka geschrieben. Die Anfrageverarbeitungskomponente erhält diese Anfrage und bezieht zusätzlich den aktuellen Streckenzustand aus Kafka. Das aktuelle Vorhersagemodell aus dem Speicher, welches via Batchverarbeitung erlernt und mittels des Streams von Echtzeitverspätungsdaten verbessert worden ist, wird daraufhin genutzt, um unter Berücksichtigung des aktuellen Streckenzustandes die Verspätung für die gewünschte Fahrt vorherzusagen. Das Ergebnis wird direkt ins entsprechende Kafka-Topic geschrieben und kann dem Nutzer übermittelt werden.

## 6.3 Machine Learning Algorithmus

Wie in Abschnitt 5.3 bereits erläutert, sind inkrementelle ML-Algorithmen, welche gleichzeitig multiclass Klassifizierung beherrschen, gut für das System geeignet. Ein solcher Algorithmus ist der Very Fast Decision Tree (VFDT)-Algorithmus [DH00; SR14]. Dies ist ein Algorithmus zur inkrementellen Erlernung eines Entscheidungsbaums. Da Entscheidungsbäume grundsätzlich für mehrere Klassen geeignet sind, gilt dies auch für anhand des VFDT-Algorithmus erlernte Entscheidungsbäume.

Beim VFDT-Algorithmus geschieht die inkrementelle Verbesserung des Baums mithilfe der Hoeffding Ungleichung [DH00; SR14]. Dies ist eine Ungleichung aus der Wahrscheinlichkeitstheorie, welche bestimmt, mit welcher maximalen Wahrscheinlichkeit eine Summe von Zufallsvariablen mehr als eine gewählte Konstante vom Erwartungswert abweicht. Eintreffende Daten werden zunächst in die Blätter des bestehenden Baums eingeordnet und dort gesammelt. Ist eine gewisse Minimalanzahl an Beispielen bei einem Blatt eingeordnet, wird nach jeder Einordnung an diesem

Blatt mittels einer Schranke basierend auf der Hoeffding Ungleichung geprüft, ob die an diesem Blatt eingeordneten Daten Anlass geben, dass an dieser Stelle gesplittet werden muss. Ist dies der Fall, so wird das Attribut bestimmt, welches an dieser Stelle die größte Aussagekraft besitzt. Der Blattknoten wird in einen Entscheidungsknoten umgewandelt, welcher anhand des gewählten Attributs Daten in einen von zwei Blattknoten, welche neu angelegt werden, einordnet.

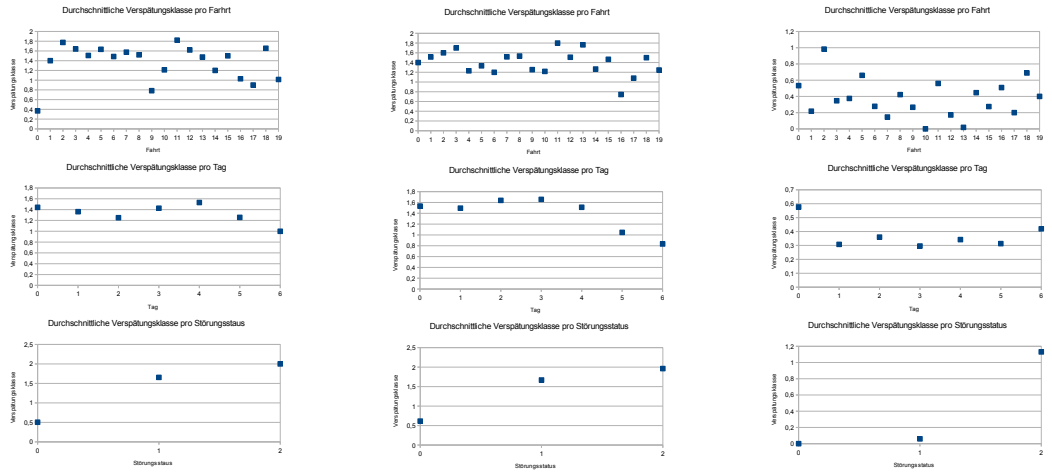
Bei der Nutzung dieses Algorithmus kann ein initialer Baum auf einem historischen Datensatz im Batch berechnet werden. Auf diese Art wird mit derselben Anzahl an Trainingsbeispielen eine größere initiale Genauigkeit erreicht, als beim reinen inkrementellen Lernen mit derselben Zahl an Beispielen. Somit kann zunächst ein Entscheidungsbaum vom Batchverarbeitungszweig mittels eines herkömmlichen Algorithmus erlernt werden. Im Streamverarbeitungszweig kann dieses Modell anhand des VFDT-Algorithmus verbessert werden. Studien haben gezeigt, dass der VFDT-Algorithmus schnelle Rechenzeiten sowie einen schnellen Anstieg der Genauigkeit durch neue Datenbeispiele ermöglicht [DH00; SR14]. Er ist somit gut geeignet, um die neu eintreffenden Echtzeitdaten in diesem System zur Verbesserung der Vorhersagegenauigkeit zu nutzen.

# 7 Prototyp

Eine vereinfachte Version der zuvor beschriebenen Architektur ist prototypisch implementiert worden. Diese wird im Folgenden beschrieben. Hierbei werden, wie in den beiden vorherigen Kapiteln, zunächst in Abschnitt 7.1 die verwendeten Daten erläutert. Daraufhin wird in Abschnitt 7.2 die umgesetzte Architektur beschrieben. In Abschnitt 7.3 wird schließlich erläutert, welcher ML-Algorithmus zur Verwendung im implementierten Prototypen gewählt ist und weshalb die Wahl auf diesen fällt.

| id | line | trip | stop | weekDay | status | delay | delayClass |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 8 | 1 | 2 | 2 | 28 | 2 |
| 1 | 2 | 8 | 2 | 2 | 2 | 39 | 2 |
| 2 | 2 | 8 | 3 | 2 | 2 | 51 | 2 |
| 3 | 2 | 8 | 4 | 2 | 2 | 58 | 2 |
| 4 | 2 | 8 | 5 | 2 | 2 | 65 | 2 |
| 5 | 2 | 18 | 1 | 1 | 0 | 0 | 0 |
| 6 | 2 | 18 | 2 | 1 | 0 | 0 | 0 |
| 7 | 2 | 18 | 3 | 1 | 0 | 0 | 0 |
| 8 | 2 | 18 | 4 | 1 | 0 | 0 | 0 |
| 9 | 2 | 18 | 5 | 1 | 0 | 0 | 0 |
| 10 | 3 | 15 | 1 | 0 | 1 | 11 | 1 |
| 11 | 3 | 15 | 2 | 0 | 1 | 14 | 1 |
| 12 | 3 | 15 | 3 | 0 | 1 | 22 | 2 |
| 13 | 3 | 15 | 4 | 0 | 1 | 30 | 2 |
| 14 | 3 | 15 | 5 | 0 | 1 | 28 | 2 |
| 15 | 3 | 15 | 1 | 1 | 0 | 1 | 0 |
| 16 | 3 | 15 | 2 | 1 | 0 | 0 | 0 |
| 17 | 3 | 15 | 3 | 1 | 0 | 0 | 0 |
| 18 | 3 | 15 | 4 | 1 | 0 | 0 | 0 |
| 19 | 3 | 15 | 5 | 1 | 0 | 2 | 0 |

**Tabelle 7.1:** Auszug aus dem Echtzeitdatensatz mit demselben Verspätungskonzept, wie beim historischen Datensatz.




**(a)** Durchschnittliche Verspätungen für den historischen Datensatz.

**(b)** Durchschnittliche Verspätungen für den ersten Echtzeitdatensatz.

**(c)** Durchschnittliche Verspätungen für den zweiten Echtzeitdatensatz.

**Abbildung 7.1:** Durchschnittliche Verspätungen der verschiedenen Datensätze.

## 7.1 Nahverkehrsdaten

Um das System ohne die aufwändige Anbindung an die Datenschnittstelle des Verkehrsunternehmens testen zu können, wurden drei Datensätze erstellt. Diese sind csv-Dateien mit je 1000 Einträgen. Einer wurde als historischer Datensatz erstellt, der zu Beginn vorliegt und einer für die Simulation von Echtzeitdaten. Bei diesen Datensätzen liegt den Verspätungen dasselbe Konzept zugrunde. Der letzte Datensatz soll Echtzeitdaten simulieren, die entstehen, wenn sich die Gegebenheiten ändern und den Verspätungen ein anderes Konzept zugrunde liegt. Alle Datensätze sind hierbei am Szenario aus Abschnitt 2.1 orientiert. In den drei Datensätzen ist für jeden Datenpunkt eine ID (id), die S-Bahnlinie (line), die Fahrt (trip), die Haltestelle (stop), der Wochentag (weekDay), der Störungsstatus (status), die aktuelle Verspätung in Minuten (delay) sowie die zugeordnete Verspätungsklasse (delayClass) vermerkt. Ein Ausschnitt aus einem der Datensätze ist in Tabelle 7.1 zu sehen.

Für die Linie sind die Werte 2 und 3 für die Linien S2 und S3 möglich. Die Fahrt kann Werte von 0 bis 19 haben, was für einen vereinfachten Fahrplan mit einer Fahrt pro Stunde von 5 Uhr morgens (Fahrt 0) bis 0 Uhr nachts (Fahrt 19) stehen soll. Die Haltestelle gibt die Zielhaltestelle an, wobei fünf Werte für die Haltestellen Universität (1), Schwabstraße (2), Stadtmitte (3), Hauptbahnhof (4) und Bad Cannstatt (5) möglich sind. Der Wochentag ist als Ganzzahl angegeben, beginnend bei 0 für Montag, bis zur 6 für Sonntag. Der Störungsstatus soll einen vereinfachten aktuellen Streckenzustand darstellen, wobei drei Werte möglich sind. Für den Fall, dass alles normal ist 0, für leichte Störungen 1 und für starke Störungen 2. Die aktuelle Verspätung stellt die zusätzliche Verspätung dar, die beginnend bei der Starthaltestelle Flughafen bis zur jeweiligen Zielhaltestelle erreicht wird. Die Verspätungsklassen sind abhängig von der aktuellen Verspätung zugeteilt. Wie

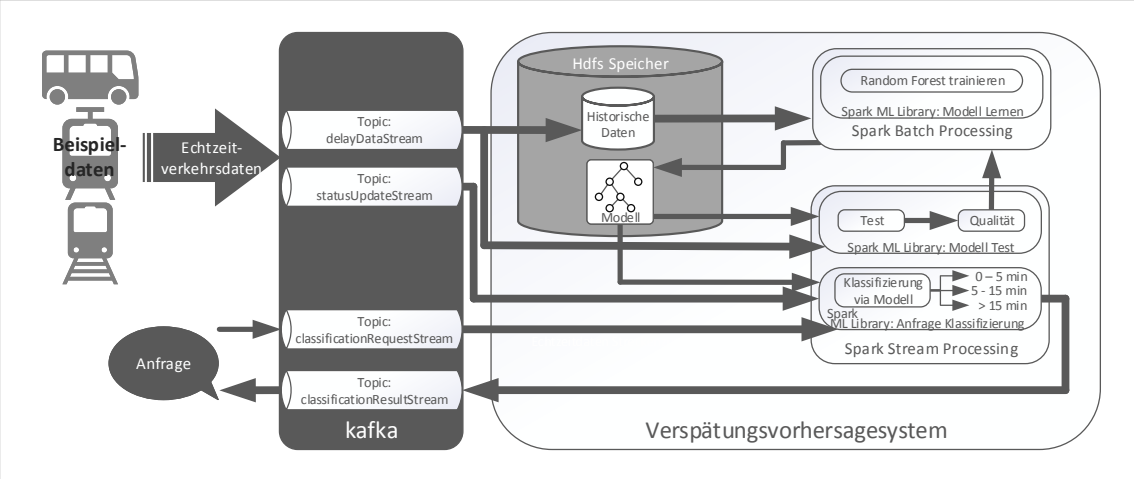



**Abbildung 7.2:** Architektur des implementierten Prototyps.

bei der Aufteilung in Abbildung 4.5a wird Verspätungsklasse 0 bei bis zu 5 Minuten Verspätung zugeordnet, Klasse 1 bei mehr als 5 und bis zu 15 Minuten und Klasse 2 bei mehr als 15 Minuten Verspätung.

Die Daten sind jeweils teilweise zufällig generiert. Hierbei kann die Verspätung bei einer Fahrt auf jeder Teilstrecke größer werden oder sich leicht verringern. Die ersten beiden Datensätze haben grundsätzlich höhere Verspätungen als der dritte. Zudem hängt die mögliche Größe der Zusatzverspätung beim dritten Datensatz nur vom Störungsstatus ab. Bei den beiden anderen Datensätzen können zudem Zusatzverspätungen für Werktage (Tage 0 bis 4), sowie für die Hauptverkehrszeiten an diesen Werktagen (Fahrten 1 bis 3 und 11 bis 13 für die Zeiten von 6 Uhr bis 8 Uhr sowie von 16 Uhr bis 18 Uhr) entstehen. In Abbildung 7.1 sind durchschnittliche Verspätungen für alle drei Datensätze zu sehen.

## 7.2 Architektur

Im Folgenden wird auf die prototypische Implementierung des Systems eingegangen. Diese ist in Abbildung 7.2 veranschaulicht.

Als Speicher wird HDFS verwendet. Die historischen Daten sind dort im csv-Format abgespeichert.

Für die Implementierung des Systems wird Scala in der Version 2.11 und das Spark Framework in der Version 2.3.2 verwendet. Spark wird hier sowohl für die Implementierung des Batchverarbeitungszweigs als auch für die Implementierung des Streamverarbeitungszweigs verwendet. Die Nutzung eines Frameworks für beide Zweige wird zum einen gewählt, da auf diese Art die vor-implementierten ML-Algorithmen der Library des Frameworks verwendet werden können. Mehr dazu und zur Auswahl des Algorithmus ist in Abschnitt 7.3 zu finden. Zum anderen besitzen die Frameworks eigene Darstellungen der Modelle. Die Modelle können mithilfe des Frameworks direkt abgespeichert und geladen werden. Hierfür wird weder die Verwendung einer Middleware noch die Übersetzung des Modells in ein externes Datenformat benötigt. Die Wahl fällt auf Spark und nicht

**Listing 7.1** Scala-Code zum Erlernen eines Klassifizierungsmodells mittels der Machine Learning Library von Spark via Batch Processing.

```
...
// scaler for features
val scaler = new StandardScaler()
  .setInputCol("unscaledFeatures")
  .setOutputCol("features")
  ...
...

val Array(train, test) = data.randomSplit(Array(0.8, 0.2))

// instantiate the random forest classifier
val randForClassifier = new RandomForestClassifier()
  .setLabelCol("label")
  .setFeaturesCol("features")
  .setNumTrees(10)

val pipeline = new Pipeline()
  .setStages(Array(scaler, randForClassifier))

// train the multiclass model
val model = pipeline.fit(train)

// predict class of test data for accuracy test
val predictions = model.transform(test)

// obtain evaluator
val evaluator = new MulticlassClassificationEvaluator()
  .setMetricName("accuracy")

// compute the accuracy of predictions on test data
val accuracy = evaluator.evaluate(predictions)
println(s"RandForest Accuracy = $accuracy")

// save trained model
model.write.overwrite().save(modelLocation)
...
```

auf Flink, da Spark im Gegensatz zu Flink zum einen eine größere Auswahl an ML-Algorithmen bietet, und da zum anderen auch Algorithmen enthalten sind, die für multiclass Klassifizierung geeignet sind.

Der Batchzweig ist als Scala-Object implementiert. Die historischen Daten werden zunächst eingelesen, wobei bei einem normalen Aufruf der gesamte historische Datensatz verwendet wird. Geschieht der Aufruf unter Mitgabe eines entsprechenden Parameters, werden nur neuere Daten geladen, genauer gesagt die Daten, die innerhalb der letzten zwei Tage hinzugefügt wurden. Diese Daten werden verwendet, um mithilfe der Spark ML-Library als Batchverarbeitung ein neues Modell zu erlernen. Ein Auszug aus dem benötigten Code ist in Listing 7.1 zu sehen.

**Listing 7.2** Definition eines Spark Streams zum Testen des Klassifizierungsmodells. Die Echtzeitdaten für den Test kommen als Stream von einem Kafka-Topic. Sie werden klassifiziert und das Ergebnis anhand einer eigens definierten Funktion evaluiert.

```
    ...
    // load model
    model = PipelineModel.load(modelLocation)
    ...

    // define function for evaluating the predictions
    val eval = defineEvalFuntion()

    // receive testing data from kafka stream
    val df = spark
      .readStream
      .format("kafka")
      .option("subscribe", "delayDataStream")
      ...

    // select relevant columns
    val data = df
      .withColumn("data", from_json(df.col("value"), dataSchema))
      ...

    // prepare data for the classification model
    val testingData = vecAssembl.transform(data).select("label", "
unscaledFeatures", "timestamp")

    // predict labels for the test data
    val predictions = model.transform(testingData)

    // evaluate according to the function
    val streamingQuery = predictions
      .writeStream
      .foreach(eval)
      .start()
    ...
```

Zum Simulieren der Echtzeitdaten liest eine separate Spark-Anwendung die Daten aus den entsprechenden csv-Dateien ein, wandelt sie in einen Datenstrom um und schreibt sie unter dem Topic „delayDataStream“ in Kafka. Zudem werden neue Störungsmeldungen unter dem Topic „statusUpdateStream“ in Kafka geschrieben. Es wird ein Kafka Server mit Kafka Version 2.0.0 verwendet.

Der Streamzweig enthält zwei Teile. Zum einen den Modelltest, zum anderen die Klassifizierung der Anfragen. Auf die Anpassung des bestehenden Modells durch Echtzeitdaten wird verzichtet (mehr hierzu ist in Abschnitt 7.3 zu finden).

Der Modelltest ist ebenfalls als Scala-Object implementiert. Da hier eine Streamverarbeitung gewünscht ist, muss zunächst ein Spark *StreamingQuery* definiert und dieses gestartet werden. Ein Auszug des hierfür benötigten Codes ist in Listing 7.2 zu sehen. Hierfür wird das Vorhersagemodell

geladen und die Operationen, welche auf dem Stream auszuführen sind, definiert. Im *StreamingQuery* werden die Daten zunächst aus dem Kafka-Topic „delayDataStream“ eingelesen und in das richtige Format umgewandelt. Daraufhin werden die Daten mithilfe des Modells klassifiziert. Die Klassifizierung wird ausgewertet, um die Genauigkeit des Modells zu bestimmen. Für die Berechnung der Genauigkeit können Datenpunkte innerhalb eines Micro-Batches auf einen gemeinsamen Arbeitsspeicher zugreifen. Diese werden gleich gewichtet für die Berechnung der Genauigkeit verwendet. Existieren bereits vorherige Genauigkeitsberechnungen, liegen die Ergebnisse derer im permanenten Speicher vor und werden aufgrund der größeren Menge an Daten, die darauf Einfluss hatten, zu zwei Dritteln gewichtet in die neue Genauigkeitsberechnung miteinbezogen. Nachdem insgesamt mindestens fünfundzwanzig Datenpunkte für die Genauigkeitsberechnung betrachtet wurden, um einen repräsentativen Wert zu erhalten, wird geprüft, ob die Genauigkeit einen Schwellenwert unterschreitet. Ist dies der Fall, wird die Berechnung eines neuen Modells anhand der neueren Daten durch den Aufruf des Modell Erlerners mit entsprechendem Parameter veranlasst. Sobald das Modell fertig berechnet ist, wird das neue Modell geladen und ein *StreamingQuery* für den Modelltest für das neue Modell definiert. Der neue Query wird gestartet, der alte Query gestoppt und die alten Daten bezüglich der Genauigkeit des Modells gelöscht. Dies ermöglicht die Anpassung des Modells an neue Gegebenheiten, um auch im Fall, dass sich die Gegebenheiten im Netz ändern, wie es im Use-Case UC4 der Fall ist, korrekte Vorhersagen treffen zu können. Parallel dazu läuft durchgehend ein *StreamingQuery*, welches die eintreffenden Daten aus dem Kafka-Topic „delayDataStream“ einliest und diese mitsamt des Zeitstempels aus Kafka im historischen Datenspeicher ablegt.

Bei der Klassifizierung der Anfragen laufen ebenfalls immer zwei Streaming Queries parallel. Der erste *StreamingQuery* ist dafür verantwortlich, die Störungsmeldungen aus dem Topic „statusUpdateStream“ in Kafka zu empfangen. Sobald ein neuer Störungszustand eintrifft, wird der andere *StreamingQuery* dahingehend neu definiert, dass der neue Zustand verwendet wird und gestartet. Der zweite *StreamingQuery* ist für die Klassifizierung der Anfragen zuständig. Da er hierzu ebenfalls das gespeicherte Modell verwendet, wird auch dieser Query mit dem neu geladenen Modell neu definiert, wann immer ein neues Vorhersagemodell erstellt worden ist. Der neue Query wird daraufhin gestartet und der alte gestoppt. Klassifizierungsanfragen werden unter dem Topic „classificationRequestStream“ in Kafka geschrieben. Der Klassifizierungsquery erhält diese Anfragen aus Kafka und klassifiziert sie mithilfe des Vorhersagemodells. Hierbei wird der aktuelle Störungszustand verwendet, sofern kein anderer in der Anfrage angegeben ist. Enthält die Anfrage einen eigenen Störungszustand, wird dieser verwendet. Auf diese Weise können bei Verspätungsvorhersagen sowohl, wie es in Use-Case UC3 der Fall ist, vom Nutzer angegebene Störungen als auch, wie in Use-Case UC2, die aktuellen Störungsmeldungen aus dem Netz verwendet werden. Ist ein Ergebnis der Klassifizierung vorhanden, wird das Ergebnis in ein weiteres Kafka-Topic, „classificationResultStream“, geschrieben. Von dort aus können Nutzer die Ergebnisse ihrer Anfragen erhalten. Einen Auszug aus der Definition des Klassifizierungsqueries ist in Listing 7.3 zu finden.

Die Implementierung für den Streamzweig sowie für den Batchzweig befinden sich in einer gemeinsamen Anwendung.

**Listing 7.3** Definition eines Spark Streams für die Klassifizierung von Anfragen. Die Anfragen kommen als Stream von einem Kafka-Topic und werden nach der Klassifizierung wieder in ein anderes Kafka-Topic geschrieben. Ist kein Status angegeben, wird der momentane Status des Verkehrsnetzes verwendet.

```
    ...
    // receive classification requests from kafka
    val df = spark
      .readStream
      .format("kafka")
      .option("subscribe", "classificationRequestStream")
      ...

    // select relevant columns and use current status of public transport network
 where none was defined
    val data = df.withColumn("data", from_json(df.col("value"), dataSchema))
      ...
      .na.fill(currentStatus, Array("status"))
      ...

    // prepare data for prediction
    val predictingData = vecAssembl.transform(data).select("id", "
unscaledFeatures", "timestamp")

    // predict delay class
    val predictions = model.transform(predictingData)
      .select("timestamp", "id", "unscaledFeatures", "prediction")

    // write prediction results to kafka
    val streamingQuery = predictions
      ...
      .writeStream
      .format("kafka")
      .option("topic", "classificationResultStream")
      ...
    ...
```

## 7.3 Machine Learning Algorithmus

Zusätzlich benötigt der Prototyp einen ML-Algorithmus, um das Vorhersagemodell erstellen zu können. Es wird hierfür die Spark ML Library[1] verwendet.

Es gibt zwei Versionen der Library, eine ältere Version der ML Library[2], welche nicht mehr weiter verbessert wird und eine aktuelle Version der ML Library[1]. Die alte Version unterstützt vier ML-Algorithmen, die für multiclass Klassifizierung geeignet sind. Dies sind Logistische Regression, Entscheidungsbäume, Random Forests (ein ML-Verfahren welches auf der Nutzung

[1] siehe: https://spark.apache.org/docs/latest/ml-guide.html
[2] siehe: https://spark.apache.org/docs/latest/mllib-guide.html

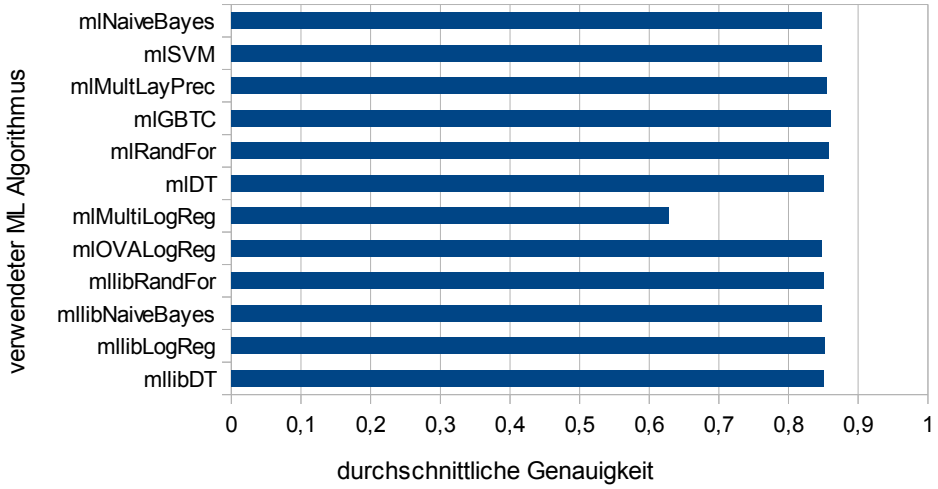



**Abbildung 7.3:** Ergebnisse der Genauigkeitstests der verschiedenen Algorithmen der Spark ML-Library.

mehrerer Entscheidungsbäume beruht) und Naive Bayes. Die aktuelle Version der Library unterstützt sechs ML-Algorithmen, die für multiclass Klassifizierung geeignet sind. Diese sind Multinomiale Logistische Regression, Entscheidungsbäume, Random Forests, Naive Bayes, Gradient-Boosted Trees (ebenfalls ein ML-Verfahren welches auf der Nutzung mehrerer Entscheidungsbäume beruht) sowie mehrschichtige neuronale Feedforward-Netze. Zusätzlich bietet diese Version der Library einen One-versus-all (OVA)-Algorithmus, mithilfe dessen zusätzlich Support Vector Machines sowie Binomiale Logistische Regression genutzt werden können.

Keine der beiden Versionen der Library enthalten multiclass Klassifizierungsalgorithmen, welche auch inkrementelles Lernen beherrschen. Daher ist dies im Prototypen nicht enthalten. Eintreffende Echtzeitdaten werden also nur zum Test der Genauigkeit des Klassifizierungsmodells verwendet, nicht um dieses zu verbessern.

Trotzdem muss einer der verfügbaren Algorithmen für die Implementierung des Prototyps gewählt werden. Aus Abschnitt 4.4 ist bekannt, dass bei der Nutzung von multiclass Klassifizierungsalgorithmen eine leichte Tendenz zu Support Vector Machines als Beste bezüglich Genauigkeit besteht. Allerdings ist die optimale Reduktionsvariante stark problemabhängig. Da über die ML-Library nur eine Reduktionsvariante verfügbar ist, ist also unklar, ob diese für den Anwendungsfall die beste Lösung ist. Um zu überprüfen, ob die Support Vector Machine mit OVA-Reduktion für den bestehenden Anwendungsfall tatsächlich am besten geeignet ist, oder ob nicht einer der anderen verfügbaren Algorithmen besser geeignet ist, sind Tests durchgeführt worden.

Hierbei sind die historischen Daten in einen Trainings- (80%) und einen Testdatensatz (20%) aufgeteilt worden. Daraufhin sind alle Algorithmen der alten Version der Library sowie alle Algorithmen der neuen Version genutzt worden, um Klassifizierungsmodelle anhand der Trainingsdaten zu

erlernen. Für die Algorithmen sind jeweils die Standardparameter verwendet worden. Danach sind alle Modelle anhand des Testdatensatzes getestet und die Genauigkeit jedes Modells berechnet worden. Dies wurde zweihundert mal wiederholt und die Ergebnisse gesammelt.

Wie in Abbildung 7.3 zu sehen, liegen die durchschnittlichen Genauigkeitswerte der verschiedenen Algorithmen fast alle sehr nah beieinander. Nur die Multinomiale Logistische Regression hat deutlich schlechter abgeschnitten, als die restlichen Algorithmen. Die zwei besten Algorithmen sind Random Forests und Gradient-Boosted Trees, beides mal aus der neuen Version der Library, mit einem knappen Abstand von etwa 0,3% weniger Fehlern untereinander und etwa 0,2% weniger Fehlern im Vergleich zum drittbesten Algorithmus.

Da die Unterschiede nur sehr gering sind, ist es egal, welcher der Algorithmen verwendet wird. Es wurde letztendlich der Random Forest Algorihtmus aus der neuen Libraryversion gewählt. Wie das Trainieren des Vorhersagemodells und der Genauigkeitstest mittels der ML-Library implementiert worden sind, ist in Listing 7.1 zu sehen. In Listing 7.4 ist ein Teil eines Random Forest Modells zu sehen, welches aus den historischen Daten erlernt wurde. Dies ist einer von zehn gleichgewichteten Entscheidungsbäumen, aus denen das Modell besteht.

**Listing 7.4** Auszug der textuellen Beschreibung eines Entscheidungsbaums, der Teil eines Random Forest Modells ist, wie sie von Spark ausgegeben wird.

```
If (feature 4 <= 0.6308794341137638)
 If (feature 2 <= 2.4864471459601365)
  If (feature 3 <= 2.25266397311339)
   If (feature 1 <= 2.2898047689525125)
    If (feature 1 <= 0.08480758403527823)
     Predict: 0.0
    Else (feature 1 > 0.08480758403527823)
     Predict: 1.0
   Else (feature 1 > 2.2898047689525125)
    Predict: 0.0
  Else (feature 3 > 2.25266397311339)
   If (feature 2 <= 1.7760336756858117)
    If (feature 1 <= 0.5936530882469476)
     Predict: 0.0
    Else (feature 1 > 0.5936530882469476)
     Predict: 0.0
   Else (feature 2 > 1.7760336756858117)
    If (feature 1 <= 0.5936530882469476)
     Predict: 0.0
    Else (feature 1 > 0.5936530882469476)
     Predict: 0.0
 Else (feature 2 > 2.4864471459601365)
  If (feature 1 <= 2.2898047689525125)
   If (feature 3 <= 2.25266397311339)
    If (feature 3 <= 1.2514799850629943)
     Predict: 2.0
    Else (feature 3 > 1.2514799850629943)
     Predict: 2.0
   Else (feature 3 > 2.25266397311339)
    If (feature 0 <= 5.006909607646602)
     Predict: 0.0
    Else (feature 0 > 5.006909607646602)
     Predict: 0.0
  Else (feature 1 > 2.2898047689525125)
   Predict: 0.0
Else (feature 4 > 0.6308794341137638)
 If (feature 4 <= 1.8926383023412914)
  If (feature 2 <= 1.065620205411487)
   If (feature 0 <= 5.006909607646602)
    Predict: 1.0
   Else (feature 0 > 5.006909607646602)
    If (feature 1 <= 1.2721137605291735)
     Predict: 1.0
    Else (feature 1 > 1.2721137605291735)
     Predict: 2.0
  ...
```

# 8 Evaluation

Im Folgenden wird bewertet, inwiefern zum einen das vorgeschlagene System aus Kapitel 6 und zum anderen der implementierte Prototyp, welcher in Kapitel 7 beschrieben wird, die Anforderungen aus Abschnitt 2.3 erfüllen. Für das vorgeschlagene Vorhersagesystem wird davon ausgegangen, dass die Abläufe, wie in Abschnitt 5.2 beschrieben, umgesetzt werden und alle in Kapitel 6 vorgeschlagenen Techniken und Systeme der Empfehlung entsprechend verwendet werden. Es wird für jeden Anwendungsfall einzeln betrachtet, ob er von den beiden Vorhersagesystemen erfüllt wird. Die Ergebnisse dieser Evaluation sind zusammengefasst in Tabelle 8.1 zu sehen.

**Anforderung A1** fordert, dass das System ein möglichst genaues Vorhersagemodell für Verspätungen auf historischen Daten erlernt. Sowohl das vorgeschlagene System aus Kapitel 6 als auch der Prototyp besitzen einen Speicher, der historische Verspätungsdaten enthält. Beide nutzen Batchverarbeitung, um ein Vorhersagemodell für Verspätungen auf den gespeicherten historischen Daten zu erlernen. Das vorgeschlagene System erstellt hierbei einen Entscheidungsbaum, während der Prototyp einen Random Forest Klassifizierer erstellt. In beiden Fällen wird ein Vorhersagemodell unter Berücksichtigung aller vorliegender historischer Daten erlernt. Dass dieses Modell möglichst genau ist, wird beim vorgeschlagenen System sichergestellt, indem das Modell unter Verwendung des VFDT-Algorithmus anhand von Echtzeitdaten nachträglich verbessert wird. Beim Prototyp ist dies durch die Auswahl eines Algorithmus gewährleistet, der sich durch Vergleichstests als besonders genau herausgestellt hat. Somit haben sowohl das vorgeschlagene System als auch der Prototyp Anforderung A1 erfüllt.

**Anforderung A2** fordert, dass das System Verspätungen vorhersagen kann. Bei diesen Vorhersagen soll es möglich sein, Echtzeitdaten zum aktuellen Streckenzustand, beispielsweise in Form von Störungsmeldungen, zu berücksichtigen. Das vorgeschlagene System löst dies, indem es den berechneten Entscheidungsbaum nutzt, um aus Kafka eintreffende Anfragen zu klassifizieren. Den aktuellen Streckenzustand erhält es hierbei in Form von Echtzeitdaten aus Kafka und bezieht diese bei der Klassifizierung mit ein. Das Ergebnis der Klassifizierung stellt die Verspätungsvorhersage dar. Das vorgeschlagene System erfüllt diese Anforderung somit.

Der Prototyp nutzt den erstellten Random Forest Klassifizierer zur Klassifizierung. Wann immer kein anderweitiger Zustand in der aus Kafka empfangenen Anfrage definiert ist, wird der aktuelle Streckenzustand aus der letzten Störungsmeldung genutzt. Die Störungsmeldungen werden als Echtzeitdaten aus Kafka empfangen und sobald sie eintreffen zur Anpassung des Klassifizierungs Streaming Queries genutzt, wodurch immer die neuste Störungsmeldung verwendet wird. Die Ergebnisse dieser Klassifizierungen stellen die Verspätungsvorhersagen dar. Auch der Prototyp hat somit Anforderung A2 erfüllt.

**Anforderung A3** fordert eine Vorhersage wie in Anforderung A2, mit dem Unterschied, dass nicht der aktuelle Streckenzustand, sondern ein vom Nutzer selbst definierter berücksichtigt werden soll. Im vorgeschlagenen System kann dies gelöst werden, indem die aus Kafka empfangenen Anfragen eigene Streckenzustände angeben dürfen. Diese können dann in einer Klassifizierung, analog zu der

unter der Evaluation von Anforderung A2 beschriebenen, statt des aktuellen Streckenzustandes, verwendet werden. Damit entsteht eine Verspätungsvorhersage unter Berücksichtigung eines vom Nutzer selbst definierten Streckenzustandes. Das vorgeschlagene System kann Anforderung A3 somit erfüllen.

Im Prototyp kann bei Anfragen ein Störungszustand mit angegeben werden. Ist dieser in der Anfrage angegeben, läuft die Klassifizierung analog zu der Klassifizierung, wie sie für Anforderung A2 beschrieben wurde, ab. Anstelle der aktuellen Störungsmeldung wird jedoch der angegebene Störungszustand berücksichtigt. Somit kann eine Vorhersage unter Berücksichtigung eines vom Nutzer definierten Streckenzustandes getroffen werden und A3 ist erfüllt.

**Anforderung A4** betrachtet den Fall, dass aufgrund einer abrupten Änderung im Netz die Vorhersagequalität des verwendeten Modells schnell stark abnimmt. Für solch einen Fall wird gefordert, dass die Genauigkeit des Modells anhand der eintreffenden Echtzeitdaten getestet wird. Wird das Modell zu ungenau, soll dies erkannt und ein neues Modell berechnet werden, wobei nur die nun relevanten neueren Daten genutzt werden.

Das vorgeschlagene System erhält aus Kafka Echtzeitdaten zu Verspätungen. Diese werden zum einen gespeichert, sodass die neuesten Daten immer im Speicher vorliegen und genutzt werden können. Zum anderen werden sie an eine Streamverarbeitungskomponente weitergegeben. Diese evaluiert die Genauigkeit des Modells anhand der eintreffenden Daten. Ist das Modell zu ungenau, wird veranlasst, dass vom Batchzweig ein neues Modell erlernt wird, wobei nur die aktuelleren Daten aus dem historischen Datensatz berücksichtigt werden. Damit kann das vorgeschlagene System Anforderung A4 erfüllen.

Auch der Prototyp erhält Echtzeitdaten zu Verspätungen aus Kafka. Diese werden von einem *StreamingQuery* erhalten und zusammen mit ihrem Zeitstempel aus Kafka im historischen Datensatz im Speicher abgelegt. Ein weiterer *StreamingQuery* klassifiziert parallel dazu die eintreffenden Daten anhand des aktuellen Modells und nutzt die Ergebnisse um die Genauigkeit des Modells zu ermitteln. Sinkt diese unter einen Schwellenwert, wird die Berechnung eines neuen Modells veranlasst. Für die Berechnung werden nur neuere historische Daten verwendet, welche anhand ihres Zeitstempels herausgefiltert werden können. Somit wird Anforderung A4 vom Prototyp erfüllt.

**Anforderung A5** betrachtet im Gegensatz zu Anforderung A4 keine abrupten starken Änderungen im Netz, sondern leichte langsame aber stetige Änderungen. Aufgrund solcher Änderungen kann ein Modell langsam ungenauer werden. Um dies zu verhindern, fordert Anforderung A5, dass das Vorhersagemodell beim Eintreffen von Echtzeitdaten zu Verspätungen durch diese angepasst wird.

Im vorgeschlagenen System wird dies über die Nutzung des VFDT-Algorithmus gelöst. Die zuvor beschriebene Streamverarbeitungskomponente nutzt die Echtzeitdaten nach der Evaluation der Genauigkeit des Entscheidungsbaums, um diesen anhand des VFDT-Algorithmus durch diese Daten anzupassen. Das Modell kann somit an kleine und langsame Änderungen angepasst werden und Anforderung A5 ist erfüllt.

Der Prototyp hingegen verwendet keinen inkrementellen ML-Algorithmus. Das Vorhesagemodell kann deshalb nicht angepasst werden und Anforderung A5 ist nicht erfüllt. Auf die Verwendung des inkrementellen ML-Algorithmus wurde bei der Implementierung des Prototypen zwar verzichtet, dies könnte in Zukunft allerdings noch implementiert werden. Durch diese Ergänzung wäre die Erfüllung von Anforderung A5 möglich.

| | A1 | A2 | A3 | A4 | A5 | A6 |
|---|---|---|---|---|---|---|
| **Implementierungsvorschlag aus Kapitel 6** | x | x | x | x | x | x |
| **Implementierter Prototyp aus Kapitel 7** | x | x | x | x | | x |

**Tabelle 8.1:** Evaluation der entwickelten Systeme gegen die Anforderungen aus Abschnitt 2.3.

**Anforderung A6** fordert, dass ein aktueller Streckenzustand im System verwendet werden soll und dieser anhand von eintreffenden Daten angepasst werden soll. Wie in der Evaluation von Anforderung A2 bereits beschrieben, beziehen beide Systeme den Streckenzustand über Echtzeitdaten aus Kafka und verwenden den aktuellen Zustand. Anforderung A6 ist somit von beiden erfüllt.

**Insgesamt** erfüllt der Prototyp alle Anforderungen bis auf A5, wie in Tabelle 8.1 zu sehen ist. Dies liegt daran, dass der Prototyp keinen inkrementellen ML-Algorithmus verwendet. Das vorgeschlagene System konnte hingegen alle Anforderungen erfüllen. Dies liegt daran, dass das vorgeschlagene System die Verwendung eines inkrementellen ML-Algorithmus vorsieht. Beide Systeme ermöglichen zusätzlich zu den Anforderungen eine Beantwortung von Anfragen in Echtzeit, indem diese als Echtzeitdaten im System ankommen und über Stream Processing verarbeitet werden. Die Systeme können daher in Echtzeit Verspätungsvorhersagen liefern.

# 9 Zusammenfassung und Ausblick

Heutzutage gibt es viele Bereiche, in denen große Mengen an Daten anfallen, und für welche die Durchführung von Analysen auf diesen Daten von Vorteil ist. Dies ist zum Beispiel in der *Industrie 4.0* [Gr16], bei *eHealth* [Sta+18] und bei der Überwachung und Regelung des *ÖPNV* [Rag+16a] der Fall. Um möglichst viele vorteilhafte Informationen aus den Daten zu gewinnen, werden umfassende Analysen benötigt, die nicht nur historische, sondern auch Echtzeitdaten berücksichtigen und die Analyseergebnisse in Echtzeit anwenden können. Ziel dieser Arbeit ist es, zu untersuchen, inwiefern die Architektur *BRAID* für die Umsetzung solcher umfassender Analysen geeignet ist.

Hierfür wurde zunächst ein Anwendungsfall aus dem Bereich des ÖPNV entwickelt, welcher umfassende Analysen benötigt. Aus diesem Anwendungsfall wurden Anforderungen abgeleitet, welche ein System erfüllen muss, um dem Anwendungsfall gerecht zu werden. Da bei einer Betrachtung existierender Arbeiten zu Analysen auf Verkehrsdaten diese Anforderungen nicht erfüllt wurden, wurde ein neues System entwickelt.

Als Grundlage für dieses System wurden verschiedene Verarbeitungskonzepte, nämlich *Batch*, *Stream* und *Hybrid Processing* betrachtet. Es wurden außerdem Architekturen, darunter *BRAID*, und Systeme vorgestellt, die diese Verarbeitungsarten ermöglichen. Zudem wurden ML-Verfahren vorgestellt, diskutiert und evaluiert, welche für die Analysen, die im Anwendungsfall benötigt werden, genutzt werden können.

Aufgrund der gewonnenen Erkenntnisse wurde mithilfe der Architektur *BRAID* ein neues System entwickelt, welches prototypisch implementiert wurde. In einer abschließenden Evaluation zeigt sich, dass dieses System die erstellten Anforderungen erfüllen kann. Somit ermöglicht es die Umsetzung umfassender Analysen.

## Ausblick

Der umgesetzte Prototyp kann die Anforderung A5 nicht erfüllen. Dies liegt daran, dass im System die Nutzung eines inkrementellen ML-Verfahrens vorgesehen war, dies im Prototyp jedoch nicht umgesetzt wurde. Die Verwirklichung des kompletten Systems, wie es vorgeschlagen wurde, wäre also ein logischer nächster Schritt.

Das System ist zudem für die Verwendung auf großen Datenmengen gedacht. Ein weiterer Ansatzpunkt für zukünftige Arbeiten ist daher, das System über einen längeren Zeitraum auf großen Datenmengen und mit großen Strömen an Echtzeitdaten zu testen. Hierfür kann das System auf einem Cluster betrieben werden. Interessant ist bei diesem Testbetrieb, wie gut das System mit den Datenmengen zurecht kommt, wie viel Zeit die Verarbeitungen benötigen und wie sich die Qualität der Verarbeitungen entwickelt.

# Literaturverzeichnis


[Aly05] M. Aly. „Survey on multiclass classification methods“. In: *Neural Netw* 19 (2005), S. 1–9 (zitiert auf S. 37).

[Bal+04] R. Balcombe, R. Mackett, N. Paulley, J. Preston, J. Shires, H. Titheridge, M. Wardman, P. White. „The demand for public transport: a practical guide“. In: (2004). URL: http://discovery.ucl.ac.uk/1349/1/2004_42.pdf (zitiert auf S. 19).

[Ber+15] M. Berlingerio, V. Bicer, A. Botea, S. Braghin, N. Lopes, R. Guidotti, F. Pratesi. „Mobility Mining for Journey Planning in Rome“. In: *Proceedings, Part III, of the European Conference on Machine Learning and Knowledge Discovery in Databases - Volume 9286*. ECML PKDD 2015. Springer-Verlag, 2015, S. 222–226. ISBN: 978-3-319-23460-1. DOI: 10.1007/978-3-319-23461-8_18. URL: https://doi.org/10.1007/978-3-319-23461-8_18 (zitiert auf S. 25, 28, 45).

[CY15] R. Casado, M. Younas. „Emerging Trends and Technologies in Big Data Processing“. In: *Concurrency and Computation: Practice and Experience* 27.8 (Juni 2015), S. 2078–2091. ISSN: 1532-0626. DOI: 10.1002/cpe.3398. URL: http://dx.doi.org/10.1002/cpe.3398 (zitiert auf S. 15, 29, 30, 46).

[DC03] C. P. Diehl, G. Cauwenberghs. „SVM incremental learning, adaptation and optimization“. In: *Proceedings of the International Joint Conference on Neural Networks, 2003.* Bd. 4. 2003, 2685–2690 vol.4. DOI: 10.1109/IJCNN.2003.1223991 (zitiert auf S. 48).

[DH00] P. Domingos, G. Hulten. „Mining High-speed Data Streams“. In: *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’00. ACM, 2000, S. 71–80. ISBN: 1-58113-233-6. DOI: 10.1145/347090.347107. URL: http://doi.acm.org/10.1145/347090.347107 (zitiert auf S. 48, 51, 52).

[FT09] P. Fuangkhon, T. Tanprasert. „An incremental learning algorithm for supervised neural network with contour preserving classification“. In: *2009 6th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*. Bd. 02. 2009, S. 740–743. DOI: 10.1109/ECTICON.2009.5137153 (zitiert auf S. 48).

[Gie+18] C. Giebler, C. Stach, H. Schwarz, B. Mitschang. „BRAID: A Hybrid Processing Architecture for Big Data“. In: *Proceedings of the 7th International Conference on Data Science, Technology and Applications*. DATA’18. SCITEPRESS, Juli 2018, S. 294–301 (zitiert auf S. 15, 30–34).

[Gro+99] R. L. Grossman, S. Bailey, A. Ramu, B. Malhi, P. Hallstrom, I. Pulleyn, X. Qin. „The management and mining of multiple predictive models using the predictive modeling markup language“. In: *Information & Software Technology* 41 (1999), S. 589–595 (zitiert auf S. 50).

[Gr16] C. Gröger, C. Stach, B. Mitschang, E. Westkämper. „A mobile dashboard for analytics-based information provisioning on the shop floor“. In: *International Journal of Computer Integrated Manufacturing* 29.12 (2016), S. 1335–1354. DOI: 10.1080/0951192X.2016.1187292. eprint: https://doi.org/10.1080/0951192X.2016.1187292. URL: https://doi.org/10.1080/0951192X.2016.1187292 (zitiert auf S. 15, 67).

[Gun+98] S. R. Gunn et al. „Support vector machines for classification and regression“. In: *ISIS technical report* 14.1 (1998), S. 5–16 (zitiert auf S. 36).

[Has17] D. W. Hassan. *Big Data Frameworks*. Feb. 2017. URL: https://www.linkedin.com/pulse/big-data-frameworks-dr-waleed-hassan-pmp-cbip-bdscp-togaf-itil (zitiert auf S. 29, 30).

[HL17] L. Heppe, T. Liebig. „Real-Time Public Transport Delay Prediction for Situation-Aware Routing“. In: *KI 2017: Advances in Artificial Intelligence*. Hrsg. von G. Kern-Isberner, J. Fürnkranz, M. Thimm. Cham: Springer International Publishing, 2017, S. 128–141. ISBN: 978-3-319-67190-1 (zitiert auf S. 28, 45).

[Jia+08] W. Jiandong, Z. Guansheng, L. Zonglei. „A New Method to Alarm Large Scale of Flights Delay Based on Machine Learning“. In: *Knowledge Acquisition and Modeling, International Symposium on(KAM)*. Bd. 00. Dez. 2008, S. 589–592. DOI: 10.1109/KAM.2008.18. URL: doi.ieeecomputersociety.org/10.1109/KAM.2008.18 (zitiert auf S. 26, 28, 42, 43).

[JR04] R. Jeong, L. Rilett. „Bus arrival time prediction using artificial neural network model“. English (US). In: *Proceedings - 7th International IEEE Conference on Intelligent Transportation Systems, ITSC 2004*. 2004, S. 988–993 (zitiert auf S. 26, 28).

[Kre14] J. Kreps. *Questioning the Lambda Architecture*. Juli 2014. URL: https://www.oreilly.com/ideas/questioning-the-lambda-architecture (zitiert auf S. 30–32).

[Li+03] T. Li, M. Ogihara, Q. Li. „A comparative study on content-based music genre classification“. In: *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*. ACM. 2003, S. 282–289 (zitiert auf S. 42, 43).

[Li+04] T. Li, C. Zhang, M. Ogihara. „A comparative study of feature selection and multi-class classification methods for tissue classification based on gene expression“. In: *Bioinformatics* 20.15 (2004), S. 2429–2437 (zitiert auf S. 41, 43).

[LW+02] A. Liaw, M. Wiener et al. „Classification and regression by randomForest“. In: *R news* 2.3 (2002), S. 18–22 (zitiert auf S. 36).

[MG13] N. Mehra, S. Gupta. „Survey on multiclass classification methods“. In: (2013) (zitiert auf S. 42, 43).

[MW15] N. Marz, J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. 1st. Greenwich, CT, USA: Manning Publications Co., 2015. ISBN: 1617290343, 9781617290343 (zitiert auf S. 30, 31).

[Pet+16] E. Petzold, H. Lehmann, S. Jacobs, K. Kleist-Heinrich, D. Spiekermann-Klaas, J. Oberländer, J. Schneider. *Warum kommt der Bus zu spät?* 2016. URL: http://haltestelle.tagesspiegel.de/ (zitiert auf S. 19).

[Pol+01] R. Polikar, L. Upda, S. S. Upda, V. Honavar. „Learn++: An Incremental Learning Algorithm for Supervised Neural Networks“. In: *Trans. Sys. Man Cyber Part C* 31.4 (2001), S. 497–508. ISSN: 1094-6977. DOI: 10.1109/5326.983933. URL: https://doi.org/10.1109/5326.983933 (zitiert auf S. 48).

[Rag+16a] J. Raghothama, V. M. Shreenath, S. Meijer. „Analytics on Public Transport Delays with Spatial Big Data“. In: *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. BigSpatial '16. ACM, 2016, S. 28–33. ISBN: 978-1-4503-4581-1. DOI: 10.1145/3006386.3006387. URL: http://doi.acm.org/10.1145/3006386.3006387 (zitiert auf S. 15, 19, 21, 67).

[Rag+16b] J. Raghothama, V. M. Shreenath, S. Meijer. „Analytics on Public Transport Delays with Spatial Big Data“. In: *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. BigSpatial '16. ACM, 2016, S. 28–33. ISBN: 978-1-4503-4581-1. DOI: 10.1145/3006386.3006387. URL: http://doi.acm.org/10.1145/3006386.3006387 (zitiert auf S. 25, 28, 45).

[RN12] S. J. Russell, P. Norvig. *Künstliche Intelligenz: Ein moderner Ansatz*. Hrsg. von F. Langenau. 3., aktualisierte Auflage. München: Pearson, Higher Education, 2012. ISBN: 978-3-86894-098-5 (zitiert auf S. 35–38).

[SB17] J. Stefanowski, D. Brzezinski. „Stream Classification“. In: *Encyclopedia of Machine Learning and Data Mining*. Hrsg. von C. Sammut, G. I. Webb. Boston, MA: Springer US, 2017, S. 1191–1199. ISBN: 978-1-4899-7687-1. DOI: 10.1007/978-1-4899-7687-1_908. URL: https://doi.org/10.1007/978-1-4899-7687-1_908 (zitiert auf S. 48).

[SF86] J. C. Schlimmer, D. Fisher. „A Case Study of Incremental Concept Induction“. In: *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*. AAAI'86. Philadelphia, Pennsylvania: AAAI Press, 1986, S. 496–501. URL: http://dl.acm.org/citation.cfm?id=2887770.2887853 (zitiert auf S. 48).

[SR14] M. V. Subrahmanyam, S. V. Rao. „VFDT Algorithm for Decision Tree Generation“. In: 2014 (zitiert auf S. 48, 51, 52).

[Sta+04] A. Statnikov, C. F. Aliferis, I. Tsamardinos, D. Hardin, S. Levy. „A comprehensive evaluation of multicategory classification methods for microarray gene expression cancer diagnosis“. In: *Bioinformatics* 21.5 (2004), S. 631–643 (zitiert auf S. 41, 43).

[Sta+18] C. Stach, F. Steimle, B. Mitschang. „The Privacy Management Platform - An Enabler for Device Interoperability and Information Security in mHealth Applications“. In: *Proceedings of the 11th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2018) - Volume 5: HEALTHINF, Funchal, Madeira, Portugal, January 19-21, 2018*. 2018, S. 27–38. DOI: 10.5220/0006537300270038. URL: https://doi.org/10.5220/0006537300270038 (zitiert auf S. 15, 67).

[Stu18] S.-B.-C. in Stuttgart. *Pünktlichkeits- und Verspätungsdiagramme*. 2018. URL: https://s-bahn-chaos.de/service/puenktlichkeits-und-verspaetungsdiagramme/ (zitiert auf S. 19).

[Sun+16] F. Sun, Y. Pan, J. White, A. Dubey. „Real-Time and Predictive Analytics for Smart Public Transportation Decision Support System“. In: *2016 IEEE International Conference on Smart Computing (SMARTCOMP)* (2016), S. 1–8 (zitiert auf S. 27, 28).

[Sun+17] Y. Sun, Q. Yan, Y. Jiang, X. Zhu. „Reliability prediction model of further bus service based on random forest“. In: *Journal of Algorithms & Computational Technology* 11.4 (2017), S. 327–335. DOI: 10.1177/1748301817725306. eprint: https://doi.org/10.1177/1748301817725306. URL: https://doi.org/10.1177/1748301817725306 (zitiert auf S. 27, 28).

[Tsy04] A. Tsymbal. „The problem of concept drift: definitions and related work“. In: *Computer Science Department, Trinity College Dublin* 106.2 (2004) (zitiert auf S. 15).

[Utg89] P. E. Utgoff. „Incremental Induction of Decision Trees“. In: *Machine Learning* 4.2 (1989), S. 161–186. ISSN: 1573-0565. DOI: 10.1023/A:1022699900025. URL: https://doi.org/10.1023/A:1022699900025 (zitiert auf S. 48).

[Vav+13] V. Vavilapalli, A. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, E. Baldeschwieler. „Apache hadoop YARN: Yet another resource negotiator“. English. In: *Proceedings of the 4th Annual Symposium on Cloud Computing, SoCC 2013*. United States: Association for Computing Machinery (ACM), Jan. 2013. DOI: 10.1145/2523616.2523633 (zitiert auf S. 35).

[Win+16] W. Wingerath, F. Gessert, S. Friedrich, N. Ritter. „Real-time stream processing for Big Data“. In: *it-Information Technology* 58.4 (2016), S. 186–194 (zitiert auf S. 32).

[Yag+12] M. Yaghini, M. M. Khoshraftar, M. Seyedabadi. „Railway passenger train delay prediction via neural network model“. In: *Journal of Advanced Transportation* 47.3 (2012), S. 355–368. DOI: 10.1002/atr.193. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/atr.193. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/atr.193 (zitiert auf S. 26, 28).

[CEN11] CEN - Europäisches Komitee für Normung. *Public transport — Service interface for real-time information relating to public transport operations — Part 5: Functional service interfaces: Situation Exchange*. 2011. URL: http://gÃijterbahnen.de/cen-ts-15531-5-nk04-md-wb20141120a.pdfx (zitiert auf S. 49).

[Fab10] Fabian Bürger. *Svm intro*. 2010. URL: https://de.wikipedia.org/wiki/Datei:Svm_intro.svg (zitiert auf S. 39, 40).

[Ver14] Verband Deutscher Verkehrsunternehmen e. V. (VDV). *VDV-Schrift Nr. 431-2, Echtzeit Kommunikations- und AuskunftsplattformEKAP*. 2014. URL: https://www.vdv.de/vdv-431-2-ekap-schnittstellenbeschreibung.pdfx (zitiert auf S. 21, 45, 49).

[Ver18a] Verkehrs- und Tarifverbund Stuttgart GmbH. *S-Bahn-Liniennetz*. 2018. URL: http://www.vvs.de/download/SBahn_Liniennetz.pdf (zitiert auf S. 20).

[Ver18b] Verkehrs- und Tarifverbund Stuttgart GmbH. *Sechs Verbünde aus ganz Deutschland starten OpenData- und OpenService-Plattform im ÖPNV*. 2018. URL: https://www.vvs.de/open-data/ (zitiert auf S. 49).

[Ver18c] Verkehrs- und Tarifverbund Stuttgart GmbH. *VVS Open Data Portal - API*. 2018. URL: https://www.openvvs.de/pages/api (zitiert auf S. 45, 49).

[Ver18d] Verkehrs- und Tarifverbund Stuttgart GmbH. *VVS Open Data Portal - Datensätze*. 2018. URL: https://www.openvvs.de/dataset (zitiert auf S. 49).

[Ver18e] Verkehrs- und Tarifverbund Stuttgart GmbH. *VVS Open Data Portal*. 2018. URL: https://www.openvvs.de/ (zitiert auf S. 21, 45, 49).

Alle URLs wurden zuletzt am 12. 11. 2018 geprüft.

**Erklärung**

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

---

Ort, Datum, Unterschrift