Technical Report

PG 594 – Big Data

Mohamed Asmi,
Alexander Bainczyk,
Mirko Bunse,
Dennis Gaidel,
Michael May,
Christian Pfeiffer,
Alexander Schieweck,
Lea Schönberger,
Karl Stelzner,
David Sturm,
Carolin Wiethoff,
Lili Xu

05/2016

technische universität dortmund

SFB 876 Verfügbarkeit von Information durch Analyse unter Ressourcenbeschränkung

Part of the work on this technical report has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project C3.

Speaker: Prof. Dr. Katharina Morik
Address: TU Dortmund University
Joseph-von-Fraunhofer-Str. 23
D-44227 Dortmund
Web: http://sfb876.tu-dortmund.de

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Aufbau der Arbeit 1

1.2 Anwendungsfall 2

1.2.1 Datenanalyse 3

1.3 Analyseziele 5

1.3.1 Gamma/Hadron-Klassifizierung 7

1.3.2 Energie-Abschätzung 8

1.4 Analyse mit den FACT Tools 8

1.4.1 Analysekette 8

1.4.2 Grenzen von `streams` 10

I Big Data Analytics **11**

2 Einführung in Big Data Systeme **13**

2.1 Nutzen von Big Data 14

2.2 Probleme mit herkömmlichen Ansätzen 14

2.3 Anforderungen an Big Data Systeme 15

3 Lambda-Architektur **17**

3.1 Batch Layer 20

3.1.1 Apache Hadoop 20

3.1.2 Apache Spark 23

3.2 Speed Layer 28

3.2.1 Apache Storm 28
3.2.2 Apache Trident 30
3.2.3 Spark Streaming 31
3.2.4 streams-Framework 34
3.3 Serving Layer 36
3.3.1 Datenbanken 36
3.3.2 RESTful APIs 41

4 Maschinelles Lernen 45

4.1 Ensemble Learning 47
4.1.1 Bagging 48
4.1.2 Boosting 50
4.1.3 Fazit 51
4.2 Clustering und Subgruppenentdeckung 51
4.2.1 Clustering 52
4.2.2 Subgruppenentdeckung 54
4.3 Verteiltes Lernen 57
4.3.1 Peer-to-Peer-K-Means 58
4.3.2 Distributed random forests 59
4.3.3 Kompression von Entscheidungsbäumen 59
4.4 Statisches und Inkrementelles Lernen 60
4.5 Concept Drift und Concept Shift 60
4.6 Learning with Imbalanced Classes 62
4.6.1 Einfluss auf Klassifikatoren 63
4.6.2 Bewertung von Klassifikatoren 63
4.6.3 Verbesserung von Klassifikatoren 65
4.7 Feature Selection 66
4.7.1 Vorteile 67
4.7.2 Problemstellung 68
4.7.3 Arten von Algorithmen 69
4.7.4 Korrelation als Heuristik 70

4.7.5 CFS 71
4.7.6 Fast-Ensembles 72
4.8 Sampling und Active Learning 75
4.8.1 Der naive Ansatz 75
4.8.2 Re-Sampling 76
4.8.3 VLDS-Ada^2Boost 77
4.8.4 Active Learning 78

II Architektur und Umsetzung **81**

5 Komponenten und Architektur **83**

6 Datenbeschreibung **87**
6.1 FITS-Dateiformat 87
6.2 Rohdaten 88
6.3 Monte-Carlo-Daten 88
6.4 Drs-Daten 88
6.5 Aux-Daten 89

7 Indexierung der Rohdaten **91**
7.1 MongoDB 91
7.2 Elasticsearch 92
7.3 PostgreSQL 93
7.4 Auswahl der Datenbank 93

8 RESTful API **95**
8.1 Design 95
8.1.1 Endpunkte 95
8.1.2 Rückgabeformate 96
8.1.3 Dokumentation 97
8.2 Implementierung 98
8.2.1 Spring Framework 98
8.2.2 Filterung 99

8.2.3 Jobs . . . 105
8.3 Ein Beispiel-Client: Die Web-UI . . . 111
8.3.1 Single Page Applications . . . 111
8.3.2 Implementierung . . . 111

9 Verteilung von Streams-Prozessen 117
9.1 Nebenläufigkeit der Verarbeitung . . . 117
9.2 XML-Spezifikation verteilter Prozesse . . . 118
9.3 Verarbeitung der XML-Spezifikation . . . 119
9.4 Verteilung der Daten . . . 119
9.5 Verteilte Batch-Prozesse . . . 120
9.5.1 Daten- und Kontrollfluss . . . 120
9.5.2 Instanziierung von Streams in den Workern . . . 121
9.6 Verteilte Streaming-Prozesse . . . 123
9.6.1 Datenfluss . . . 123
9.6.2 Arbeitsweise der Receiver . . . 125

10 Einbindung von Spark ML 127
10.1 Spark ML vs. MLlib . . . 127
10.2 XML-Spezifikation . . . 130
10.3 Umsetzung . . . 134

11 Verteilte Ein- und Ausgabe 141
11.1 MultiStream-Generatoren . . . 141
11.2 REST-Stream . . . 142
11.2.1 `RestFulStream` . . . 142
11.2.2 `RestFulMultiStream` . . . 143
11.3 Verteilte CSV-Ausgabe . . . 144

12 Organisation 147
12.1 Agiles Projektmanagement . . . 147
12.1.1 Probleme Nicht-Agiler Verfahren . . . 148
12.1.2 Das Agile Manifest . . . 148
12.1.3 Scrum . . . 149
12.1.4 Kanban . . . 151
12.2 Wahl des Verfahrens . . . 153

III Evaluation und Ausblick 155

13 Verteilte Streams-Prozesse 157

13.1 Batch-Prozesse 157

13.1.1 Rechenleistung 157

13.1.2 Arbeitsspeicher 158

13.1.3 Fehlertoleranz und Generalisierbarkeit 158

13.2 Streaming-Prozesse 159

13.2.1 Rechenleistung 159

13.2.2 Arbeitsspeicher 160

13.2.3 Fehlertoleranz 160

13.3 Performanz der Erweiterungen 160

13.3.1 Feature Extraction auf MC-Daten 160

13.3.2 Feature Extraction auf Teleskop-Daten 162

14 Modellqualität in Spark ML 165

14.1 Vergleich der Klassifikationsmodelle 165

14.2 Vergleich der Regressionsmodelle 168

14.3 Trainingszeit von Modellen 168

14.4 Einfluss der Waldgröße auf die Modellqualität 170

15 Fazit 175

15.1 Ergebnisse 175

15.2 Ausblick 176

15.3 Retrospektive der Organisation 176

15.3.1 Projekt-Initialisierung 177

15.3.2 Organisation im ersten Semester 177

15.3.3 Organisation im zweiten Semester 178

15.3.4 Abschließende Bewertung 179

IV Benutzerhandbuch 181

16 Vorbereitung eines Clusters 183
16.1 Verfügbarkeit von Dependencies . . . 184
16.2 Starten der REST API & Web-UI . . . 184
16.2.1 Standard . . . 185
16.2.2 Docker . . . 185

17 Shell-Script 187

18 Web-UI 189
18.1 Konfiguration . . . 189
18.2 Starten und Managen von Jobs . . . 190
18.3 Schedulen von Jobs . . . 192
18.4 Testen von Filtern . . . 193
18.5 Einsehen der REST-API Dokumentation . . . 194

19 Maschinelles Lernen mit TELEPhANT 195
19.1 Datenaufbereitung . . . 195
19.2 Modelltraining und Evaluation . . . 199
19.2.1 Training und Klassifikation . . . 199
19.2.2 Evaluation . . . 202
19.2.3 Parameterstudie . . . 205
19.3 Der TreeParser . . . 206
19.3.1 Struktur der Lernbäume . . . 207
19.3.2 Die Parser-Klasse . . . 208
19.3.3 CombinedTreeFeatures . . . 208

A Liste der Operatoren 209

B XMLs zum Kapitel „Modellqualität“ 211
B.1 Experiment 14.1: Vergleich von Klassifikationsmodellen . . . 211
B.2 Experiment 14.4: Parameterstudie zur Waldgröße von Random Forests . . . 214
B.3 Experiment 14.2: Vergleich von Regressionsmodellen . . . 215
B.4 Experiment 14.3: Trainingszeit in Abhängigkeit der Clusterressourcen . . . 218

C Liste der referenzierten Software **219**

Abkürzungsverzeichnis **221**

Abbildungsverzeichnis **225**

Literaturverzeichnis **235**

Kapitel 1

Einleitung

In der heutigen Welt wird die Verarbeitung großer Mengen von Daten immer wichtiger. Dabei wird eine Vielzahl von Technologien, Frameworks und Software-Lösungen eingesetzt, die explizit für den Big-Data-Bereich konzipiert wurden oder aber auf Big-Data-Systeme portiert werden können. Ziel dieser Projektgruppe (PG) ist der Erwerb von Expertenwissen hinsichtlich aktueller Tools und Systeme im Big-Data-Bereich anhand einer realen, wissenschaftlichen Problemstellung. Vom Wintersemester 2015/2016 bis zum Ende des Sommersemesters 2016 beschäftigte sich diese Projektgruppe mit der Verarbeitung und Analyse der Daten des durch den Fachbereich Physik auf der Insel La Palma betriebenen First G-APD Cherenkov Telescope (FACT). Dieses liefert täglich Daten im Terabyte-Bereich, die mit Hilfe des Clusters des Sonderforschungsbereiches 876 zunächst indiziert und dann auf effiziente Weise verarbeitet werden müssen, sodass diese Projektgruppe im besten Falle die Tätigkeit der Physiker mit ihren Ergebnissen unterstützen kann. Wie genau dies geschehen soll, sei auf den nachfolgenden Seiten genauer beleuchtet - begonnen mit dem dezidierten Anwendungsfall, unter Berücksichtigung der notwendigen fachlichen sowie technischen Grundlagen, bis hin zu den finalen Ergebnissen.

1.1 Aufbau der Arbeit

Zunächst beschreiben wir die Grundlagen zum Anwendungsfall und die für uns relevanten Analyseziele der Physiker. Weiterhin werden die bisherigen Ansätze der Physiker zur Erreichung dieser Ziele und die FACT Tools vorgestellt, mit deren Hilfe die aktuelle Analysekette durchgeführt wird. Der Rest des Endberichts ist in vier Teile gegliedert.

Der erste Teil befasst sich mit dem Thema Big Data Analytics. Zunächst wird in die Big Data Thematik eingeführt, wobei nicht nur der Begriff geklärt wird, sondern auch erläutert wird, welche Herausforderungen Big Data mit sich bringt und warum es sich lohnt, auf diese Herausforderungen einzugehen. Danach folgt eine Beschreibung der Lambda-Architektur,

welche typischerweise für Big Data Anwendungen umgesetzt wird. In den darauffolgenden drei Kapiteln wird näher darauf eingegangen, mit welchen Methoden und mit welcher Software die Architektur verwirklicht werden kann. Abschließend zu diesem Teil folgt eine Einführung in das maschinelle Lernen.

Der zweite Teil gibt einen Einblick in die Architektur unserer Software und die Umsetzung derselben. Dazu wird dargestellt, wie wir die Rohdaten mit Hilfe verschiedener Datenbanken indexieren, wie die REST-API umgesetzt wird und welche Erweiterungen wir aus welchen Gründen am `streams`-Framework vorgenommen haben.

Der dritte Teil widmet sich der Evaluation unserer Software zum Ende der Projektgruppe. Dabei geht es zuerst um die Verbesserungen, die mit einer verteilen Ausführung von `streams`-Prozessen möglich sind. Anschließend werden verschiedene Modelle entsprechend des Anwendungsfalles trainiert und evaluiert. Außerdem fassen wir abschließend zusammen, welche Ergebnisse wir in den zwei Semestern der Projektgruppe erzielt haben, und geben einen kurzen Ausblick über denkbare Erweiterungen.

Das Benutzerhandbuch mit Informationen zur Installation und Ausführung im Cluster sowie zur Web-Oberfläche und einigen Tipps zum maschinellen Lernen findet sich im letzten Teil.

1.2 Anwendungsfall

Ein Teilgebiet der Astrophysik ist die Untersuchung von Himmelsobjekten, welche hochenergetische Strahlung ausstoßen. Beim Eintritt dieser Strahlung in die Erdatmosphäre werden Lichtimpulse erzeugt, die sogenannte Cherenkov-Strahlung, welche mithilfe von Teleskopen aufgezeichnet und analysiert werden können. Ein Teil der Analyse umfasst das Erstellen von Lichtkurven, welche das emittierte Licht in Relation zur Zeit stellen, sodass Eigenschaften des beobachteten Himmelsobjektes hergeleitet werden können. Mithilfe solcher Kurven können dann unter anderem Supernovae klassifiziert werden [21, 86].

Das auf der kanarischen Insel La Palma aufgebaute FACT dient der Beobachtung dieser Gammastrahlung im TeV Bereich ausstoßenden Himmelsobjekte. Es setzt sich aus einer mit 1440 *geiger-mode avalanche photodiods* (G-APD) Pixel ausgerüsteten Kamera zusammen, welche die Cherenkov-Strahlung in der Atmosphäre aufzeichnen kann. Ein Ziel des FACT Projekts ist es, herauszufinden, ob die G-APD Technologie zur Beobachtung von Cherenkov-Strahlung eingesetzt werden kann [3].

Cherenkov-Strahlung entsteht, wenn energiereiche geladene Teilchen, z.B. Gammastrahlung, die Erdatmosphäre mit sehr hoher Geschwindigkeit durchqueren. Dabei kollidieren diese Teilchen mit Partikeln der Atmosphäre, wodurch neue geladene Teilchen aus dieser Kollision entstehen, welche wiederum Lichtblitze erzeugen und mit weiteren Partikeln

kollidieren können. Ein solche Kaskade von Kollisionen wird unter anderem als Gamma-Schauer bezeichnet. Die Lichtblitze können dann von Teleskopen wie dem FACT wahrgenommen und analysiert werden, um z.B. den Ursprung der kosmischen Teilchen zu bestimmen (siehe Figure 1.1).

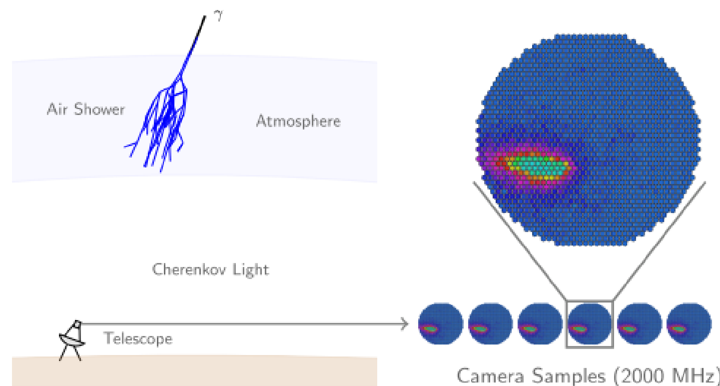


Abbildung 1.1: Visuelle Darstellung eines Gamma-Showers (oben links), welcher von Teleskopen aufgezeichnet wird (unten links) und in Grafiken der einzelnen Aufnahmen dargestellt werden kann (rechts) [17]

Ein Hauptproblem in diesem Unterfangen ist dabei die Klassifizierung der aufgezeichneten Lichtblitze, denn neben der Cherenkov Strahlung wird durch Hintergrundrauschen das aufgezeichnete Bild gestört. Die Einteilung der Cherenkov-Strahlung, hervorgerufen durch die kosmische Gammastrahlung, und des Hintergrundrauschens wird zudem erschwert, da die beiden Klassen stark ungleichmäßig verteilt sind. Bockerman et al. [17] nennen hier eine Gamma-Hadron Klassenverteilung von 1:1000 bis 1:10000. Aufgrund dieser stark ungleichmäßigen Verteilung ist eine sehr große Menge von Daten für eine relevante Klassifizierung erforderlich.

Ein wichtiges Merkmal in der Klassifizierung dieser Daten ist, dass zum Lernen Simulationen der eigentlichen Beobachtungen verwendet werden müssen, da sie selbst keine Label besitzen. Dazu wird die *Cosmic Ray Simulations for Kascade* (CORSIKA) [44] Monte-Carlo-Simulation verwendet, welche für eine Reihe von Eingaben eine statistische Simulation eines in die Atmosphäre eintreffenden Partikel, wie unter anderem Photonen und Protonen, berechnet. Die Ausgaben einer solchen Simulation sind dann gelabelt und können als Trainingsdaten für Lernmodelle verwendet werden.

1.2.1 Datenanalyse

Die Auswertung von Beobachtungen solcher Schauer ist ein schwieriges Unterfangen. Nicht nur wegen der ungleichmäßigen Verteilung, sondern auch aufgrund der gigantischen Masse

an Daten, die analysiert werden muss. Ein mögliches Vorgehen ist dabei, die Daten auf ein beliebiges verteiltes Dateisystem zu lagern, sodass diese auf Abruf angefordert werden können. Zugriffe auf diesen Daten können dann über Ressourcenmanager, wie unter anderem TORQUE (Terascale Open-source Resource and QUEue Manager) [1] einer ist, verteilt werden.

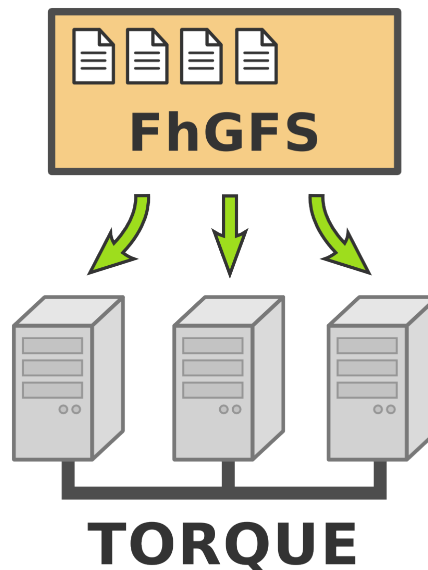


Abbildung 1.2: Beispielhafte Verwaltung mit TORQUE und FhGFS (jetzt BeeGFS)

Diese Vorgehensweise bietet die Möglichkeit, Datenspeicherung und Datenverarbeitung voneinander zu trennen, hat jedoch den Nachteil, dass die Daten zunächst an die Verarbeitungsknoten gesendet werden müssen (Figure 1.2).

Code-2-Data Ein alternativer Ansatz verfolgt das Ziel, die Datenverarbeitung und -speicherung miteinander zu kombinieren, wodurch eine performante Verarbeitung der Daten ermöglicht werden kann. Apache bietet mit Hadoop (Kapitel 3.1.1) und Spark (3.1.2) eine solche Umgebung an.

Figure 1.3 zeigt ein mögliches Konzept für die Datenanalyse von gespeicherten Daten mittels HDFS und Spark. Hierbei wird versucht, die Datenanalyse an die Quelle zu bringen, sodass keine Daten mehr zeitaufwändig an die jeweiligen Rechenzentren geschickt werden müssen. Der große Vorteil eines solchen Systems ist, dass zum Einen Daten direkt an den Quellen bearbeitet werden können, aber gleichzeitig noch die Möglichkeit besteht, Daten aus benachbarten Knoten anzufordern. Dies wird meist im Falle von Systemausfällen und Störungen benötigt, um die Fehlertoleranz der Datenanalyse zu verbessern.

Die Verteilung der Aufgaben kann hier zum Beispiel von YARN übernommen werden.

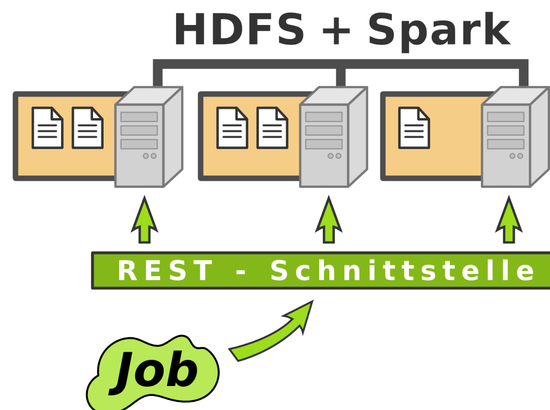


Abbildung 1.3: Code-2-Data mit Hadoop und Spark

Gleichzeitig kann eine API entworfen werden, welche die Schnittstelle zwischen Endnutzer und Datenmanagement herstellt.

1.3 Analyseziele

Alle diese grundlegenden Informationen gingen aus einem Treffen mit einem Repräsentanten der Physiker hervor, welches zu Beginn unserer Projektgruppe stattfand. Wir machten uns nicht nur mit den physikalischen Hintergründen bekannt, sondern legten auch gemeinsam die exakten Analyseziele fest. Im Nachhinein fassten wir das gewonnene Wissen in *User Stories* zusammen, welche nicht nur einen Überblick über diese Ziele geben, sondern auch das Entwickeln von *Sprints* vorbereiten sollten, so wie sie in Kapitel 12.1.3 über das Projektmanagement mit SCRUM beschrieben werden. Im Folgenden werden die aus unserer Sicht wichtigsten Analyseziele zusammengefasst, welche wir mit unserer Software zum Ende der Projektgruppe ermöglichen wollten.

Durchsuchbarkeit der Events Zuerst ist es wichtig, einen Überblick über die Events bekommen zu können. Dazu soll man die Events nach ihren Metadaten durchsuchen können. Mithilfe einer REST-API (zur Beschreibung siehe subsection 3.3.2, für unsere Umsetzung siehe chapter 8) sollen vom Anwender Metadaten spezifiziert werden, zu denen alle passenden Events zurückgeliefert werden. Damit wird es einfach, alle Events zu suchen, die beispielsweise in einem kontinuierlichen Zeitintervall liegen.

Normalisierung der Rohdaten Ein weiteres Anliegen ist die Normalisierung der Rohdaten. Wie man in Kapitel 6.4 nachlesen kann, existiert zu jeder Aufnahmedatei eine Drs-Datei zur Kalibrierung. Es ist mühsam, zu jeder Aufnahmedatei per Hand die passende Drs-Datei zu finden. Um das System so benutzerfreundlich wie möglich zu gestalten,

soll diese Kalibrierung daher selbstständig durchgeführt werden, d.h., die passenden Drs-Dateien werden automatisch gesucht und gefunden.

Gamma-Hadron-Separation Eine große Aufgabe bilden außerdem die maschinellen Lernaufgaben. Zum Einen soll die Gamma-Hadron-Separation ermöglicht werden, sodass aus den aufgezeichneten Teleskopdaten die für die Physiker interessanten Gammastrahlungen erkannt und separiert werden können. Dabei ist es wieder praktisch, nach Metadaten durchsuchen zu können, um beispielsweise alle Gammastrahlungen einer bestimmten Region oder eines bestimmten Zeitraumes anzusehen. Da es viele verschiedene Klassifikationsverfahren zur (binären) Klassifikation gibt, sollen in unserer Software Methoden enthalten sein, mit denen man verschiedene Lernverfahren einfach evaluieren kann, sodass die Eignung der Verfahren im Bezug auf die Gamma-Hadron-Separation abgeschätzt werden kann. Eine Übersicht mit für uns möglicherweise interessanten Lernverfahren ist in chapter 4 zu finden.

Energieschätzung Zu den Lernaufgaben gehört außerdem die Energieschätzung, bei welcher die Energie der gefundenen Gammastrahlungen beziehungsweise der darin involvierten Partikel geschätzt wird. Dies soll über eine Graphical User Interface (GUI) oder eine Application Programming Interface (API) einfach möglich sein, sodass die Schätzung mit nur einem Mausklick oder einem einfachen Aufruf angestoßen werden kann. Die dabei entstehenden Ergebnisse sollen sich außerdem grafisch als Lichtkurven darstellen lassen.

Realzeitliche Verarbeitung Eine große Rolle spielt die realzeitliche Einsetzbarkeit unserer Software. Wenn die Teleskopdaten in Echtzeit gespeichert und weiterverarbeitet werden, kann vor Ort über mögliche Gammastrahlungen in Echtzeit informiert werden, um eventuelle weitere Arbeitsschritte auf die Daten anzuwenden, welche Gammastrahlungen enthalten. Dazu gehört unter anderem auch realzeitliches Filtern. Dabei sollen Daten, die offensichtlich nicht für die Analyse wertvoll sind und auf keinen Fall eine Gammastrahlung enthalten, sofort gelöscht werden. Anstatt die Ressourcen zu verbrauchen, sollen diese Daten gar nicht erst gespeichert und weiterverarbeitet werden. Für möglicherweise interessante Daten soll eine automatische Speicherung und Indexierung erfolgen, sodass dieser Teil der Arbeit nicht jeden Morgen nach der Aufzeichnung manuell angestoßen werden muss. Einblicke in realzeitliches Arbeiten und Streamen gibt section 3.2.

Instrumenten-Monitoring Mit Hilfe der kürzlich aufgenommenen Daten soll darüber hinaus Instrumenten-Monitoring betrieben werden. Es soll geprüft werden, ob alle Instrumente einwandfrei funktionieren oder ob es Hinweise auf ein Versagen der Technik gibt. In diesem Fall soll das System die Nutzer vor Ort warnen, sodass eine Reparatur oder ein Austausch der beschädigten Teile möglichst schnell erfolgen kann.

Inkrementelle Ergebnisausgabe Hinzu kommt, dass, abhängig von der Lernaufgabe, Teilergebnisse abgefragt werden sollen. Möchte der Nutzer nicht die komplette Laufzeit abwarten, bis das Endergebnis komplett berechnet wurde, kann es sinnvoll sein, das Ergebnis während des Rechenprozesses inkrementell zur Verfügung zu stellen, sofern das Lernverfahren es zulässt. So können schon während der weiteren Verarbeitung erste Hypothesen über die Daten angestellt werden und basierend darauf weitere Entscheidungen zum Handling der Daten getroffen werden.

Datenexport Für alle Aufgaben ist es außerdem wichtig, dass Dateien und Ergebnisse exportiert werden können. Dazu zählt nicht nur der möglicherweise komprimierte Export von Klassifikationsergebnissen, sondern auch der Export von Log-Dateien und Grafiken, beispielsweise der Lichtkurven, welche bei der Schätzung der Energie entstehen können.

Insgesamt werden viele Forderungen an unsere Software gestellt, welche korrekt und benutzerfreundlich umgesetzt werden müssen. In den folgenden beiden Unterkapiteln wird kurz beschrieben, welche Methoden zu den Klassifikations- beziehungsweise Regressionsaufgaben der oben aufgeführten Analyseziele genutzt werden können.

1.3.1 Gamma/Hadron-Klassifizierung

Im Gebiet des maschinellen Lernens gibt es viele unterschiedliche Ansätze zur binären Klassifizierung von Daten. Im Bereich der Klassifizierung von Gamma- und Hadron-Events wurden Untersuchungen zu den wohl bekanntesten bereits durchgeführt. Dazu zählen unter anderem

- *Direct selection in the image parameters*,
- *Random Forest*,
- *Support Vector Machine (SVM)* und
- *Artificial Neural Network*,

welche von Bock et al. [14] und Sharma et al. [82] näher untersucht wurden, mit dem Ergebnis, dass der *Random Forest* die besten Ergebnisse liefert.

Zum Vergleich der jeweiligen Methoden wurden verschiedene Qualitätsmaße verglichen. Ein wichtiges solches Maß ist der Qualitätsfaktor $Q = \frac{\varepsilon_\gamma}{\sqrt{\varepsilon_P}}$, wobei ε_γ die Anzahl der korrekt klassifizierten Gamma-Events und ε_P die Anzahl der als Gamma klassifizierten Hadron-Events beschreibt. Der Q-Faktor ist damit vergleichbar mit der statistischen Signifikanz.

1.3.2 Energie-Abschätzung

Ein weiteres Anwendungsgebiet für maschinelles Lernen ist die Abschätzung der Energie von klassifizierten Gamma-Events. Da mithilfe der Energie viele physikalische Eigenschaften bestimmt werden können, besteht eine wichtige Aufgabe darin, eine korrekte Energieangabe zu erhalten.

Die eigentliche maschinelle Lernaufgabe ist eine typische Regression, bei der ein Modell gefunden werden muss, welches die Energie basierend auf einer Reihe von Features vorhersagen kann. Untersuchungen von Berger et al. [11] besagen, dass bereits das Feature `size` für eine gute Einschätzung mithilfe eines *Random Forest* genügt.

1.4 Analyse mit den FACT Tools

Nachdem die Analyseziele detailliert erläutert wurden, befasst sich dieser Abschnitt mit den bisherigen Ansätzen der Physiker zur Analyse der Teleskopdaten. Für die Verarbeitung von Flexible Image Transport System (FITS)-Dateien (siehe chapter 6), die mit Hilfe des FACT-Teleskops aufgenommen werden, wurden die FACT-Tools als Erweiterung des `streams`-Frameworks implementiert.

Bei den FACT-Tools [17] wurden Inputs und Funktionalitäten für `streams` implementiert, die für die Verarbeitung der Teleskop-Rohdaten notwendig sind. Dabei wurde z.B. ein Stream `fact.io.fitsStream` implementiert, der in der Lage ist, eine FITS-Datei von einem Input zu lesen. Darüber hinaus ermöglichen es die FACT-Tools, eine Datenanalyse mit allen Schritten, die für die Physiker von Wichtigkeit sind und in diesem Abschnitt erläutert werden, durchzuführen. Dazu gehören alle Vorverarbeitungsschritte sowie das Einbinden von Bibliotheken für maschinelles Lernen.

1.4.1 Analysekette

Die von dem FACT-Teleskop erzeugten Daten werden für die Erforschung der Gammastrahlen mit verschiedenen Methoden des maschinellen Lernens analysiert. In diesem Abschnitt wird die Analysekette der Daten von der Aufnahme der Daten bis zu den ersten Ergebnissen der Datenanalyse betrachtet.

Die Datenanalyse kann dabei in drei Schritte unterteilt werden: Datensammlung, Datenvorverarbeitung und Datenanalyse.

Datensammlung

Bei dem Eintreten eines Teilchen in die Atmosphäre wird ein Schauer erzeugt. Der Schauer entsteht durch die Interaktion des Teilchens mit Elementen in der Atmosphäre. Dieser Schauer strahlt ein Licht aus, das von den Kameras des FACT-Teleskops aufgenommen wird. Die entstandenen Bilder werden in den FITS-Dateien gespeichert.

Dabei werden nicht nur die Bilder des Schauers gespeichert, sondern auch andere nützliche Informationen wie zum Beispiel die Rauschfaktoren, die Stärke des Mondlichts und anderer Lichtquellen etc. Diese Informationen können später bei der Auswertung der Daten von größter Wichtigkeit sein.

Datenvorverarbeitung

Nach der Datensammlung werden nun die Vorverarbeitungsschritte mithilfe der FACT-Tools durchgeführt. Darunter fallen zum Beispiel das Imagecleaning, das Kalibrieren der Daten sowie das Extrahieren von Features.

Unter Imagecleaning versteht man das Filtern der Rauschinformation. Es wird ermittelt, welche Pixel der Aufnahme überhaupt Teil des Schauers sind. Alle anderen Pixel werden entfernt. So wird vermieden, dass wertlose Informationen gespeichert werden, die unsere Datenmenge noch zusätzlich vergrößern.

Um die Daten für maschinelle Lernverfahren aufzubereiten, wird eine *Feature-Extraktion* durchgeführt. Dabei werden die bereinigten Bilddaten (Pixelintensitäten) zu numerischen Merkmalen abstrahiert, wie etwa die Länge und Breite eines ellipsenförmigen Schauers im Bild. In den FACT-Tools ist die Extraktion einer ganzen Menge von Features implementiert.

Die FACT-Tools bieten allerdings nicht nur diese Verarbeitungsschritte an, sondern können je nach Analyseaufgabe auch verschiedene andere Vorverarbeitungsschritte durchführen [17]. Ist die Datenvorverarbeitung abgeschlossen, kann mit der eigentlichen Datenanalyse begonnen werden.

Datenanalyse

Die Datenanalyse besteht in unserem Fall aus der Separation der Gamma- und Hadron-Strahlen sowie der Energie Einschätzung der Gammastrahlen.

Gamma- /Hadron-Separation: Durch das Anwenden von Klassifikationsverfahren, zum Beispiel RandomForest, können Gamma-Strahlen von anderen Events unterschieden werden. Die Modelle werden dabei mithilfe der simulierten Daten (Monte-Carlo-Daten) section 6.3 trainiert. Danach werden sie auf die „echten“ Teleskop-Daten angewendet.

Energie-Einschätzung: Mithilfe der Spektrumskurve und den aus der Datenanalyse gewonnen Informationen kann nun die emittierte Energie vorhergesagt werden.

Abbildung 1.4: Analysekette

Der Ablauf der Analysekette wird in Figure 1.4 veranschaulicht.

1.4.2 Grenzen von `streams`

Das FACT-Teleskop sammelt jede Nacht neue Daten, weshalb die Größe der gesammelten Daten sehr schnell wächst. Die Analyse dieser Daten ist also ein Big-Data-Problem und es ist daher nicht sinnvoll, sie auf einem einzelnen Rechner durchzuführen.

Da das `streams`-Framework von sich aus nicht verteilt ausführbar ist, stößt es deshalb bei dieser Datenmenge an seine Grenzen. Unsere Experimente haben gezeigt, dass auch bei Ausführung der FACT-Tools auf einem Rechencluster die einzelnen Prozessoren immer sequentiell ausgeführt wurden. Daher würde eine interne verteilte Ausführung der Prozessoren vom `streams`-Framework nicht gewährleistet. Deshalb scheint das `streams`-Framework bzw. die FACT-Tools für unsere Aufgabe zunächst ungeeignet.

Die Aufgabe der PG wird von daher sein, eine Erweiterung der FACT-Tools zu implementieren, die das Verteilen von Prozessen und somit das Ausführen der FACT-Tools auf einem Cluster erlaubt. Dies würde es erlauben, die FACT-Tools zur Bearbeitung von großen Datenmengen zu nutzen. Eine solche Erweiterung besteht bereits für Apache Storm, in dieser PG soll jedoch eine Spark-Erweiterung für die FACT-Tools entwickelt werden.

Teil I

Big Data Analytics

Kapitel 2

Einführung in Big Data Systeme

Für den Begriff *„Big Data“* gibt es keine allgemeingültige Definition, vielmehr ist er ein Synonym für stetig wachsende Datenmengen geworden, die mit herkömmlichen Systemen nicht mehr effizient verarbeitet werden können. Wird nach Charakteristika von Big Data gefragt, werden oftmals die *5 Vs* [62] zitiert, die in Figure 2.1 veranschaulicht sind:

- **Volume** *(Menge)* Die Menge an Daten, die produziert werden, steigt in einen Bereich, der es für herkömmliche Systeme schwer macht, diese zu speichern und zu verarbeiten, und auch die Grenzen traditioneller Datenbanksysteme überschreitet.
- **Velocity** *(Geschwindigkeit)* Die Geschwindigkeit, mit der neue Daten generiert werden und sich verbreiten, steigt. Um diese (in Echtzeit) zu analysieren, benötigt es neue Herangehensweisen.
- **Variety** *(Vielfalt)* Die Daten stammen nicht mehr nur aus einer oder ein paar wenigen, sondern aus einer Vielzahl unterschiedlicher Quellen wie zum Beispiel Sensoren, Serverlogs und nutzergenerierten Inhalten und sind strukturiert oder unstrukturiert.
- **Veracity** *(Vertrauenswürdigkeit)* Bei der Menge an produzierten Daten kann es passieren, dass sie Inkonsistenzen aufweisen, unvollständig oder beschädigt sind. Bei der Analyse gilt es, diese Aspekte zu berücksichtigen.
- **Value** *(Wert)* Oftmals werden so viele Daten wie möglich gesammelt, um einen Gewinn daraus zu schlagen. Dieser kann beispielsweise finanzieller Natur sein oder darin bestehen, neue Erkenntnisse durch Datenanalyse für wissenschaftliche Zwecke zu gewinnen.

In erster Hinsicht besteht die Herausforderung nun darin, diese Masse an Daten auf irgendeine Art und Weise zu speichern, verfügbar und durchsuchbar zu machen und effizient zu analysieren. Die folgenden Abschnitte geben daher einen kurzen Einblick in die Anwendungsgebiete von Big Data, erläutern die Probleme mit herkömmlichen Ansätzen und beschäftigen sich mit Anforderungen an Big-Data-Systeme.

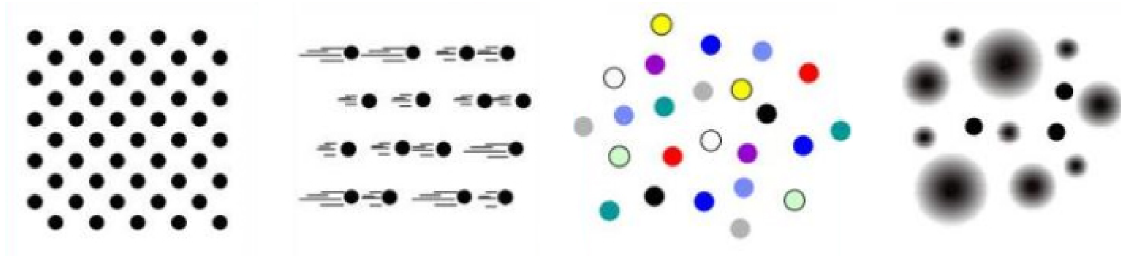


Abbildung 2.1: Veranschaulichung der ersten vier *Vs* von Big Data. Von links nach rechts: Volume, Velocity, Variety und Veracity [91]

2.1 Nutzen von Big Data

Der große Nutzen von Big Data besteht in den Ergebnissen der Datenanalyse. Diese können etwa dazu dienen, um personalisierte Werbung anzuzeigen oder wie in unserem Anwendungsfall um neue, unbekannte Daten zu erkennen und zu klassifizieren. Eine Möglichkeit der Analyse besteht in der Anwendung maschineller Lernverfahren, dessen Konzepte in chapter 4 vorgestellt werden. Im Kern geht es dabei darum, in Datensätzen Muster und andere Regelmäßigkeiten zu finden. Es liegt nahe, dass, je größer die bestehende Datenmenge ist, Modelle genauer trainiert werden können, wenn die Daten nicht höchst verschieden sind. Um große Datenmengen effizient zu analysieren, benötigt es auch hier spezielle Verfahren, die vor allem in section 4.3 angesprochen werden und entsprechende Software, die auf die Analyse von Big Data zugeschnitten ist (s. Figure 3.1.2).

2.2 Probleme mit herkömmlichen Ansätzen

Bei einer handelsüblichen Festplatte mit 2 TB Speicher und einer Lesegeschwindigkeit von im Schnitt 120 MB/s dauert alleine das Lesen der Festplatte ungefähr 4,6 Stunden. Bei noch größeren Datenmengen und zeitkritischen Analysen ist diese Zeitspanne jedoch nicht akzeptabel, weshalb Ansätze darauf abzielen, die Daten und Berechnungen auf mehrere Server zu verteilen, um nur einen Bruchteil dieser Zeit zu benötigen. Ein wichtiger Begriff in diesem Zusammenhang ist die *Skalierbarkeit*.

Skalierbarkeit beschreibt die Fähigkeit eines Systems, bestehend aus Soft- und Hardware, die Leistung durch das Hinzufügen von Ressourcen möglichst linear zu steigern. Generell unterscheidet man hierbei zwischen *vertikaler* und *horizontaler* Skalierbarkeit (s. Figure 2.2).

Unter vertikaler Skalierung spricht man dann, wenn sich eine Leistungssteigerung eines einzelnen Rechners durch mehr Ressourcen, in etwa durch mehr Arbeitsspeicher, Prozessorleistung oder Speicher, ergibt. Ein Nachteil dieses Verfahrens ist seine Kostspieligkeit, da meistens nur die Anschaffung eines neueren, leistungsstärkeren Systems möglich ist,

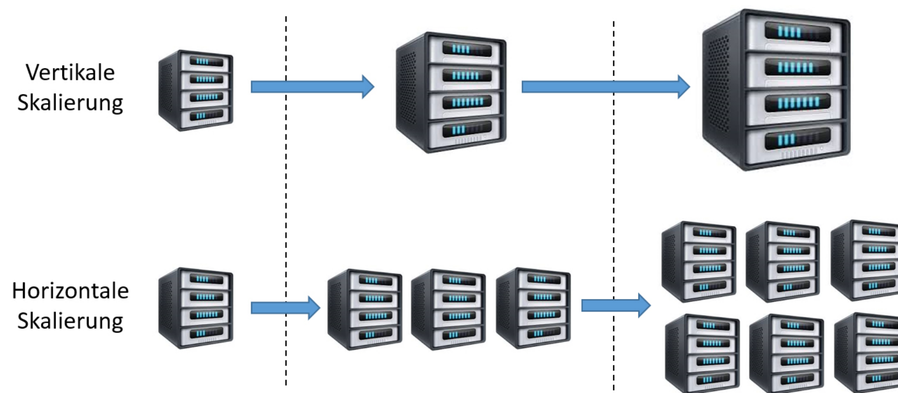


Abbildung 2.2: Arten der Skalierung

wenn das alte an seine Grenzen stößt. Fürs Big Data Processing ist diese Art der Skalierung somit eher ungeeignet, da es an irgendeinem Punkt nicht mehr möglich ist, sei es aus technischer Sicht oder aus Gründen der Kosten, mehr Ressourcen in ein System einzuspeisen. Außerdem stellen die einzelnen Komponenten eines vertikalen Systems einen *single-point-of-failure* dar. Dies bedeutet, dass der Ausfall einer Komponente den Ausfall des ganzen Systems nach sich zieht, wodurch etwa komplexe Berechnungen nicht vollendet werden können.

Im Gegensatz dazu spricht man von horizontaler Skalierung, wenn in ein bestehendes System weitere Rechner eingespeist werden. Für so einen *Cluster* wird meistens kostengünstige Serverhardware genommen, die über eine schnelle Netzwerkverbindung miteinander verbunden ist. Ein Beispiel für eine derartige, horizontal skalierbare Architektur stellt die λ-Architektur dar, die in chapter 3 thematisiert wird. In Fällen von Big Data werden horizontal skalierbare Lösungen bevorzugt, da sie kostengünstiger in der Anschaffung im Verhältnis zum Datenzuwachs sind und Ressourcen flexibel und je nach Bedarf hinzugefügt werden können [65, Kap. 1], [93]. Horizontal skalierte Systeme werden im Allgemeinen als ausfallsicher angesehen, was meistens auf die genutzte Software zurückzuführen ist. Diese sorgt dafür, dass selbst beim Ausfall von einzelnen Hardwarekomponenten, Knoten oder im schlimmsten Fall von ganzen Netzwerkpartitionen Berechnungen zum Ende gebracht werden.

2.3 Anforderungen an Big Data Systeme

Eine derartige Skalierung, wie sie im vorigen Abschnitt beschrieben ist, stellt auch neue Anforderungen an die Datenmodellierung und an die verwendete Software. Gewünschte Eigenschaften von Big-Data-Systemen sind unter anderem:

Fehlererkennung und -toleranz In einem verteilten System muss die Annahme gel-

ten, dass zufällig jede beliebige Komponente zu jedem beliebigen Zeitpunkt ausfallen kann. Mit der Anzahl an Knoten in einem Cluster steigt dieses Risiko. Kann ein solcher Fehler nicht zuverlässig erkannt werden, können Endergebnisse verfälscht oder nicht produziert werden. Infolgedessen müssen Big-Data-Systeme so konstruiert sein, dass das Ausfallrisiko oder der Verlust von Daten miteinkalkuliert ist. Um Fehlertoleranz zu gewährleisten, wird meistens auf eine Kombination aus Datenredundanz und wiederholter Ausführung von fehlgeschlagenen Teilaufgaben gesetzt. Die Fehlererkennung selbst geschieht zumeist auf algorithmischer Basis und soll hier nicht weiter vertieft werden [57, Kap. 15].

Geringe Latenzen Auch bei Datenmengen im Bereich von mehreren Tera- oder Petabyte sollen Daten so schnell wie möglich abrufbar sein. Dies wird oft über Datenredundanzen realisiert. Motiviert von der großen Varianz von Daten haben sich nicht-relationale Datenbanken (s. section 7.1, section 7.2) etabliert, die ebenfalls verteilt arbeiten, um geringe Latenzen zu garantieren.

Skalierbarkeit Mit steigender Datenmenge soll das System horizontal mitskalieren, indem mehr Ressourcen hinzugefügt werden. Entsprechende Software wie Hadoop & YARN (Figure 3.1.1) muss die neuen Ressourcen entsprechend verwalten und auf Anwendungen verteilen. Eine skalierbare Architektur für Big-Data-Systeme wird mit der λ–Architektur in chapter 3 präsentiert.

Generalisierbarkeit Ein eigen konzipiertes Big-Data-System für jeden beliebigen Anwendungsfall ist aus Sicht der Wartbarkeit und Interoperabilität nicht praktikabel. Die λ-Architektur bietet eine generelle Struktur und mit Software wie MapReduce (Figure 3.1.1) und Spark (subsection 3.1.2) lassen sich viele Probleme auf einheitlicher Basis lösen.

Bei der Datenverarbeitung in Big-Data-Systemen stellen sich neben den erwähnten Anforderungen noch weitere Herausforderungen. Etwa muss sich die Frage gestellt werden, wie Daten in einem Cluster verteilt werden, sodass sie möglichst effizient verarbeitet werden können, und wie sich vorhandene Ressourcen für diese Aufgabe möglichst gut nutzen lassen. Dies soll jedoch nicht Gegenstand dieser Projektgruppe sein, da wir auf bereits existierende Lösungen setzen, die für diese Probleme Mechanismen integriert haben.

Was uns jedoch beschäftigt, ist die Portierung von bestehenden Softwarelösungen, um genau zu sein, des `streams`-Frameworks, auf Big Data Plattformen. Dazu gilt es zum Einen, bestehenden Code so zu erweitern oder abzuändern, dass er grundsätzlich verteilt ausgeführt werden kann, und zum Anderen müssen Schnittstellen zur Ausführungsplattform *(Hadoop & Spark)* geschaffen und genutzt werden.

Kapitel 3

Lambda-Architektur

Im vorangegangenen chapter 1 wurde bereits die Herausforderung motiviert: Datenmengen in der Größenordnung von Tera- bis Petabyte müssen indiziert, angemessen verarbeitet und analysiert werden. Bisher wurde im Rahmen der Projektgruppe eine Teilmenge der Teleskopdaten auf dem verteilten Dateisystem eines Hadoop-Clusters (vgl. section 3.1) abgelegt und für die Verarbeitung herangezogen. Big-Data-Anwendungen zeichnen sich jedoch nicht nur dadurch aus, dass sie eine große Menge persistierter Daten möglichst effizient vorhalten, sodass Nutzeranfragen und damit verbundene Analysen zeitnah beantwortet werden können. Vielmehr ist auch die Betrachtung von Datenströmen ein essentieller Bestandteil einer solchen Anwendung, um eintreffende Daten in Echtzeit verarbeiten zu können. Im Folgenden soll verdeutlicht werden, wie eine solche Big-Data-Anwendung im Sinne der sog. Lambda-Architektur umgesetzt wird.

Motivation Die Problematik besteht in der Vereinigung der persistierten Datenmenge und der Daten des eintreffenden Datenstroms, der in Echtzeit verarbeitet werden soll. Auch beansprucht die Beantwortung von Anfragen auf den wachsenden Datenmengen zunehmend viel Zeit, sodass klassische Architekturansätze an ihre Grenzen kommen.

Bei der Ausführung von Transaktionen sperren relationale Datenbanken bspw. betroffene Tabellenzeilen oder die komplette Datenbank während der Aktualisierung der Daten, wodurch die Performanz und Verfügbarkeit eines Systems vorübergehend reduziert werden. Der Einfluss dieses Flaschenhalses kann mit Hilfe von Shardingansätzen reduziert werden.

Sharding beschreibt die horizontale Partitionierung der Daten einer Datenbank, sodass alle Partitionen auf verschiedenen Serverinstanzen (z.B. innerhalb eines Clusters) verteilt werden, um die Last zu verteilen. Die Einträge einer Tabelle werden somit zeilenweise auf separate Knoten ausgelagert, wodurch die Indexgröße reduziert und die Performanz deutlich gesteigert werden kann. Allerdings ist diese Methode auch mit Nachteilen verbunden. Durch den Verbund der einzelnen Knoten zu einem Cluster ergibt sich eine starke Abhängigkeit zwischen den einzelnen Servern. Die Latenzzeit wird ggf. erhöht, sobald die

Anfrage an mehr als einen Knoten im Rahmen einer Query gestellt werden muss. Insgesamt leidet die Konsistenz bzw. die Strapazierfähigkeit des Systems, da die Komplexität des Systems steigt und somit auch die Anfälligkeit gegenüber Fehlern.

Bisher wurde auf den Einsatz von Sharding verzichtet, obwohl die eingesetzten Datenbanksysteme (vgl. chapter 7) diese Methode unterstützen, da die persistierten und indizierten Event-Daten und die zugehörige Metadaten noch keine kritische Größe erreicht hatten.

Daraus resultierend ergibt sich die Notwendigkeit einer alternativen Architektur bei der Verarbeitung von besonders großen Datenmengen im Big-Data-Umfeld.

Architektur Um dem Anspruch der simultanen Verarbeitung von Echtzeitdaten und der historischen bzw. persistierten Daten gerecht zu werden, hat Nathan Marz die Lambda-Architektur [65] eingeführt, die einen hybriden Ansatz verfolgt: Es werden sowohl Methoden zum Verarbeiten von *Batches* (also den historischen Daten, vgl. section 3.1), als auch zum Verarbeiten von *Streams* (Echtzeitdaten, vgl. section 3.2) miteinander kombiniert. Durch die Anwendung von geeigneten Methoden für den entsprechenden Datensatz wird eine Ausgewogenheit zwischen der Latenzzeit (*latency*), dem Durchsatz (*throughput*) und der Fehlertoleranz (*fault-tolerance*) erreicht.

Der Unterschied zu klassischen Ansätzen beginnt bereits beim Datenmodell, welches sich durch eine unveränderliche Datenquelle auszeichnet, die lediglich durch das Hinzufügen neuer Einträge erweitert werden kann. Im vorliegenden Fall werden die Events aus den Teleskopdaten bzw. den FACT-Dateien extrahiert (vgl. chapter 6), in die Datenbank überführt und indiziert (vgl. chapter 7).

Allgemein besteht die Lambda-Architektur (Figure 3.1) aus drei Komponenten: Batch Layer (section 3.1), Speed Layer (section 3.2) und Serving Layer (section 3.3).

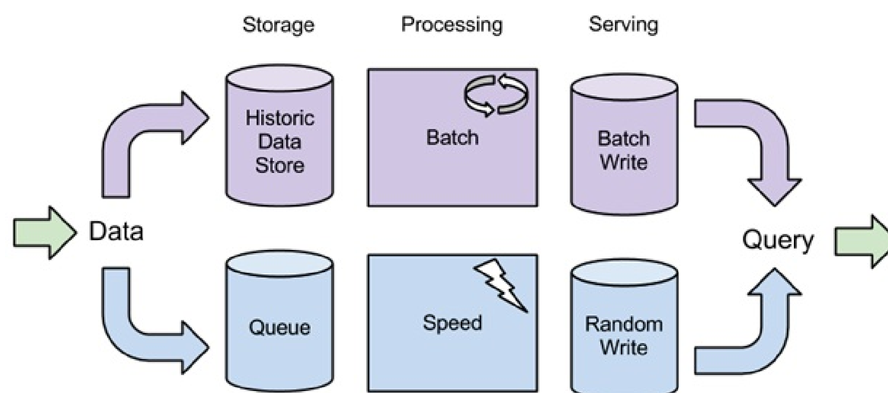


Abbildung 3.1: Lambda-Architektur [50]

Der **Batch Layer** enthält die dauerhaft gespeicherten Daten in ihrer Gesamtform. Dies sind zum Einen die auf dem Dateisystem vorliegenden Rohdaten im FITS-Format sowie die extrahierten Events und ihre zugehörigen Metadaten. Durch die große Menge an Daten, die durch diesen Layer verwaltet werden, steigen die Latenzzeiten, sodass die Performanz dieses Layers nicht besonders hoch ist. Während eine Berechnung auf diesem Datenbestand durchgeführt wird, werden neu hinzugefügte Daten bei der Berechnung nicht betrachtet. Auch werden entsprechende Ansichten auf den Datenbestand über diese Schicht erstellt und zur Verfügung gestellt. Wurden neue Daten hinzugefügt, so werden auch die entsprechenden Views aktualisiert bzw. neu berechnet.

Der **Speed Layer** verarbeitet Datenströme in Echtzeit und vernachlässigt den Anspruch des Batch Layers hinsichtlich der Vollständigkeit und Korrektheit der Ansichten auf die aktuell verarbeiteten Daten, die von dieser Schicht bereitgestellt werden. Die neu eingelesenen Daten werden temporär zwischengespeichert und stehen zur Ausführung von Berechnungen bereit. Sobald die temporär gespeicherten Daten des Speed Layers auch im Batch Layer zur Verfügung stehen, werden diese aus dem Speed Layer entfernt.

Die Komplexität des Speed Layers entsteht durch die Aufgabe, die temporär zwischengespeicherten Daten aus dem Datenstrom mit dem bereits persistierten Datenbestand des Batch Layers zusammenzuführen.

Werden neue Teleskopdaten an den Cluster übergeben, so sollen die Events in Echtzeit eingelesen und der Prozesskette hinzugefügt werden, um in den anstehenden Analysen (vgl. section 1.4) bereitzustehen.

Der **Serving Layer** dient als Schnittstelle für Abfragen, die nach erfolgter Berechnung ein Ergebnis zur Folge haben. Diese Ergebnisse werden aufgrund der hohen Latenz des Batch Layers zwischengespeichert, um das Ergebnis bei erneuter Abfrage schneller ausliefern zu können. Dabei werden die ausgewerteten Daten sowohl von Speed- als auch Batch-Layer indiziert, um die Abfragen zu beantworten.

Eine abgeschlossene Berechnung führt schließlich dazu, dass alle Daten im Serving Layer mit den neuberechneten ersetzt werden. Dadurch entfallen unnötig komplexe Updatemechanismen und die Robustheit gegenüber fehlerhaften Implementierungen werden erhöht.

Um die Events gemäß bestimmten Kriterien bereitzustellen und analysieren zu können, wird eine REST-Schnittstelle (vgl. subsection 3.3.2) zur Verfügung gestellt, über die die Anwendung u.a. auch von außerhalb angesprochen werden kann.

3.1 Batch Layer

Wie im eben beschrieben, werden im Batch-Layer mithilfe eines verteilten Systems große Mengen an Daten verarbeitet. In diesem Zusammenhang sind während der initialen Seminarphase verschiedene Technologien vorgestellt und evaluiert worden. Im Folgenden werden daher das Ökosystem um *Apache Hadoop* und *Apache Spark* vorgestellt, dessen Konzepte veranschaulicht, Vor- und Nachteile besprochen und die Wahl der später genutzten Software begründet.

3.1.1 Apache Hadoop

Bei dem Apache-Hadoop-Projekt handelt es sich um ein Open Source Framework, das Anwendern ermöglicht, schnell eine verteilte Umgebung bereitzustellen, mit der sich Hardware Ressourcen in einem Rechen-Cluster verwalten und große Mengen an Daten speichern und verteilt verarbeiten lassen.

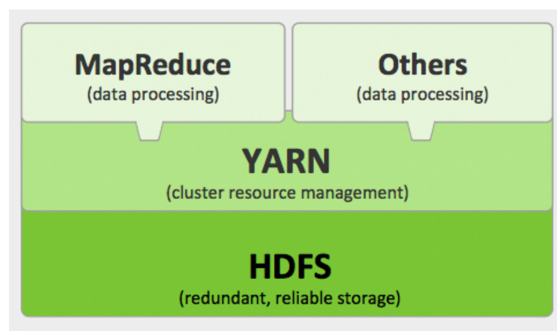


Abbildung 3.2: Architektur des Apache Hadoop Projekts [73]

Wie in Figure 3.2 zu sehen ist, setzt sich das Projekt aus drei modularen Komponenten zusammen, dessen Konzepte und Nutzen für unseren Anwendungsfall in den folgenden Abschnitten thematisiert werden.

HDFS

Für den Storage-Layer in einem Rechnercluster zeichnet sich das *Hadoop Distributed File System* (HDFS) verantwortlich und basiert auf dem Google File System [40]. Dieses eignet sich insbesondere für den Bereich des Data Warehousing, also Einsatzzwecke, wo es darauf ankommt, eine große Menge an Daten über eine lange Zeit hinweg hoch verfügbar und ausfallsicher vorzuhalten.

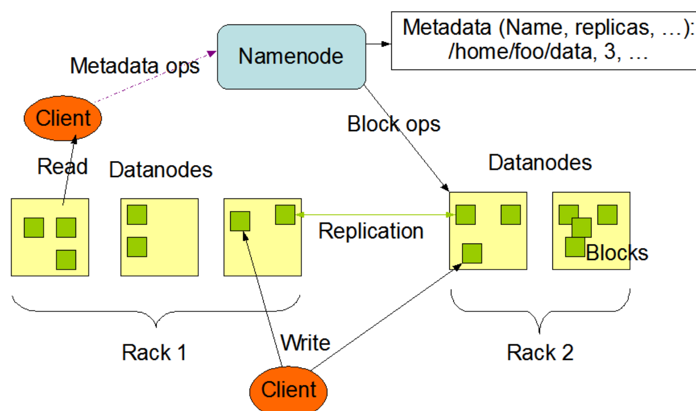


Abbildung 3.3: Funktionsweise eines HDFS Clusters

Der Aufbau eines HDFS-Clusters ist in Figure 3.3 illustriert. Wie zu erkennen ist, werden Daten auf sogenannten *Datanodes* in gleich großen *Blocks* gespeichert. Um Ausfallsicherheit zu garantieren, besitzt das System einen Replikationsmechanismus, bei dem Blocks bei Bedarf mehrfach redundant (bestimmt durch einen Replikationsfaktor) auf verschiedenen Datanodes und Racks gespeichert werden. Im Falle eines Ausfalls kann so der Replikationsfaktor von betroffenen Blöcken durch Neuverteilung im Cluster wiederhergestellt werden, vorausgesetzt die nötigen Kapazitäten sind vorhanden.

Beim *Namenode* handelt es sich um eine dedizierte Einheit, auf der keine Daten gespeichert werden. Dieser enthält Informationen über den Zustand des Systems, was das Wissen über den Aufenthaltsort von Blöcken und dessen Replikationen im Cluster beinhaltet. Durch einen periodisch ausgeführten *Heartbeat* werden alle Datanodes kontaktiert und aufgefordert, einen Zustandsbericht über gespeicherte Daten zu senden. Schlägt ein Heartbeat mehrmals fehl, gilt der Zielknoten als tot und der beschriebene Replikationsmechanismus greift ein. Darüber hinaus kann der Namenode selbst repliziert werden, da er sonst einen *single-point-of-failure* in diesem System darstellt.

Der Zugriff auf Daten von einem Klienten geschieht je nach dem, welche Operation ausgeführt werden soll. Bei Leseoperationen einer Datei wird zunächst der Namenode angefragt, da dieser über ein Verzeichnis über alle Daten im Cluster verfügt. Dieser gibt dann den Ort der angefragten Datei an. Schreiboperationen werden typischerweise direkt auf den Datanodes durchgeführt. Mittels der Heartbeats wird der Namenode schließlich von den Änderungen informiert und veranlasst die Replikation der neu geschriebenen Daten. Weiterhin wird für Klienten eine einfache Programmierschnittstelle angeboten, die die Verteilung der Daten nach außen hin abstrahiert und somit wie ein einziges Dateisystem wirkt.

Für die Projektgruppe wurde zu Anfang ein aus sechs Rechnern bestehendes Hadoop-Cluster vom LS8 mit dem HDFS zur Verfügung gestellt. Im zweiten Semester ist das

Projekt auf ein größeres Cluster zwecks mehr Speicherkapazität und Rechenleistung umgezogen. Das Dateisystem kommt in unserem Anwendungsfall hauptsächlich für die Persistenz der in chapter 6 beschriebenen Teleskopdaten zum Einsatz. Das verteilte Dateisystem erwies sich bereits als sehr zuverlässig in Bezug auf Ausfallsicherheit [40] und wird in Produktivsystemen zum Speichern und Verarbeiten mehrerer Petabyte genutzt, womit es eine solide Grundlage für den Anwendungszweck darstellt.

YARN

Yet Another Resource Allocator (YARN) wirkt als Mittelsmann zwischen dem Ressourcenmanagement im Cluster und den Anwendungen, die gegebene Ressourcen für Berechnungen nutzen möchten. Die Architektur setzt sich aus einem dedizierten *RessourceManager* (RM) und mehreren *NodeManager* (NM) zusammen, wobei auf jedem Rechner im Cluster ein NM läuft. Der RM stellt Anwendungen Ressourcen als sogenannte Container, also logische, auf einen Rechner bezogene Recheneinheiten zur Verfügung, die den Anforderungen der Anwendung, wenn möglich, entsprechen. Ein von der Anwendung eingereichter Job wird dann im Container verarbeitet. Nach Beendigung gibt der RM die Ressourcen wieder frei.

Aufgrund dieser offenen Struktur sind Ressourcen in einem Hadoop-Cluster nicht nur für Software aus dem selben Ökosystem zugänglich, sondern können auch von Dritt-Programmen wie *Apache Spark* und *Apache Storm* reserviert und genutzt werden [89].

MapReduce

Bei *Hadoop MapReduce* handelt es sich um eine YARN-basierte Umgebung zum parallelen Verarbeiten von Datenmengen in einem Hadoop-Cluster. Die Idee basiert auf einem Verfahren aus der funktionalen Programmierung, bei der es eine *map* und eine *reduce* Funktion gibt. Erstere wird auf jedes Element einer Menge unabhängig voneinander durchgeführt, die errechneten Ergebnisse mit letzterer Funktion zusammengeführt. MapReduce macht sich insbesondere die Unabhängigkeit der Daten zunutze, um beide Funktionen massiv parallel auszuführen, sodass sich folgendes Verfahren ergibt:

$$(k_1, v_1) \xRightarrow{map} list(k_2, v_2) \xRightarrow{group} (k_2, list(v_2)) \xRightarrow{reduce} list(v_2).$$

Um das Prinzip zu veranschaulichen, kann das Zählen von Events pro Nacht benutzt werden. Rechner, die einen map-Job ausführen (*Mapper*), erhalten als Eingabe jeweils eine fits-Datei (s. chapter 6), zählen die Events und speichern jeweils eine Liste $list(night, 1)$ als Zwischenergebnis ab. MapReduce gruppiert die Zwischenergebnisse aller Mapper, was zu einer Menge von $(night_i, list(1, 1, \dots))$ führen würde. Rechner, die für den reduce-Funktion

ausgewählt worden sind (*Reducer*), würden die Zwischenergebnisse zusammenführen und Daten der Form $(night_i, n_i)$ abspeichern, wobei n_i die Anzahl aufgenommener Events der Nacht $night_i$ beschreibt. Es ist anzumerken, dass ,selbst wenn einer der Jobs fehlschlagen sollte, der gesamte Prozess nicht abgebrochen, sondern der entsprechende Job ggf. auf einem anderen Rechner erneut ausgeführt wird. Die Erkennung eines toten Knotens geschieht durch ständige Statusanfragen des Masters an Mapper und Reducer. In Experimenten zeigte sich, dass dieses Prinzip eine hohe Wahrscheinlichkeit für die Terminierung aufweist [26].

Hadoop MapReduce hat in der Projektgruppe keine Anwendung gefunden, wofür sich zwei Gründe angeben lassen. Zum Einen haben direkte Vergleiche gezeigt, dass andere Frameworks wie Apache Spark Vorteile bezogen auf die Performance haben, was auch darauf zurückzuführen ist, dass bei MapReduce viele Lese- und Schreibzugriffe auf das Speichermedium ausgeführt werden, anstatt Daten im Arbeitsspeicher vorzuhalten. Weiterhin gestaltet sich die Suche nach einem MapReduce basierten Framework zum verteilten, maschinellen Lernen als schwierig. Zwar existiert mit *Apache Mahout* eine entsprechende, ausgereifte Lösung, nach Angaben der Entwickler wird die Entwicklung des Frameworks sich jedoch aus Gründen der Performance auf Apache Spark konzentrieren.

3.1.2 Apache Spark

Bei Apache Spark handelt es sich um ein Cluster Computing Framework, mit dessen Hilfe Aufgaben auf mehrere Knoten eines Clusters (Rechnerverbunds) verteilt und somit parallel verarbeitet werden können. Dies hat einen deutlichen Geschwindigkeitsvorteil gegenüber der Berechnung auf einem einzelnen Knoten zur Folge, was insbesondere bei der Verarbeitung großer Datenmengen deutlich wird. Im Gegensatz zu Apache Hadoop setzt Apache Spark auf die Vorhaltung und Verarbeitung der Daten im Hauptspeicher und erzielt so einen Perfomancevorteil, durch den Berechnungen bis zu hundertmal schneller durchgeführt werden können [94].

Das Framework setzt sich grundlegend aus vier Komponenten zusammen: Spark Core, Spark SQL, Spark Streaming, GraphX, sowie der MLlib Machine Learning Library. Mit diesen Komponenten werden somit die essentielle Bestandteile des Projekts (Clustering, Querying, Streaming und Datenanalyse) prinzipiell abgedeckt, sodass Apache Spark eine besonders interessante Option als Systemgrundlage darstellt. Ebenso wird eine Vielzahl an verteilten Dateisystemen unterstützt, wodurch die Anbindung des Frameworks an verschiedene Datenquellen erheblich vereinfacht wird.

Spark Core

Spark Core bildet die Grundlage von Apache Spark und ist mitunter für die folgenden Aufgaben verantwortlich: Speichermanagement, Fehlerbeseitigung, Verteilung der Aufgaben an die einzelnen Knoten, das Prozessscheduling und die Interaktion mit verteilten Dateisystemen.

Ferner definiert Spark Core die Programmierschnittstelle, um auf dem Cluster zu arbeiten und Aufgaben zu definieren. Dabei handelt es sich um sog. *resilient distributed datasets* (kurz: RDDs), die wiederum Listen von einzelnen Elementen repräsentieren, deren Partitionen auf die einzelnen Knoten verteilt und parallel auf allen Knoten manipuliert werden können, wie es in Abbildung 3.4 ersichtlich wird. Die Verteilung und die parallele Ausführung der Operationen wird dabei vom Framework selbst übernommen. Dies ist ein weiterer Vorteil von Apache Spark: Ursprünglich komplexe Aufgaben wie das Verteilen und parallele Ausführen von Prozessen auf mehreren Knoten werden durch das Framework vollkommen abstrahiert und somit stark vereinfacht.

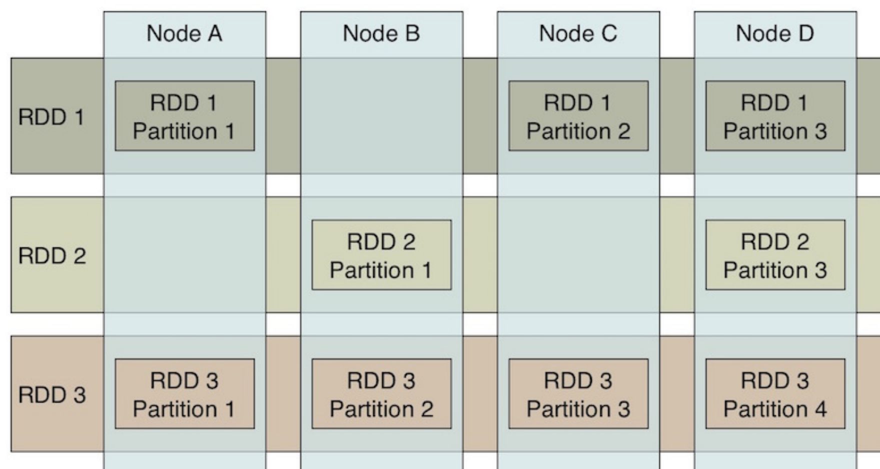


Abbildung 3.4: Verteilung der Partitionen eines RDDs auf unterschiedliche Knoten [92]

Die Daten können zum einen, wie bereits erwähnt, aus statischen Dateien eines (verteilten) Dateisystems bezogen werden oder aber auch aus anderen Datenquellen wie Datenbanken (MongoDB, HBase, ...) und Suchmaschinen wie Elasticsearch.

Es wird zwischen zwei Arten von Operationen unterschieden, die auf den RDDs ausgeführt werden können. *Transformationen* (wie das Filtern von Elementen) haben ein neues RDD zur Folge, auf dem weitere Operationen ausgeführt werden. Transformationen werden jedoch aus Gründen der Performanz nicht direkt ausgeführt, sondern erst wenn das finale Ergebnisse nach einer Reihe von Transformationen ausgegeben werden soll. Diese Technik wird *Lazy Evaluation* genannt und bietet den Vorteil, dass die Kette von Transformatio-

nen zunächst einmal vom Framework sinnvoll gruppiert werden kann, um die Scans des Datensatzes zu reduzieren. *Aktionen* berechnen (wie das Zählen der Elemente in einem RDD) ein Ergebnis und liefern dieses an den Master Node zurück oder halten es in einer Datei auf einem verteilten Dateisystem fest.

Spark SQL

Spark SQL unterstützt die Verarbeitung von SQL-Anfragen, um sowohl die Daten der RDDs als auch die externer Quellen in strukturierter Form zu manipulieren. Dadurch wird nicht nur die Kombinationen von internen und externen Datenquellen (JSON, Apache Hive, Parquet, JDBC (und somit u.a. MySQL und PostgreSQL), Cassandra, ElasticSearch, HBase, u.v.m.) erleichtert, sondern ebenfalls die Persistierung von Ergebnissen, Parquet-Dateien oder Hive-Tabellen und somit die Zusammenführung mit anderen Daten ermöglicht.

Eine zentrale Komponente ist das *DataFrame*, welches an das *data frame*-Konzept aus der Programmiersprache R anlehnt und die Daten wie in einer relationalen Datenbank in einer Tabelle bestehend aus Spalten und Zeilen repräsentiert. Dabei wird dieses DataFrame analog zu den RDDs dezentral auf die bereitstehenden Knoten verteilt. Analog zu den RDDs können auf den DataFrames Transformationen, wie `map()` und `filter()` aufgerufen werden, um die Daten zu manipulieren. Technisch gesehen besteht ein DataFrame auf mehreren `Row`-Objekten, die zusätzliche Schemainformationen wie z.B. die verwendeten Datentypen für jede Spalte enthalten.

Hinsichtlich der Performance schickt sich Spark SQL an, aufgrund der höheren Abstraktion durch SQL und den zusätzlichen Typinformationen besonders effizient zu sein.

Spark MLlib

Da Apache Spark nicht nur zum Ziel hat, Daten effizient zu verteilen, sondern diese auch zu analysieren, existiert die Bibliothek *MLlib* als weitere Komponente, um Algorithmen des maschinellen Lernens auf den eingelesenen Daten ausführen zu können. Dabei werden prinzipiell nur Algorithmen angeboten, die auch dafür ausgelegt sind, verteilt zu arbeiten.

Allgemein existieren mehrere Arten von Lernproblemen wie Klassifikation, Regression oder Clustering, deren Lösungen verschiedene Ziele verfolgen. Alle Algorithmen benötigen eine Menge an Merkmalen für jedes Element, das dem Lernalgorithmus zugeführt wird. Betrachtet man beispielsweise das Problem der Identifizierung von Spamnachrichten, das eine neue Nachricht als Spam oder Nicht-Spam klassifizieren soll, so könnte ein Merkmal z.B. der Server sein, von dem die Nachricht versandt wurde, die Farbe des Texts und wie oft bestimmte Wörter verwendet wurden.

Die meisten Algorithmen sind darauf ausgelegt, lediglich numerische Merkmale zu betrachten, sodass die Merkmale in entsprechende numerische Werte übersetzt beziehungsweise in entsprechende Vektoren transformiert werden müssen.

Mithilfe dieser Vektoren und einer mathematischen Funktion wird schlussendlich ein Modell berechnet, um neue Daten zu klassifizieren. Zum Trainieren des Modells wird der bestehende und bereits klassifizierte Datensatz in einen Trainings- und Testdatensatz aufgeteilt. Mit Ersterem wird das Modell trainiert und mit Letzterem schließlich die Vorhersage evaluiert, wie es in Abbildung 3.5 dargestellt wird.

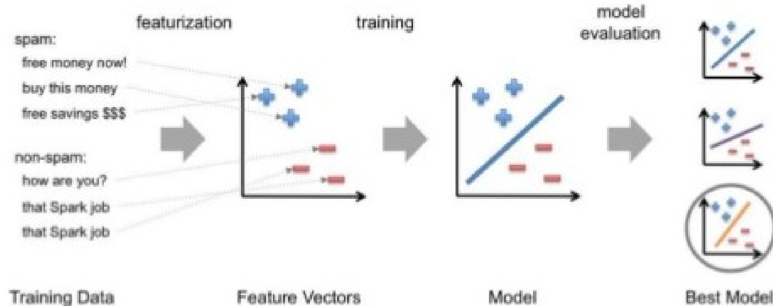


Abbildung 3.5: Maschinelles Lernen mit Spark MLlib [87]

Mithilfe der von MLlib bereitgestellten Klassen können die Schritte zum Lösen eines Lernproblems in einer Apache-Spark-Applikation nachvollzogen werden und die Algorithmen darauf trainiert werden. Auch zur Evaluierung der Vorhersage stellt MLlib entsprechende Methoden zur Verfügung.

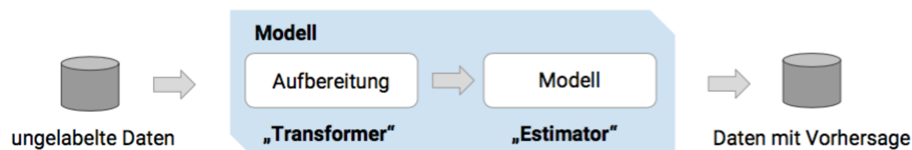


Abbildung 3.6: Pipeline-Struktur von Spark ML

Die MLlib-Bibliothek gliedert sich in zwei Pakete: *spark.mllib* ist das ursprüngliche Paket, welches auf Basis der zuvor vorgestellten RDDs arbeitet. Es wird nicht mehr weiterentwickelt, allerdings noch unterstützt. *spark.ml* ist die neue Version, die aktuell weiterentwickelt wird. Das Paket arbeitet auf Basis von den in Spark SQL eingeführten DataFrames. Außerdem werden alle Arbeitsschritte in einer *Pipeline* zusammengefasst. Eine solche Pipeline besteht aus *Stages*, welche sequentiell ausgeführt werden. Daten werden also von

Stage zu Stage gereicht. Eine Stage kann ein *Transformer* oder ein *Estimator* sein. Ein Transformer implementiert die `transform()`-Methode, welche einen gegebenen DataFrame verändert. Beispiele für typische Transformer ist die Merkmalsselektion oder die Klassifikation. Ein Estimator implementiert die `fit()`-Methode, welche ein Modell auf Basis eines DataFrames trainiert. Das Konzept einer solchen Pipeline ist in Abbildung 3.6 dargestellt, ein konkretes Beispiel liefert Abbildung 3.7. Ein Dokument soll in Worte zerlegt werden, welche dann in numerische Merkmale überführt werden. Anschließend soll ein Modell mit Hilfe der logistischen Regression trainiert werden. Die Transformer sind blau umrandet, der Estimator rot. Generell kann es auch mehrere Estimator in einer Pipeline geben. Jeder dieser Lerner wird in der Pipeline auf den Trainingsdaten trainiert. Wird auf dem Modell, welches aus einer solchen Pipeline hervorgeht, klassifiziert, erhält der Anwender auch mehrere Klassifikationsergebnisse, nämlich genau eines pro Lerner.

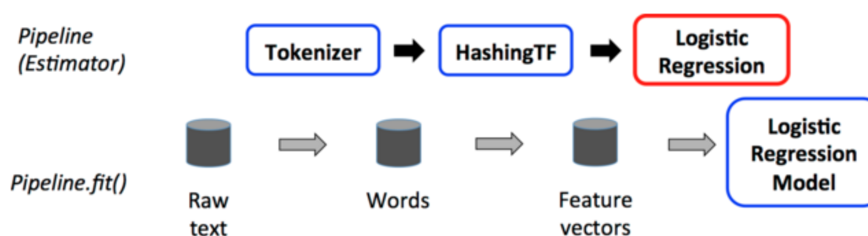


Abbildung 3.7: Konkretes Beispiel für eine Pipeline in Spark ML [6]

Spark MLlib besteht insgesamt aus zwei Paketen: Die ältere Version MLlib, die der Bibliothek den Namen gab, und die neuere Version ML, welche aktuell auch weiterentwickelt wird. Zu Beginn der Projektgruppe erfolgten einige Experimente, in welchen die Eignung der beiden Pakete für unsere Zwecke untersucht wurde. Das Ergebnis lässt sich in chapter 10 nachlesen. Dort wird außerdem beschrieben, wie die Bibliothek letztendlich in unsere Software integriert wurde.

3.2 Speed Layer

Im Unterschied zum Batch Layer wird mittels eines Speed Layers versucht, sich der echtzeitlichen Datenanalyse zu approximieren. Neu eintreffende Daten sollen dabei unmittelbar nach ihrer Ankunft verarbeitet und an den bzw. die Klienten weitergeleitet werden.

Im Rahmen dieser Projektgruppe wurden Informationen zu gängigen Werkzeugen, die für die realzeitliche Verarbeitung von Datenströmen infrage kommen, gesammelt, um somit Stück für Stück den Speed Layer zu entwickeln.

3.2.1 Apache Storm

Apache Storm ist ein Open-Source-Tool, welches zur realzeitlichen Analyse von Daten genutzt werden kann.

Abbildung 3.8 zeigt eine Übersicht der in Storm vorhandenen Komponenten, `Spouts` und `Bolts`, welche an späterer Stelle näher betrachtet werden. Storm-Aufgaben werden über gerichtete, azyklische Graphen spezifiziert. Dabei werden die `Spouts` und `Bolts` als Knoten realisiert und die Kanten als Datenstreams zwischen den Knoten. Derartige Aufgaben werden in Storm als Topologie bezeichnet.

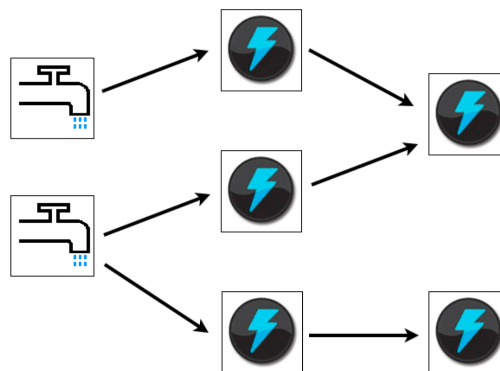


Abbildung 3.8: Beispiel einer Storm Topologie als DAG. Zu sehen sind `Spouts` (links, erste Ebene) und `Bolts` (rechts, ab zweite Ebene) [63]

Storm Topologien

Wie bereits erwähnt, handelt es sich bei Topologien um die Spezifikationen von Storm-Aufgaben in Graphenform. Sie bestehen aus zwei Knotentypen sowie einer Menge von Kanten, die als Datenstreams zu verstehen sind und eine endlose Sequenz von Tupeln

darstellen. Figure 3.8 zeigt eine solche Beispieltopologie. In diesen Abschnitt werden die Komponenten nochmals näher betrachtet.

Spout Ein `Spout` realisiert eine Quelle für Datenstreams und liest im Wesentlichen Eingaben ein, welche er im Anschluss in Form von Datenstreams an die folgenden Knoten weitergibt. `Spouts` können als *reliable* oder *unreliable* markiert werden, wodurch das Verfahren im Falle eines Lesefehlers festgelegt wird. Wie in Figure 3.8 zu sehen ist, kann ein `Spout` auch mehr als einen Stream erzeugen.

Bolt Ein `Bolt`-Knoten dient zur Verarbeitung der Daten in Storm. Ähnlich zum MapReduce Ansatz können über `Bolts` Filterung, Funktionen, Aggregationen, Joins usw. durchgeführt werden. `Bolts` können mehrere Streams einlesen, aber auch ausgeben.

Storm-Cluster

Ein Storm-Cluster hat Ähnlichkeit mit einem Hadoop-Cluster (siehe subsection 3.1.1), unterscheidet sich aber in der Ausführung. Auf Hadoop werden MapReduce Aufgaben verarbeitet, wohingegen in Storm Topologien ausführt werden. Die Konzepte unterscheiden sich vor allem darin, dass MapReduce Aufgaben irgendwann enden müssen. Storm-Topologien werden solange ausgeführt, bis von außen ein „Stopp“ (*kill*) gesendet wird.

Knoten im Cluster Innerhalb eines Storm-Clusters existieren zwei Typen von Knoten: `Master Node` und `Worker Node`. Figure 3.9 stellt den Aufbau eines solchen Clusters dar.

Master Node Der `Master Node` ist verantwortlich für die Verteilung des Codes, die Fehlerüberwachung und die Aufgabenverteilung. Zu diesem Zweck läuft im Hintergrund ein Programm namens *Nimbus*.

Worker-Knoten Die `Worker Nodes` führen die eigentliche Arbeit aus. Worker sind verteilt auf mehrere Maschinen und führen immer Teile einer Topologie aus. Auf diese Weise kann eine Topologie auf mehreren Worker verteilt abgearbeitet werden. Auf jedem `Worker Node` läuft ein *Supervisor* Daemon.

Zookeeper Zwischen Master-Knoten und Worker-Knoten gibt es einen Koordinator, der *Zookeeper* genannt wird. Alle Zustandsinformationen werden im Zookeeper gespeichert, sodass es möglich ist, einen laufenden Nimbus oder Supervisor zu stoppen, ohne dass das ganze Programm angehalten werden muss. Gleichzeitig können die Daemons erneut gestartet werden und mit ihrer Arbeit von Neuem beginnen.

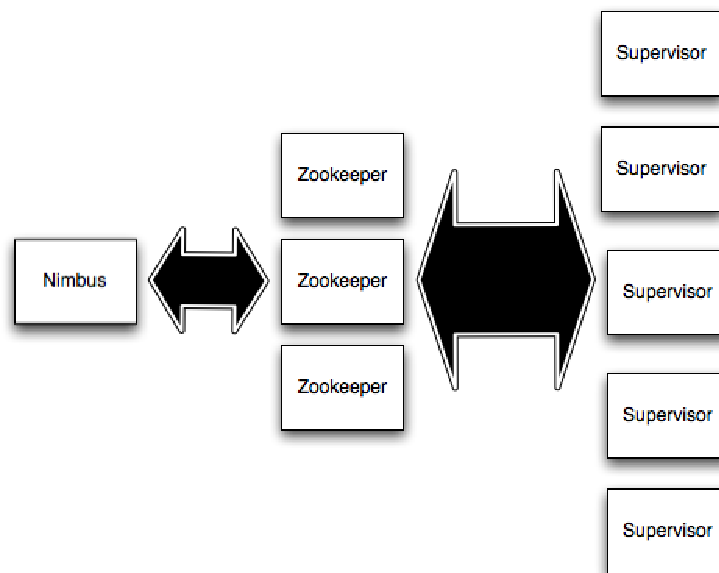


Abbildung 3.9: Aufbau eines Storm Clusters [63]

3.2.2 Apache Trident

Trident ist eine High-Level-Abstraktion auf Basis von Storm und kann als Alternative zum Storm-Interface verwendet werden. Es ermöglicht die Verarbeitung von vielen Daten sowie die Verwendung von zustandsbasierter Datenstreambearbeitung. Im Unterschied zu Storm erlaubt Trident eine *exactly-once*-Verarbeitung, transaktionale Datenpersistenz und eine Reihe von verbreiteten Operationen auf Datenstreams, welche sich in 5 Kategorien unterteilen lassen:

- lokale Operationen ohne Netzwerkbelastung
- Repartitionierung der Daten über das Netzwerk
- Aggregation als Teil einer Operation mit Netzwerkbelastung
- Gruppierung
- Merges und Joins

Trident-Topologien

Trident-Topologien werden mittels eines Compilers in optimale Storm-Topologien kompiliert. Figure 3.10 zeigt eine Trident-Topologie, welche mit zwei Datenstreams, also bereits aus Storm bekannten `Spouts`, initialisiert wird. Diese werden über lokale Operationen (hier `each`) bearbeitet und anschließend gruppiert bzw. partitioniert. Der obere Stream wird anschließend in einen Zustand persistiert, sodass der untere Stream aus Queries Infor-

mationen des oberen erhalten und mitverarbeiten kann. Zudem ist zu sehen, dass mehrere Streams über den `join`-Operator miteinander kombiniert werden können.

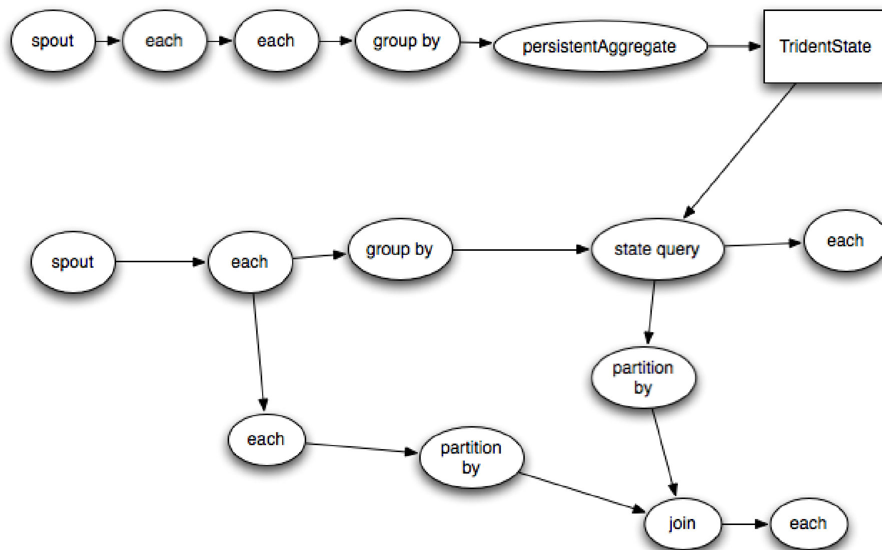


Abbildung 3.10: Beispielhafte Trident Topologie [64]

Figure 3.11 stellt die kompilierte Storm-Topologie dar. Dabei werden die Datenstreams wieder als die bekannten `Spouts` initialisiert. Damit die kompilierte Topologie maximal optimiert wird, müssen Datenübertragungen nur stattfinden, wenn Daten über das Netzwerk übertragen werden. Aufgrund dessen wurden lokale Operationen in `Bolts` zusammengefasst. Die Gruppierung und die Partitionierung der Daten sind daher als Teil der Kanten in der Storm-Topologie und somit als Datenströme zu interpretieren.

3.2.3 Spark Streaming

Als Datenstrom wird ein kontinuierlicher Fluss von Datensätzen bezeichnet, dessen Ende nicht abzusehen ist. Die Daten werden verarbeitet, sobald sie eintreffen, wobei die Größe der Menge an Datensätzen, die pro Zeiteinheit verarbeitet wird, nicht festgelegt ist. Datenströme unterscheiden sich von statischen Daten insofern, als dass die Daten in fester, zeitlich vorgegebener Reihenfolge eintreffen und nicht an beliebiger Stelle manipuliert werden können. Die Datenströme werden also nur Satz für Satz fortlaufend (sequentiell) verarbeitet und lediglich bei ihrem Eintreffen um neue Informationen erweitert.

Mit *Spark Streaming* steht eine Komponente zur Verarbeitung innerhalb des Apache Spark Framework bereit, die eine *Micro-Batch Architektur* implementiert: Streams werden als

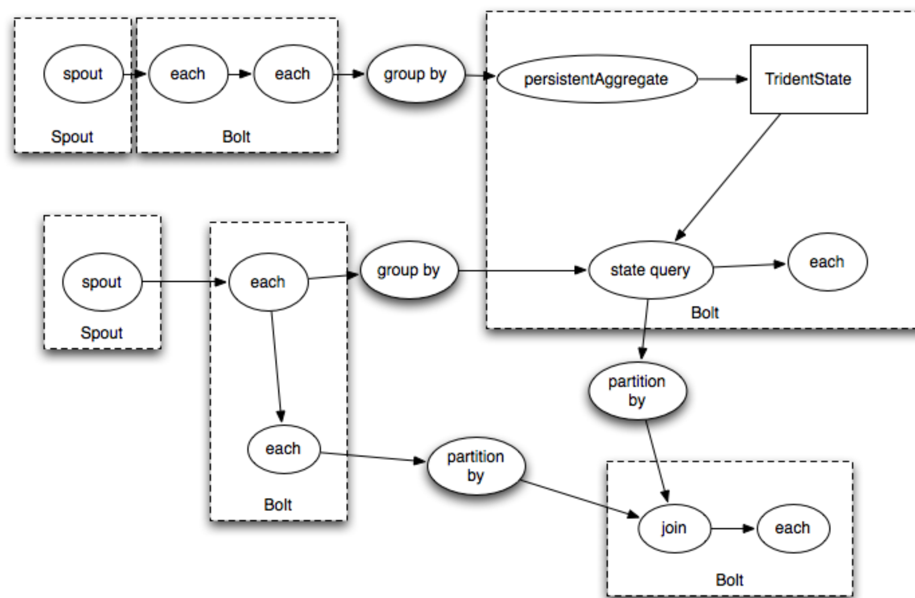


Abbildung 3.11: Figure 3.10 als kompilierte Storm Topologie [64]

eine kontinuierliche Folge von Batchberechnungen aufgefasst, wie es in Abbildung 3.12 dargestellt wird. Neue *Batches* werden immer in regelmäßigen Abständen erstellt und alle Daten, die innerhalb eines solchen Intervalls ankommen, werden dem Batch hinzugefügt. Bei den Batches handelt es sich um die bereits im Abschnitt 3.1.2 eingeführten RDDs.

Spark Streaming unterstützt verschiedenste Eingangsquellen (z.B. Flume, Kafka, HDFS), für die sog. *receiver* gestartet werden, die die Daten von diesen Eingangsquellen sammeln und in RDDs speichern. Im Sinne der Fehlertoleranz wird das RDD im Anschluss auf einen weiteren Knoten repliziert und die Daten werden im Speicher des Knotens zwischengespeichert, wie es auch bei gewöhnlichen RDDs der Fall ist. In periodischen Abständen wird schließlich ein Spark Job gestartet, um diese RDDs zu verarbeiten und mit den vorangegangenen RDDs zu konkatenieren.

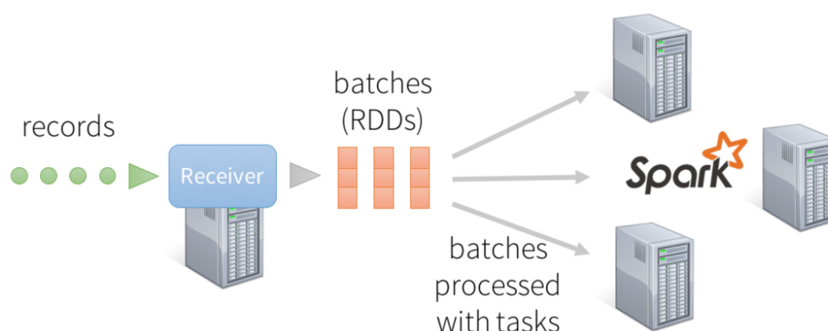


Abbildung 3.12: Verarbeitung von Datenströmen zu Batches (Quelle: https://databricks.com/blog/2015/07/30/diving-into-spark-streamings-execution-model.html)

Auf technischer Ebene baut Spark Streaming auf dem Datentyp *DStream* auf, der eine Folge von RDDs über einen bestimmten Zeitraum kapselt, wie es in der Abbildung 3.13 veranschaulicht wird. Ähnlich wie bei den RDDs können DStreams transformiert werden, woraus neue DStream Instanzen entstehen. Oder es werden die bereitstehenden Ausgabeoperationen genutzt, um die Daten zu persistieren.

Abbildung 3.13: DStream als Datentyp zur Kapselungen von RDDs (Quelle: http://www.slideshare.net/frodriguezolivera/apache-spark-streaming)

Um die eingegangenen Daten zu verarbeiten, stehen zwei Arten von Transformationen zur Verfügung. Mit den zustandslosen Transformationen werden die üblichen Transformationen wie Mapping oder Filtern bezeichnet. Diese Transformationen werden auf jedem RDD ausgeführt, das von dem betreffenden DStream gekapselt wird. Die zustandlose Transformierung ist unahängig von dem vorangegangen Batch, wodurch sie sich von der zustandbehafteten Transformierung unterscheidet. Die zustandsbehaftete Transformation hingegen baut auf den Daten des vorangegangenen Batches auf, um die Ergebnisse des aktuellen Batches zu berechnen. Es wird zwischen zwei Typen von Transformationen unterschieden: *Windowed Transformations* und *UpdateStateByKey Transformation*.

Bei den *Windowed Transformations* wird ein Zeitintervall betrachtet, das über die zeitliche Länge eines Batches hinausgeht. Es wird also ein Fenster festgelegt, das eine gewisse Anzahl an Batches umfasst, sodass die entsprechende Berechnung auf den Batches in diesem Fenster ausgeführt wird. Dieses Fenster wiederum wird immer um ein bestimmtes Verschiebungsintervall verschoben und die Berechnung erneut ausgeführt.

Die *UpdateStateByKey Transformation* dient dazu, einen Zustand über mehrere Batches hinweg zu erhalten. Ist ein DStream bestehend aus (Schlüssel,Event) Tupeln gegeben, so kann mit dieser Transformation ein DStream bestehend aus (Schlüssel,Zustand) Tupeln erzeugt werden. Dabei wird, ähnlich wie bei der *ReduceByKey* Operation, eine Funktion übergeben, die definiert, wie der Zustand für jeden Schlüssel aktualisiert wird, wenn ein neues Event eintritt.

Ein Beispiel hierfür wären Seitenbesuche als Events und eine Session- oder Nutzer-ID als Schlüssel, über den die Seitenbesuche aggregiert werden. Die resultierende Liste bestünde aus den jeweiligen Zuständen für jeden Nutzer, die wiederum die Anzahl der besuchten Seiten reflektieren würden.

Spark Streaming stellt demnach ein mächtiges Tool zur Verarbeitung von Datenströmen

dar und integriert sich nahtlos in eine bestehende Apache Spark Applikation. Durch die Unterstützung verschiedenster Datenquellen, insbesondere dem verteilten Dateisystem HDFS, bietet es sich insbesondere zur Verarbeitung von eingehenden Events in Echtzeit an.

3.2.4 `streams`-Framework

Das `streams`-Framework [16] ist eine in Java entwickelte Bibliothek, welche eingesetzt werden kann, um Datenströme zu verarbeiten. Die Verarbeitung der Daten wird über Prozesse geregelt, welche unter anderem für das Klassifizieren der Daten eingesetzt werden können. Dafür wurde das existierende Softwarepaket Massive Online Analysis (MOA) [12] integriert und ein Plugin für RapidMiner entwickelt.

Prozesse werden in `streams` über eine XML Datei spezifiziert. Es können auch eigene Prozesse in Java geschrieben und für die Verarbeitung verwendet werden. Die grundlegenden Elemente von `streams` sind `<container>`, `<stream>`, `<process>` und `<service>`.

Der *Container* ist der Vater aller weiteren Elemente und definiert den eigentlichen `stream` Prozess. Nur Elemente innerhalb eines *Container* werden ausgeführt.

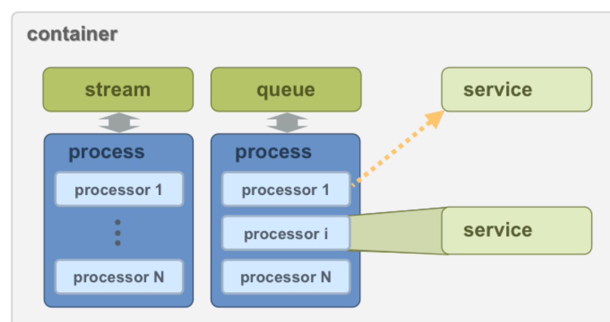


Abbildung 3.14: Schematischer Aufbau eines *Container* [15]

Der *Stream* wird genutzt, um die Quellen der Daten zu definieren. Ein Stream liest einen Strom von Daten, welcher dann beispielsweise an Prozesse weitergegeben werden kann.

Das *Process* Element besteht aus einer Reihe von Prozessoren, welche den Strom von Daten abarbeiten. Dafür wird der Strom in Datenpakte aufgeteilt, welche nacheinander durch Prozessoren geschoben werden. Prozessoren können die einzelnen Datenpakte lesen, verändern oder komplett neue erstellen und an die nächsten Prozessoren weitergeben.

Service Elemente erlauben das Abrufen von Funktionen in jeder Phase der Verarbeitung. Ein *Service* kann so z.B. dafür eingesetzt werden, um innerhalb eines Prozessors Datenbankanfragen zu stellen.

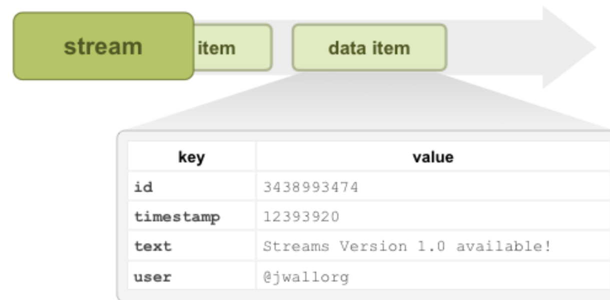


Abbildung 3.15: Funktionsweise eines *Stream* [15]

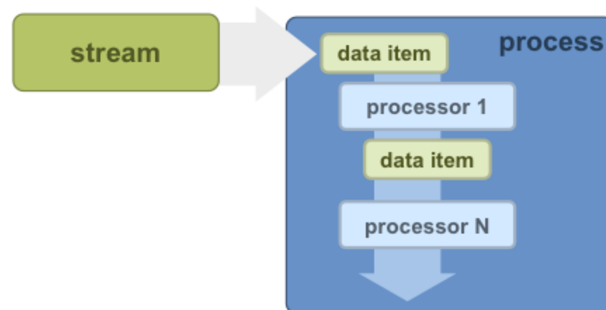


Abbildung 3.16: Arbeitsschritte eines *Process* [15]

3.3 Serving Layer

Die letzte Schicht der in diesem Kapitel beschriebenen *Lambda-Architektur* ist der *Serving Layer*. Während der *Batch Layer* und der *Speed Layer* sich vor allem um die Verarbeitung der Daten gekümmert haben, übernimmt diese Schicht die Kommunikation mit den Nutzern. Die zugrundeliegenden Daten werden dazu üblicherweise indexiert und gewonnene Ergebnisse aus den anderen Schichten werden (zwischen-)gespeichert, damit auch größere Datenmengen und komplexere Anfragen den Anwendern schnell zur Verfügung gestellt werden können.

Hierzu werden in diesem Abschnitt verschiedene Datenbank-Systeme präsentiert, wobei ein Schwerpunkt auf sogenannte „Not only SQL (NoSQL)“-Systeme gelegt wird. Weiterhin wird das Prinzip eines Service-Interfaces mithilfe einer RESTful API erörtert.

3.3.1 Datenbanken

Für eine spätere Anwendung, die die vom Teleskop erzeugten Daten verarbeiten soll, ist nicht pragmatisch, jedes Mal die Daten aus den einzelnen Dateien auszulesen. Daher bietet es sich an, die häufig benötigten Daten in einer Datenbank zu erfassen.

Die verwendete Datenbank muss mit großen Datenmengen zurechtkommen und idealerweise erlauben, den Inhalt der Datenbank auf mehrere Knoten im Netzwerk zu verteilen.

Im Folgenden werden daher einige aktuelle Datenbanksysteme vorgestellt und auf ihre Eignung hin überprüft.

MongoDB

Die MongoDB zählt zu den dokumentenbasierten Datenbanksystemen. Im Gegensatz zu einer relationalen Datenbank, die Tabellen mit fester Struktur und festen Datentypen enthält, verwaltet MongoDB *Collections* von potenziell unterschiedlich strukturierten Dokumenten. Dies bedeutet auch, dass Anfragen an die MongoDB nicht per SQL sondern mit einer eigenen Anfragesprache [70] durchgeführt werden. Somit zählt MongoDB zu den NoSQL-Datenbanksystemen.

MongoDB unterstützt mehrere Konzepte, die die Verfügbarkeit der Daten und die Skalierbarkeit der Datenbank begünstigen. Beim *Sharding* wird eine Collection in mehrere Teile (*Shards*) partitioniert, die dann auf jeweils einem Rechner abgelegt werden. Auf diesem Weg können auch große Datenmengen gespeichert und durchsucht werden.

Dieses Konzept ist in der Datenbank-Community bereits unter dem Namen *horizontale Skalierung* bekannt. Horizontale Skalierung steht der bisher oft anzutreffenden vertikalen

Skalierung entgegen, bei der ein einzelner Rechner im Falle von zu geringer Leistung durch einen einzelnen, leistungsfähigeren Rechner ersetzt wird.

Die *Replication* erlaubt es, dieselben Daten auf mehreren Rechnern abzulegen. Sollte ein Rechner nicht verfügbar sein, können Lese- und Schreibanfragen dann auf den verbliebenen Kopien durchgeführt werden. Dadurch bleibt die Verfügbarkeit der Datenbank auch bei technischen Ausfällen von Teilen des Netzwerks oder einigen Rechnern gewährleistet. Zusätzlich können Leseanfragen auf die verfügbaren Kopien verteilt werden, sodass die Latenzen und der Gesamtlesedurchsatz verbessert werden. Allerdings müssen Schreibanfragen auf alle Kopien dupliziert werden, sodass ein trade-off zwischen dem Lesedurchsatz und dem Schreibdurchsatz stattfindet.

Da Sharding und Replication beliebig kombinierbar sind, muss je nach den Anforderungen des Projekts eine zugeschnittene Feinjustierung vorgenommen werden.

Elasticsearch

Bei Elasticsearch handelt es sich um eine von Shay Bannon im Jahr 2010 entwickelte, verteilte, hochskalierbare Such-Engine, die auf der Suchmaschine Apache Lucene basiert. Die Speicherung der Daten erfolgt bei Elasticsearch ebenso wie bei MongoDB dokumentenbasiert, daher bezeichnet man die kleinste durchsuchbare Einheit als *document*. Jedes *document* ist von einem ganz bestimmten *type* und bildet gemeinsam mit vielen weiteren *documents* - oder im Zweifelsfall auch allein - einen *Index*. Vergleicht man diesen Aufbau mit jenem herkömmlicher Datenbanken, so lässt sich ein *Index* mit einer Datenbank, ein *type* mit einer Tabelle und ein *document* mit einer einzelnen Tabellenzeile gleichsetzen. Jeder *Index* lässt sich in mehrere sogenannte *shards* unterteilen, die, falls Elasticsearch auf mehreren Rechenknoten betrieben wird, auf ebendiese aufgeteilt werden können, um die Geschwindigkeit sowie bei redundanter Verteilung ebenfalls die Ausfallsicherheit zu erhöhen. Jeder *shard* wird intern mittels eines Lucene-Index realisiert.

Elasticsearch kann entweder auf einem oder auf mehreren Rechenknoten, sogenannten *Nodes*, betrieben werden. Verwendet man lediglich einen einzigen Node, so bildet dieser den gesamten Cluster. Werden hingegen mehrere Nodes verwendet, so muss ein Master-Node spezifiziert werden, der die übrigen Nodes koordiniert und darüberhinaus als Erster alle Queries entgegennimmt, um sie daraufhin an einen oder mehrere entsprechende andere Nodes weiterzupropagieren.

Das Formulieren von Suchabfragen an Elasticsearch erfolgt mit Hilfe einer RESTful API, an welche die jeweilige Query als JSON-Dokument gesendet wird. Die daraufhin erhaltene Response befindet sich ebenfalls im JSON-Format. Für diese RESTful API existiert zudem eine Unterstützung durch Spring Data, die es ermöglicht, das Formulieren nativer JSONs zu umgehen und das Stellen von Queries sowie die Verarbeitung der Responses zu vereinfachen. Dies sei an späterer Stelle genauer erläutert.

Es lässt sich also feststellen, dass Elasticsearch geradezu ideal für die Zwecke dieser Projektgruppe ist, da es verteilt einsetzbar und zudem hochskalierbar ist, was im Bereich des Big Data unabdingbar ist, und da darüberhinaus eine komfortable Java-Anbindung gegeben ist, sodass Elasticsearch ohne großen Aufwand in das Projekt integriert werden kann.

Cassandra

Ein weiteres NoSQL-Datenbanksystem, das sich für die Zwecke dieser Projektgruppe einsetzen ließe, ist Apache Cassandra. Dabei handelt es sich um eine hochskalierbare, sehr ausfallsichere, verteilte Datenbank, die zur Persistierung von Daten eine Kombination aus Key-Value-Store und spaltenorientiertem Ansatz nutzt. Ersteres bedeutet in grundlegender Form, dass zur Speicherung von Daten nicht wie bei herkömmlichen Datenbanken Tabellen verwendet werden, sondern jedem zu speichernden Wert (*value*) ein eindeutiger Schlüssel (*key*) zugeordnet wird, mittels dessen auf den entsprechenden Datensatz zugegriffen werden kann. Jeder derartige Datensatz wird in einer sogenannten Spalte (*column*) abgelegt und mit einem Zeitstempel versehen. Mehrere *columns* lassen sich - analog zu einer Tabelle bezogen auf relationale Datenbanken - zu einer *column family* zusammenfassen. Eine *column* kann darüber hinaus als *super column* markiert werden, sodass sie nicht nur mit Hilfe von Schlüsselwerten, sondern auch anhand der Zeitstempel sortiert werden kann.

Auf technischer Ebene besteht ein Cassandra-Cluster aus einer Menge von Nodes, die mittels des *Gossip Protocol* kommunizieren. Dies funktioniert analog zu der dem Protokollnamen entsprechenden Kommunikation im realen Leben folgendermaßen: Jeder Rechenknoten tauscht mit einem oder mehreren ihm bekannten Knoten sein Wissen aus, welche wiederum auf ebendiese Weise verfahren, bis schließlich alle Nodes denselben Wissensstand besitzen.

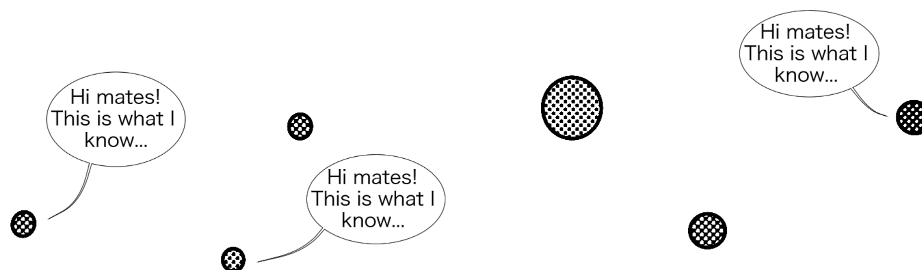


Abbildung 3.17: Veranschaulichung des Gossip Protocol [2]

Die Menge der persistierten Datensätze eines sogenannten *Keyspace*, also einer Menge von Schlüsselwerten, ist als Ring zu betrachten, für die Verwaltung dessen Teilmengen

jeweils ein Node zuständig ist. Die Zuweisung der Zuständigkeiten erfolgt dabei durch einen Partitioner. Jeder Cassandra-Cluster besitzt einen oder mehrere Keyspaces, für die jeweils ein sogenannter *Replication Factor* festgelegt wird. Dieser bestimmt die Anzahl verschiedener Rechenknoten, auf denen die Speicherung eines Datensatzes erfolgen muss, und dient zur Erhöhung der Redundanz und somit der Ausfallsicherheit der Datenbank.

Zur Replikation von Datensätzen existieren zwei verschiedene Ansätze, deren einfachere Variante in der *Simple Replication Strategy* besteht. Gemäß dieses Verfahrens wird ein Datensatz in jeweils einem Knoten gespeichert und daraufhin im Uhrzeigersinn durch eine dem *Replication Factor* entsprechende Anzahl von Knoten repliziert. Bei der *Network Topology Strategy* handelt es sich um eine Replikationsstrategie für größere Cluster. In diesem Fall gilt der *Replication Factor* pro Datacenter, sodass jeder Datensatz durch eine dem *Replication Factor* entsprechende Zahl von Nodes eines anderen Racks, also Teilbereiches, des Datacenters repliziert werden muss.

Während zur Durchführung einer Read/Write-Operation in der *Simple Replication Strategy* ein beliebiger Knoten angesprochen und die Daten unmittelbar weiterpropagiert werden können, fungiert der in der Variante der *Network Topology Strategy* angesprochene Knoten als *Coordinator*, der mit den sogenannten *Local Coordinators* der jeweiligen Datacenters kommuniziert, welche wiederum dort für ein lokales Weiterpropagieren der Daten sorgen.

Es ist möglich, das Konsistenzlevel einer Read/Write-Operation festzulegen, indem eine Anzahl von Knoten bestimmt wird, die dem Coordinator geantwortet haben müssen, bevor dieser eine Antwort an den die Operation ausführenden Client weitergeben kann. An dieser Stelle befindet sich ein Schwachpunkt von Cassandra, da mit wachsender Konsistenz die Geschwindigkeit, mit der eine Operation durchgeführt werden kann, sinkt, eine steigende Geschwindigkeit jedoch Einbußen in der Konsistenz zur Folge hat.

PostgreSQL

Eine weitere Möglichkeit ist der Einsatz einer klassischen relationalen Datenbank. Eine solche bietet verschiedene Vorteile:

Mächtige Anfragesprache Das relationale Modell und die damit verbundene Anfragesprache SQL erlaubt die Formulierung von einer Vielzahl von deklarativen Anfragen. Auch komplexe Datenanalysen können von einem relationalen Datenbanksystem durchgeführt werden, was beispielsweise mit Cassandra aufgrund der restriktiveren Anfragesprache im Allgemeinen nicht möglich ist.

Jahrzehntelange Optimierung Relationale Datenbanken sind seit Jahrzehnten der Standard im Datenbankbereich, und dementsprechend hoch entwickelt. Somit können sie

architekturbedingte Nachteile unter Umständen durch geschickte Optimierung wettmachen.

Transaktionssicherer Betrieb Im Gegensatz zu anderen Systemen bieten relationale Datenbanken eine Vielzahl von Garantien, was die Ausfall- und Transaktionssicherheit angeht.

Relationale Datenbanken stehen oft unter dem Ruf, dass diese Vorteile dadurch erkauft werden, dass die Verarbeitung von sehr großen Datenmengen nicht effizient möglich ist. In der Tat haben relationale Datenbanken zwei Eigenschaften, die sie für den Big-Data-Kontext als nicht sehr geeignet erscheinen lassen. Zum Einen verfügen sie über ein starres Datenbankschema, das genau definiert, welche Typen die Einträge in der Datenbank haben müssen. Es ist also schwierig, mit nachträglichen Änderungen oder schwach strukturierten Daten umzugehen. Zum Anderen sind die meisten großen relationalen Datenbanksysteme auf den Betrieb auf einem einzelnen Rechner ausgelegt. Dies limitiert die Skalierbarkeit des Systems.

Data Warehousing Es ist allerdings möglich, diese Nachteile ein Stück weit auszugleichen, wenn die Datenbank so konzipiert ist, dass die Ausführung der vorgesehenen Analysen effizient möglich ist. Dafür bestimmte Prinzipien werden seit den 90er Jahren unter den Begriffen *Data Warehousing* und *Dimensional Modelling* zusammengefasst [53].

Die Essenz dieser Verfahren besteht darin, dass der Fokus, anders als bei herkömmlichen, auf Normalisierung basierenden Datenbankdesigns, nicht auf der Vermeidung von Redundanz, sondern auf der Minimierung des Rechenaufwands für Analyseanfragen liegt. Vor allem Join-Operationen zwischen großen Tabellen werden zu vermeiden versucht. Um dies zu erreichen, wird bei dimensionaler Modellierung zwischen zwei Tabellentypen unterschieden: Faktentabellen, deren Einträge zu den Ereignissen korrespondieren, die primär analysiert werden sollen, und deutlich kleineren Dimensionstabellen, die die möglichen Ausprägungen dieser Ereignisse darstellen. Diese werden üblicherweise sternförmig angeordnet, sodass Joins jeweils immer nur zwischen einer Fakten- und einer Dimensionstabelle durchgeführt werden müssen. Ein typisches Schema ist in Abbildung 3.18 dargestellt.

Eine Konsequenz dieser Modellierung ist, dass Daten mitunter redundant gespeichert werden. Beispielsweise könnte in einer Dimensionstabelle derselbe String in verschiedenen Tupeln wiederholt vorkommen. Dies wird in Kauf genommen, um die Analyseperformanz zu verbessern.

PostgreSQL PostgreSQL wird gemeinhin als das am höchsten entwickelte relationale Open-Source Datenbanksystem betrachtet [28]. Es unterstützt den gesamten SQL-Standard sowie das ACID-Paradigma zur Transaktionssicherheit. Es ist somit unter den relationalen Datenbanken die offensichtliche Wahl für den Einsatz in der PG.

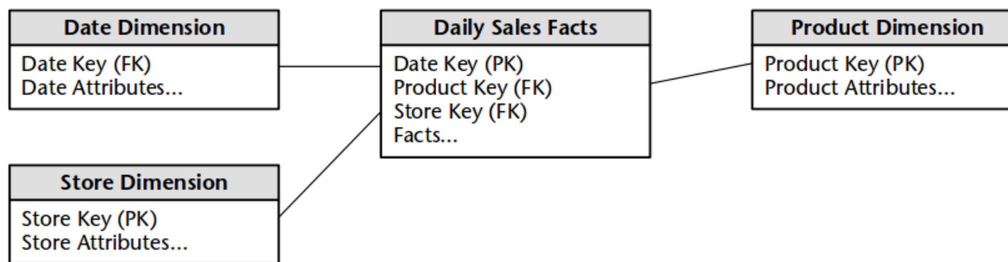


Abbildung 3.18: Ein typisches Datenbankschema nach dimensionaler Modellierung, hier am Beispiel einer Vertriebsdatenbank [53]

PostgreSQL ist zudem attraktiv, weil es JSON als Datentyp unterstützt. JSON-Dokumente können nicht nur in relationalen Tabellen abgelegt werden, sondern auch über spezielle Operatoren modifiziert und ausgewertet werden. Dies kann eingesetzt werden, um auch weniger strukturierte Daten mit PostgreSQL zu verarbeiten.

Ein interessanter Ableger von PostgreSQL ist Postgres-XL. Hierbei handelt es sich um ein Projekt mit dem Ziel, PostgreSQL für den Betrieb als verteilten Datenbankcluster zu erweitern. Es führt dazu Mechanismen für *Sharding* ein, also für das Aufspalten von Tabellen auf mehrere Clusterknoten. Gleichzeitig bewahrt es die Vorteile von PostgreSQL, wie zum Beispiel die ACID-Garantien. Für Fälle, in denen die Datenmengen zu groß für eine einzelne Maschine sind, stellt Postgres-XL eine mögliche Lösung dar.

3.3.2 RESTful APIs

In diesem Abschnitt soll nun gezeigt werden, wie die Indexdaten und zwischengespeicherten Ergebnisse aus den Datenbanken Nutzern zur Verfügung gestellt werden können. Dazu wird die Idee eines Service-Interfaces verdeutlicht und danach werden die Grundlagen einer RESTful APIs vorgestellt.

Grundlegende Idee

Mit der weiteren Verbreitung von unterschiedlichen Endgeräten werden die Anforderungen an Software-Projekte immer komplexer. Reichte es früher aus, nur eine klassische Desktop-Anwendung bereitzustellen, wird heute auch eine Webseite, eine App usw. gewünscht. Somit muss die Geschäftslogik an drei oder mehr unterschiedlichen Stellen implementiert werden. Dies ist offensichtlich alles andere als einfach zu warten und ein Fehler in einer Anwendung kann die Logik einer anderen beeinträchtigen, da alle auf denselben Daten arbeiten. Schon seit etlichen Jahren hat es sich in der Praxis als nützlich erwiesen, wenn die Geschäftslogik und die Anzeige der Daten getrennt voneinander implementiert werden. Wenn man nun diese Trennung nicht nur intern in einer Anwendung beachtet, sondern die

Geschäftslogik zentral auf einem Server bereitstellt und die unterschiedlichen Anwendungen als Clients darauf zugreifen lässt, umgeht man das Problem der verteilten Logik und kann dennoch für jeden Anwendungsfall die passende Darstellung erzielen.

Darüberhinaus hat es sich in der Praxis bewährt, wenn solche Schnittstellen keine klassischen Sitzungen pro Nutzer haben, sondern *Stateless* sind. Hierdurch können komplizierte Mechanismen zur Sitzungsverwaltung und die sonst nötigen großen Zwischenspeicher für die Sessions entfallen. Somit wird die Implementierung der APIs deutlich einfacher und die Nutzer dieser Schnittstellen können einem eindeutig definierten Verhalten pro Aufruf, ohne Blick auf die Sitzungshistorie, vertrauen.

Dabei beschreibt Representational State Transfer (REST) keine festen Regeln oder gar ein starres Protokoll, sondern ist mehr als eine Liste von Vorschlägen zu verstehen, wie man eine solche API designen sollte. Hält man sich möglichst genau an diese Vorschläge, ist es auch für Außenstehende einfacher, sich in eine für sie neue API einzuarbeiten. Auch wenn die Vorschläge die meisten Anwendungsfälle abdecken, so kann es immer Situationen geben, in denen es möglicherweise besser ist, den Standard nicht zu beachten. REST ist somit äußerst flexibel [31, 76].

HTTP

Grundlegend für RESTful APIs ist hierbei die Kommunikation über das Hyper Text Transfer Protocol (HTTP). Dies ist heutzutage möglich, denn fast alle Geräte verfügen über einen Internetanschluss, der sich als Basis für den Austausch zwischen dem Server und den Client eignet. Da das HTTP umfangreich ist und sich als ein Standard-Protokoll für den Austausch von Daten über das Internet etabliert hat, können die nötigen Operationen darüber abgewickelt werden, ohne das ein neues Protokoll designt und implementiert werden muss. HTTP ist dabei ein klassisches Client-Server-Protokoll, bei dem die Kommunikation immer vom Client aus gestartet wird. HTTP regelt dabei die Syntax und Semantik der gesendeten Daten und baut auf TCP/IP auf.

HTTP-Anfragen Eine Anfrage an einen HTTP-Server enthält nicht nur die IP-Adresse des Servers sondern auch einen Server-Pfad, der die gewünschte Ressource näher beschreibt. Diese Kombination wird auch als Uniform Resource Locator (URL) bezeichnet.

Neben der URL wird ein Header-Teil mitgeschickt, der zusätzliche Meta- und Zusatz-Informationen enthält. Dazu können Daten zur Authentifizierung, die gewünschten Formatierung der Antwort oder auch die Größe des Datenfeldes zählen. Eine der wichtigsten Header-Informationen ist hierbei die gewünschte Methode, die der Server unter der URL ausführen soll:

POST Drückt aus, dass die im Body des Request gesendeten Daten erstellt werden sollen.

Code	Text	Beschreibung
200	OK	Drückt aus, dass die Anfrage erfolgreich war.
201	CREATED	Wird oft zurück gegeben wenn ein Datensatz erfolgreich erstellt wurde.
400	BAD REQUEST	Die Anfrage konnte nicht vom Server gelesen werden, da sie falsch formatiert war oder anders als fehlerhaft erkannt wurde.
404	NOT FOUND	Die Anfrage konnte nicht erfolgreich bearbeitet werden, da die Resource nicht gefunden wurde.
500	INTERNAL SERVER ERROR	Der Server hat intern einen (schwerwiegenden) Fehler und kann daher die Anfrage nicht richtig beantworten.

Tabelle 3.1: Übersicht von geläufigen HTTP Status Codes

GET Wird verwendet, wenn Daten vom Server gelesen werden sollen.

PUT Leitet ein Update von schon bestehenden Daten ein.

DELETE Bittet den Server bestimmte Daten zu löschen.

OPTIONS Fragt den Server, welche (anderen) Methoden für eine bestimmte URL zulässig sind.

Durch diese Methoden werden die grundlegenden Create, Read, Update and Delete (CRUD)-Operationen unterstützt.

Abschließend kann die Anfrage auch Daten enthalten, welche aus reinem Text bestehen, jedoch beliebig formatiert sein können. Dies ist besonders bei *POST*- und *PUT*-Aufrufen wichtig, um dem Server die zu erstellenden bzw. zu aktualisierenden Daten mitzuteilen. Bei *GET*- und *DELETE*-Aufrufen bleiben diese Daten zumeist leer.

HTTP-Antworten Die Antwort des Servers enthält auch einen Header-Teil, in dem der Server bestimmte Meta- und Zusatz-Informationen zurückschickt. Üblicherweise zählen dazu das Datum und die aktuelle Uhrzeit, die Größe der Antwort im Datenfeld und welches Format dieses hat. Hierbei spielt der Status Code eine besondere Rolle, da dieser eine Antwort zu Erfolg, Problemen und Misserfolg der Anfrage liefert (vgl. Tabelle 3.1).

Ähnlich zur Anfrage kann natürlich auch die Antwort Daten enthalten, welche bei allen Methoden entstehen können. Auch diese Daten sind reiner Text, können jedoch unterschiedlich formatiert sein [32].

JSON

Auch wenn es keine vorgeschriebene Art bzw. Formatierung gibt, wie Daten über eine

RESTful API ausgetauscht werden sollen, so wird in der Praxis häufig die Extensible Markup Language (XML) oder die JavaScript Object Notation (JSON) verwendet.

Da beide Optionen relativ ähnlich in ihrer Ausdrucksstärke sind, liegt die Wahl, ob man eine der beiden oder gar eine dritte Möglichkeit verwendet, beim Designer der Schnittstelle. In früheren APIs wurde stark auf XML gesetzt, sodass viele Anwendungen dieses auch heute noch bevorzugen. In letzter Zeit ist jedoch ein Trend hin zu JSON zu beobachten. Dies liegt darin begründet, dass viele Clients Single-Site-Webapplications sind, die in JavaScript implementiert wurden und JSON als Teil der JavaScript-Welt so direkt interpretiert werden kann. Somit bleibt ein aufwändiger und langsamer Parser erspart. JSON ist darüberhinaus auch noch recht einfach von Menschen zu lesen, sodass auch eine Interaktion mit der API ohne speziellen Client möglich ist.

Im Kern besteht ein JSON-Dokument aus Key-Value-Paaren, die in *Objekten* zusammengefasst sind. Der Schüssel dieses Paares ist dabei immer ein Text, während der Wert unterschiedlichste Typen annehmen kann. Dazu zählen Text, Nummern (ganzzahlig oder mit Fließkomma), boolsche Werte (*true* und *false*), ein Array oder wiederum ein Objekt [24]. Ein Beispiel für ein solches JSON-Dokument wird in Listing 3.1 gezeigt.

```
1  {
2          "hello": "world",
3          "true": false,
4          "array": [
5                  1, 2, 3
6          ],
7          "kord": {
8                  "x": 1.23,
9                  "y": 4.56
10         }
11 }
```

Listing 3.1: Ein Beispiel für ein JSON Dokument

Kapitel 4

Maschinelles Lernen

Das letzte Kapitel im Teil *Big Data Analytics* bildet das maschinelle Lernen. Wie in Kapitel 2 erläutert, besteht der Zweck des Umgangs mit den riesigen Datenmengen in der Analyse. Das bedeutet, dass automatisch erlernt werden soll, wie sich die gegebenen Informationen verallgemeinern lassen. Dieser Schritt ist wichtig, damit das Erlernte auf neue, bisher noch nicht betrachtete Daten angewendet werden kann und nicht nur für die bereits angeschauten Daten gilt. Die gefundenen Regelmäßigkeiten sollen dementsprechend ermöglichen, dass automatisiert Erkenntnisse über neue Daten erlangt werden können. Zuerst soll es in diesem Kapitel um die Grundbegriffe des maschinellen Lernens und die formalen Konzepte zur Datenanalyse gehen. Die dafür benötigten Grundlagen wurden aus [72], [96] und [35] zusammengetragen. Anschließend folgen einige vertiefende Abschnitte, welche Verfahren diskutieren, die speziell auf *Big Data* zugeschnitten sind. Schließlich bildet die Analyse von riesigen Datenmengen neue Herausforderungen an maschinelle Lernverfahren, wie in Kapitel 2 gezeigt wurde.

(Un-)Überwachtes Lernen Man unterscheidet zuerst zwischen überwachtem und unüberwachten Lernen. Beim überwachten Lernen liegen zusätzlich zu den gesammelten Daten auch Informationen darüber vor, in welche Klassen oder Kategorien man die Daten einteilen kann. Genau diese Zuteilung soll zukünftig für neu beobachtete Daten vorhergesagt werden. Meistens entsteht die Annotation der vorliegenden Daten mit einer passenden Klasse durch einen Experten. Beim unüberwachten Lernen hingegen liegen diese Klasseninformationen zu den gesammelten Daten nicht vor. Mit speziellen Lernverfahren wird versucht, die vorliegenden Daten in passende Klassen einzuteilen. Die Einteilung basiert nur auf den in den Daten gefundenen Regelmäßigkeiten und geschieht automatisch. In den nun folgenden einführenden Worten soll es genau um das überwachte Lernen gehen. section 4.2 beschäftigt sich schließlich mit den Formalien beim unüberwachten Lernen.

Die Lernaufgabe Etwas formaler besteht die Lernaufgabe aus dem Trainieren eines Modells, welches das gelernte Wissen repräsentieren soll, und aus der Anwendung des

Modells auf neue Daten. Für das Training werden annotierte Trainingsdaten

$$\mathcal{T} = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), ..., (\vec{x}_N, y_N)\} \subset X \times Y$$

benötigt, wobei X für das gesamte Universum möglicher Daten steht und Y für die Menge an verfügbaren Klassen. Bei einer Klassifikation sind dies endlich viele vorgegebene Klassen, bei einer Regression sind dies die reellen Zahlen. Jedes Datum besteht aus einem Vektor $\vec{x}_i$, welcher die Merkmale des individuellen Datums repräsentiert, und aus einer Annotation y_i. Unter einem Merkmal (engl. *Feature*) versteht man eine für die Vorhersage nützliche Größe. Merkmale können direkt physikalisch messbar oder aus messbaren Größen berechenbar sein. Beispielsweise können für die Klassifizierung von Texten die Vorkommen bestimmter Wörter (direkt zählbar) oder das Vorkommen von Wortstämmen (daraus ableitbar) Merkmale darstellen. Die Annotation steht für die Klasse, zu der das betrachtete Datum gehört. Sie ist essentiell für das überwachte Lernen und den Erfolg der maschinellen Lernverfahren.

In unserer Projektgruppe fällt mit der Gamma-Hadron-Separation eine typische Klassifikationsaufgabe an. Dabei bilden die durch die Monte-Carlo-Simulation erlangten Daten den Trainingsdatensatz. Die Klassen sind in unserem Fall $Y = \{\texttt{gamma}, \texttt{hadron}\}$ und sind Annotationen solcher Aufnahmen, welche mit Hilfe der Simulation entweder als Gamma- oder als Hadronstrahlung eingeordnet wurden. Mit diesem Trainingsdatensatz werden maschinelle Lernverfahren trainiert und mit den resultierenden Modellen wollen wir versuchen, für Rohdaten vorherzusagen, ob in einer Aufnahme eine für die Physiker interessante Gammastrahlung vorliegt oder nicht. Außerdem liegt mit der anschließenden Energieschätzung für die Partikel einer gefundenen Gammastrahlung eine Regressionsaufgabe vor, welche ebenfalls mit maschinellen Lernverfahren gelöst werden kann.

Qualitätsmaße Es gibt etliche Lernverfahren, mit denen sich Modelle trainieren lassen. Um das beste Modell für die Lernaufgabe zu finden, sollte die Generalisierungsleistung des Modells im Auge behalten werden. Darunter versteht man die Anwendbarkeit auf neue Daten, für welche die Klasse unbekannt ist. Die sogenannte Fehlklassifikationsrate kann dazu beitragen, die Generalisierungsleistung eines Modells zu quantifizieren. Häufig werden Modelle nicht auf dem gesamten verfügbaren Trainingsdatensatz trainiert, sondern es wird eine Teilmenge der Trainingsdaten zurückgehalten. Diese bilden die Testdaten, welche von dem trainierten Modell klassifiziert werden. Im Nachhinein können vorhergesagte und wahre Klasse verglichen werden, um die Fehler dieses Modells auf unbekannten Daten einschätzen zu können. Um die Fehlklassifikationsrate zuverlässig zu bestimmen, müssten unendlich viele Testdaten klassifiziert werden, sodass man in der Praxis auf empirische Schätzungen wie folgende zurückgreift:

$$\epsilon(h) = \mathbb{E}_{x \sim \mathcal{D}}[\mathbb{I}(h(x) \neq f(x))] \quad [96]$$

wobei h ein trainiertes Modell, $\mathbb{E}_{x\sim\mathcal{D}}[g(x)]$ der Erwartungswert der Funktion $g(x)$, wenn x nach $\mathcal{D}$ verteilt ist und $\mathbb{I}(g(x))$ die Indikatorfunktion (1, wenn $g(x) = true$ und 0 sonst). Gewählt wird der Lerner h, welcher den Fehler $\epsilon(h)$ minimiert. Weitere Qualitätsmaße werden in subsection 4.6.2 beleuchtet.

Dieser kurzen Einführung in das maschinelle Lernen folgen nun Vertiefungen. Es werden Lernverfahren und Techniken beleuchtet, welche sich in der Praxis bewiesen haben und daher für unsere Projektgruppe interessant sein können. Dabei wird vor allem Wert darauf gelegt, dass diese Techniken für Big Data anwendbar sind. Große Datenmengen sollen nicht nur schnell bearbeitet werden, es sollen auch die Vorteile eines Rechenclusters ausgenutzt werden können. Es soll besonders darauf eingegangen werden, wie sich Lernverfahren parallelisieren lassen, sodass verteilt gelernt und auch klassifiziert werden kann. Einen weiteren Aspekt bilden die inkrementellen Verfahren, bei welchen die Trainingsdaten nicht zwingend komplett zu Beginn des Trainings vorliegen müssen. Da wir uns mit riesigen Datenmengen beschäftigen, könnte es ein Vorteil sein, diese Daten nach und nach vom Lerner unserer Wahl bearbeiten zu lassen. Ein weiteres Problem unserer Trainingsdaten ist außerdem, dass üblicherweise sehr viele Hadronstrahlungen, aber nur wenige Gammastrahlungen vorliegen. Deswegen soll das Lernen mit nicht balancierten Klassen ebenfalls vertieft werden. Den Abschluss dieses Kapitels bilden Techniken, mit denen die Daten vor dem Lernen organisiert werden können. Dazu gehört zum Einen die Extraktion von Merkmalen, welche besonders gut für die Vorhersage der Klassen geeignet ist, zum Anderen die passende Einteilung in Trainings- und Testdatensätze. Schließlich sollen die trainierten Modelle zum Schluss evaluiert werden, sodass eine Aussage über deren Qualität möglich ist.

4.1 Ensemble Learning

Die Idee des Ensemble Learnings ist, auf viele Modelle zurückzugreifen, anstatt sich nur auf die Vorhersagen eines Modells zu verlassen. Nach Dietterich [27] sind die drei meistgenannten Gründe für das Nutzen von Ensembles die folgenden:

Statistik Ähnlich unserem realen Leben soll mehreren Expertenmeinungen anstatt nur einer vertraut werden. Es kann schwierig sein, sich für genau ein Modell zu entscheiden, welches möglicherweise nur zufällig auf dem gerade genutzten Testdatensatz die kleinste Fehlerrate hat. Außerdem können durchaus mehrere Modelle mit einer ähnlich akzeptablen Fehlerrate für den Anwender interessant sein. Im Ensemble soll nicht strikt ein Modell ausgesucht werden, sondern eine Kombination entstehen.

Berechnung Zum Training einiger Modelle wird eine Optimierung durchgeführt, welche in lokale Optima enden kann. Trainiert man Modelle von verschiedenen Startpunkten aus und kombiniert diese, kann es zu einer Verbesserung kommen.

Repräsentierbarkeit Manchmal kann die gesuchte wahre Funktion nicht von den Modellen im Hypothesenraum repräsentiert werden. Auch hier kann eine Kombination von Modellen dazu beitragen, den Raum darstellbarer Funktionen zu vergrößern.

In dieser Einführung wird davon ausgegangen, dass den Modellen dasselbe Lernverfahren zugrunde liegt. Meist ist dieses Verfahren von recht einfacher Struktur, sodass mehrere schwache Lerner zu einem starken Lerner durch eine gemeinsame Entscheidungsregel zur Klassifikation neuer Daten kombiniert werden. Die einfachen Lerner sollen dabei möglichst verschieden sein, damit eine Kombination erst sinnvoll wird. Um verschiedenartige Lerner eines gleichen Basisalgorithmus zu erzielen, gibt es verschiedene Ansätze. Im Folgenden stehen *Bagging* (insbesondere Random Forests nach [59]) und *Boosting* (insbesondere Ada-Boost nach [34]) im Fokus. Neben diesen beiden Quellen wurden auch Grundlagen aus [96], [27] und [75] über das Ensemble Learning entnommen und können für weitere Informationen nachgeschlagen werden. Die Grundideen der beiden Ensemble Learning Methoden sollen erläutert werden, sowie deren möglicher Einsatz in unserer Projektgruppe.

4.1.1 Bagging

Beim Bagging (Bootstrap Aggregation) werden für jeden Lerner Bootstrap-Stichproben genutzt. Das bedeutet, dass für jeden Lerner neue Trainingsdaten generiert werden, indem n Beispiele aus den originalen n Beobachtungen mit Zurücklegen gezogen werden. Manche Beispiele können somit mehrfach in einem Trainingsdatensatz vorkommen, andere gar nicht.

Ein prominenter Vertreter der Bagging-Methoden ist der **Random Forest** oder auch **Zufallswald**. Der Basislerner zu einem solchen Zufallswald ist ein Entscheidungsbaum, wie er beispielhaft in Abbildung 4.1 zu sehen ist.

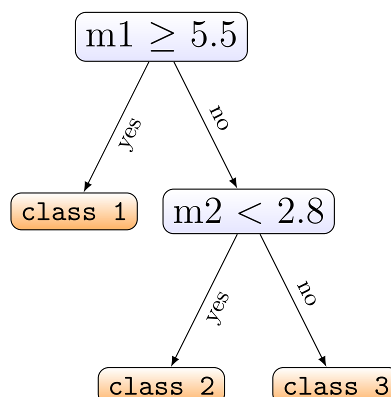


Abbildung 4.1: Beispielhafter Entscheidungsbaum

Data: Trainingsdatensatz $\mathcal{T} = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), ..., (\vec{x}_N, y_N)\}$,
Anzahl T der Bäume im Wald,
Anzahl M der Merkmale, die für Splits verwendet werden sollen
Result: T trainierte Entscheidungsbäume, welche den Zufallswald bilden
Ziehe T Bootstrap-Stichproben mit Zurücklegen;
for $t = 1, ..., T$ **do**
  Trainiere einen Baum mit der Bootstrap-Stichprobe t mit folgender Modifikation: Ziehe zufällig M Merkmale aus den Originalmerkmalen der Beobachtungen. Für die Splits werden nur diese gezogenen Merkmale betrachtet.
  Entstehender Baum wird nicht gestutzt.
end

Algorithm 1: Konstruktion von Zufallswäldern [59]

Die Blätter in einem solchen Baum entsprechen den Klassen, die inneren Knoten entsprechen Splits anhand von Merkmalen. Die Splits werden jeweils so gewählt, dass möglichst viele Beobachtungen getrennt werden können. Es werden so lange neue Splits gewählt, bis die aktuell betrachtete Beobachtungsmenge nur noch aus einer Klasse stammt.

Jeder Baum im Wald wird mit einer Bootstrap-Stichprobe trainiert. Außerdem werden für Splits nicht alle Attribute des Trainingsdatensatzes genutzt, sondern nur eine zufällige Teilmenge. So entstehen möglichst viele verschiedene Entscheidungsbäume, welche zusammen den Zufallswald bilden. Die Vorgehensweise für die Konstruktion eines Zufallswaldes ist in Algorithmus 1 zu sehen.

Neue Daten werden von jedem Baum klassifiziert, anschließend erfolgt ein Mehrheitsentscheid. Je mehr Bäume im Wald sind, desto besser für die Klassifikation. Im Gegensatz zu einem einzelnen Baum besteht das Problem des Overfittings nicht, da für jeden Baum eine zufällige Teilmenge der Merkmale ausgewählt wird. Führt man dies nicht durch und nimmt beispielsweise an, dass es zwei Merkmale mit einem sehr starken Beitrag zur Klassentrennung gibt, dann würden alle Bäume im Wald genau diese Merkmale für ihre Splits wählen. Daraus folgt eine starke Korrelation zwischen den Bäumen, was genau zum Overfitting führt. Wählt man nun aber wie oben beschrieben für jeden Baum eine zufällige Teilmenge an Merkmalen aus, dann taucht keine starke Korrelation auf und der Mehrheitsentscheid ist stabil. Außerdem sind Zufallswälder praktisch bei vielen Merkmalen, welche nur einen kleinen Beitrag zur Klassentrennung liefern und durch diese zufällige Merkmalsauswahl genau die gleiche Chance, haben für einen Split gewählt zu werden wie andere Merkmale, welche einen möglicherweise größeren Beitrag liefern.

Für unsere Projektgruppe könnte außerdem von Vorteil sein, dass sowohl Konstruktion als auch Klassifikation mit Zufallswäldern gut parallelisierbar ist. Die Konstruktion erfolgt unabhängig von den anderen trainierten Bäumen und die Ergebnisse vieler Bäume auf verschiedenen Rechnern können am Schluss gemeinsam ausgewertet werden.

Ein Nachteil der Zufallswälder ist allerdings, dass die Verständlichkeit verloren geht, die

Data: Trainingsdatensatz $\mathcal{T} = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), ..., (\vec{x}_N, y_N)\}$ mit $y_i \in \{-1, +1\}$,
Anzahl T der Lerner und deren Basisalgorithmus
Result: $H(\vec{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\vec{x})\right)$
$\mathcal{D}_1(i) = 1/N$ als initiale Gewichte;
for $t = 1, ..., T$ **do**
Trainiere Lerner h_t mit Datensatz $\mathcal{T}$ und den aktuellen Gewichten in $\mathcal{D}_t$;
Berechne den Fehler $\epsilon_t = \Pr_{i \sim \mathcal{D}_t}[h_t(\vec{x}_i) \neq y_i]$;
Setze das Gewicht des Basislerners t auf $\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t}\right)$;
Updaten der Gewichte: $\mathcal{D}_{t+1}(i) = \frac{\mathcal{D}_t(i) \cdot \exp(-\alpha_t y_i h_t(\vec{x}_i))}{Z_t}$
dabei wird Z_t zur Normalisierung genutzt ;
end

Algorithm 2: AdaBoost [34]

ein entscheidender Vorteil bei der Wahl von einzelnen Entscheidungsbäumen sein kann. Durch die grafische Darstellung erschließt sich die Klassifikation auch Laien gut, was bei einem Zufallswald von 100 oder mehr Bäumen nicht mehr der Fall ist.

4.1.2 Boosting

Beim Boosting werden Gewichte für jedes Trainingsbeispiel eingeführt. Initial werden Gleichgewichte gewählt, im Laufe des Trainings sollen die „schwierigen“ Beispiele, welche immer wieder falsch klassifiziert werden, höher gewichtet werden. Entscheidet man sich im Vorfeld für ein Ensemble aus T einfachen Lernern, so gibt es T Trainingsrunden, in denen jeweils ein Lerner mit den gewichteten Beispielen trainiert wird. Nach jeder dieser Runden erfolgt eine Evaluation und Anpassung der Gewichte. Das entstehende Ensemble wird zugunsten der schwierigen Beobachtungen im Lerndatensatz adaptiert.

Populär ist der Ansatz **AdaBoost** von Freund und Schapire. Im Folgenden soll die ursprüngliche Version von 1997 für ein Zwei-Klassen-Problem vorgestellt werden, für welche die Vorgehensweise in Algorithmus 2 abgebildet ist.

Einfache Lerner werden nach ihrer Qualität gewichtet. Ist der Fehler $\epsilon_t < 0.5$, so ist das Gewicht $\alpha_t > 0$. Je kleiner der Fehler, desto größer das Gewicht des Lerners. Beobachtungen werden nach ihrer Schwierigkeit gewichtet. Der neue Wert hängt nach jeder Trainingsrunde von dem Term $\exp(-\alpha_t y_i h_t(\vec{x}_i))$ ab. Wenn richtig klassifiziert wurde, ist $y_i h_t(\vec{x}_i) = 1$, dann wird der Term $\exp(-\alpha_t)$ klein und so auch das neue Gewicht. Wenn allerdings falsch klassifiziert wurde, ist $y_i h_t(\vec{x}_i) = -1$, dann wird der Term $\exp(\alpha_t)$ groß und das neue Gewicht ebenso. Nach dem Ablauf aller Trainingsrunden erfolgt die Klassifikation neuer Daten durch einen gewichteten Mehrheitsentscheid.

Parallelisieren lassen sich Boosting-Ansätze nur schwer, da in jeder Trainingsrunde eine Abhängigkeit zur vorhergehenden Runde besteht. Außerdem wächst das Risiko des Over-

fitting mit der Anzahl T der Lerner. Die Lerner sollten in der Lage sein, Verteilungen der Trainingsdaten zu beachten, ansonsten muss der Trainingsdatensatz in jeder Iteration der Verteilung angepasst werden.

4.1.3 Fazit

Es gab mehrere Versuche, Bagging und Boosting miteinander zu vergleichen. Dietterich [27] fand heraus, dass AdaBoost viel besser als Bagging-Ensembles abschnitt, sofern die Trainingsdaten wenig bis kein Rauschen aufwiesen. Sobald jedoch 20% künstliches Rauschen hinzugefügt wurde, schnitt AdaBoost plötzlich sehr viel schlechter ab. Quinlan [75] experimentierte mit unterschiedlichen Lernerzahlen T. Ist T klein, scheint AdaBoost die bessere Wahl zu sein. Je größer jedoch T wird, desto schlechter wird das Ergebnis der Boosting-Methode und desto brauchbarer werden Zufallswälder.

Die Ergebnisse lassen sich damit erklären, dass Zufallswälder robust gegen Overfitting sind, wohingegen AdaBoost eher anfällig dafür ist. Beim Boosting wird zu viel Fokus auf die schwierigen Beobachtungen gelegt, denn deren Gewicht wird nach jeder Iteration erhöht. Nach und nach verschwinden die einfachen Beispiele, wodurch Lerner in hohen Trainingsrunden mit einem stark angepassten Trainingsdatensatz arbeiten. Die Konsequenz ist das Overfitting für große T.

Insgesamt lässt sich sagen, dass Ensembles das Gesamtergebnis erheblich verbessern können. Die populärsten Verfahren Bagging und Boosting wurden mit ihren Vor- und Nachteilen vorgestellt. Für unsere Projektgruppe rücken die Zufallswälder in den Fokus. Sie sind nicht nur gut parallelisierbar und robust gegenüber Overfitting, sondern werden aktuell von den Physikern für ihre Klassifikationen verwendet. Daher ist es essentiell für unsere Anwendung, sich ebenfalls mit Zufallswäldern auseinanderzusetzen und diese Möglichkeit der Klassifikation im Endprodukt anzubieten.

4.2 Clustering und Subgruppenentdeckung

In diesem Kapitel wird hauptsächlich das unüberwachte Lernen erläutert. Dabei werden die zwei Lernverfahrensmethoden Clustering und die Subgruppen-Entdeckung erläutert. Während beim überwachten Lernen Hypothesen gesucht werden, die möglichst gute Vorhersagen über bestimmte schon vorgegebene Attribute geben, wird bei unüberwachten Lernmethoden nach unbekannten Mustern gesucht.

4.2.1 Clustering

Clustering [49] ist eine unüberwachte Lernmethode. Sie ist die am meisten verwendete Methode für das Entdecken von Wissen aus einer großen Datenmenge. Bei ihr geht es im Allgemeinen darum, dass Objekte, die ähnliche Eigenschaften besitzen, in einer Gruppe zusammengefasst werden. Dabei werden neue Klassen identifiziert. Die einzelnen Gruppen werden Cluster genannt.

Es gibt verschiedene Arten von Clustering-Verfahren, die sich in ihren algorithmischen Vorgehensweisen unterscheiden. Dazu zählen:

- Partitionierende Verfahren, z.B. der k-means Algorithmus.
- Hierarchische Verfahren, die entweder bottom-up oder top-down vorgehen.
- Dichtebasierte Verfahren, z.B. der DBSCAN Algorithmus.
- Kombinierte Verfahren, bei welchen Methoden aus den oben vorgestellten Verfahren kombiniert werden.

Partitionierende Verfahren

Bei den partitionierenden Verfahren muss die Anzahl der gesuchten Klassen bzw. Cluster am Anfang festgelegt werden. Die Verfahren, die dieser Methodik folgen, starten meistens mit einem zufälligen Partitionieren der Objekte. Im Laufe der Ausführung wird diese Partitionierung schrittweise optimiert. Der k-means Algorithmus [90] gehört beispielsweise zu diesen Verfahren und soll im Folgenden erläutert werden.

Sei $\vec{x} = \{d_1, d_2, ..., d_n\}$ ein Vektor, der ein Objekt im Merkmalsraum repräsentiert. Die Distanz zwischen zwei Vektoren $\vec{x}$ und $\vec{y}$ ist durch $|\vec{x} - \vec{y}| = \sqrt{\sum_{i=0}^{n} (x_i - y_i)^2}$ definiert. Der Mittelpunkt $\vec{\mu}$ einer Menge c_i von Vektoren ist durch $\vec{\mu} = \frac{1}{|c_i|} \sum_{\vec{x} \in c_i} \vec{x}$ definiert. Sei k die Anzahl der gesuchten Cluster. Am Anfang des Algorithmus wird k entweder zufällig oder nach der Durchführung eines Optimierungsverfahren festgelegt. Außerdem werden k Punkte als Cluster-Zentren ausgewählt und die restlichen Objekte dem Cluster mit dem nächsten Zentrum zugewiesen. Bei jedem Durchlauf des Algorithmus werden die Mittelpunkte $\vec{\mu}$ neu berechnet und die Objekte wieder dem Cluster mit dem nächsten Zentrum zugewiesen. Es wird immer weiter iteriert, bis alle Cluster stabil sind.

Der k-means Algorithmus ist für numerische Daten gedacht. Er ist effizient und leicht anzuwenden. Dagegen hat der Algorithmus gewisse Nachteile, da die Cluster stark von k und den am Anfang ausgewählten Cluster-Zentren abhängen. Darüberhinaus zeigt der Algorithmus eine Schwäche, wenn die Daten kugelförmig verteilt sind oder große Abweichungen in Dichte und Größe aufweisen.

Hierarchische Verfahren

Die beliebte Alternative zu den partitionierenden Verfahren sind die hierarchischen Verfahren [23]. Bei ihnen werden die identifizierten Cluster hierarchisch angeordnet. Es wird ein Baum erzeugt, in dem jeder Elternknoten Zweige mit seinen Teil-Clustern besitzt. Die Wurzel repräsentiert den Cluster mit allen Objekten (oberste Ebene). Bei der Identifizierung von Clustern unterscheidet man zwei Vorgehensweisen, nämlich bottom-up oder top-down.

Top-down Clustering auch devisives Clustering genannt. Am Anfang gehören alle Objekte zu einem Cluster. Dieser wird schrittweise aufgeteilt, bis jeder Cluster nur noch ein Objekt enthält.

Bottom-up Clustering auch agglomerativ genannt. Bei diesem Verfahren enthält jeder Cluster am Anfang nur ein Objekt. Danach werden die Cluster im Laufe des Verfahrens vereinigt.

Dichtebasierte Verfahren

Cluster bestehen grundsätzlich aus Objekten, die dicht aneinander sind. Die dichtebasierten Verfahren nutzen diese Eigenschaft aus, um Cluster aufzufinden. Der DBSCAN-Algorithmus [13] ist ein Vertreter und soll nun genauer betrachtet werden.

Um den DBSCAN-Algorithmus zu veranschaulichen, werden zuerst einige Definitionen eingeführt. Eine ϵ-Umgebung definiert die Anzahl der Punkte in einem bestimmten Radius ϵ. **MinPts** ist die Mindestanzahl der Punkte in einer ϵ- Umgebung. Ein **Kernpunkt** ist ein Punkt, der mindestens MinPts in seiner Umgebung hat. Ein **Randpunkt** ist ein Punkt in der ϵ-Umgebung, der kein Kernpunkt ist. Ein **Rauschpunkt** ist ein Punkt außerhalb der ϵ-Umgebung. Zwei Punkte p und q sind **Dichte-erreichbar**, wenn p ein Kernpunkt und q in der ϵ-Umgebung von p ist. Es gibt direkte und indirekte Dichte-Erreichbarkeit. Wenn p von p_1 direkt Dichte-erreichbar ist und p_1 ist direkt Dichte-erreichbar von q, dann ist p indirekt Dichte-erreichbar von q. Aber die andere Richtung gilt nicht.

Die Parameter ϵ und MinPts werden vor der Ausführung des Algorithmus festgelegt. Sie können entweder zufällig gewählt oder durch die Anwendung heuristischer Verfahren bestimmt werden. Der DBSCAN-Algorithmus iteriert über alle Objekte in der Datenmenge und wenn ein Objekt noch nicht klassifiziert und das Objekt ein Kernobjekt ist, dann werden alle von diesem Punkt aus Dichte-erreichbaren Objekte (Punkte) in einem Cluster zusammengefasst. Wenn dies nicht der Fall ist, dann wird das Objekt als Rauschpunkt markiert. Es wird solange iteriert, bis alle Punkte betrachtet wurden.

Kombinierte Verfahren

Man kann die vorgestellten Clustering-Verfahren kombinieren. Das kann nützlich sein, um Parameter eines anderen Verfahrens zu bestimmen. Zum Beispiel führt man eine hierarchische Clusteranalyse durch, um die Anzahl k der Cluster zu bestimmen, die man später als Eingabeparameter an k-means übergibt. Das hat den Vorteil, dass eine optimale Anzahl von Clustern ermittelt wird. Leider ist dieses Verfahren sehr speicher- und zeitaufwendig, da zwei Verfahren immer gleichzeitig angewendet werden müssen.

4.2.2 Subgruppenentdeckung

Die bekannteste Methode zur Erkennung von Mustern mit vorgegebenen Eigenschaften ist die *Subgruppenentdeckung*. Zum ersten Mal wurde sie von Kloesgen und Wrobel [54, 55] eingeführt. Die Subgruppenentdeckung [61] liegt zwischen den zwei Bereichen des maschinellen Lernens, da bei der Subgruppenentdeckung die Vorhersage genutzt werden soll, um eine Beschreibung der Daten zu liefern. Andere Data-Mining-Methoden zur Erkennung von Mustern sind in [18] zu finden.

Definition der Subgruppenentdeckung

Sei $\mathcal{D}$ ein Datensatz, der aus Datenitems $\vec{d_i}$ besteht. Ein Datenitem $\vec{d_i} = (\vec{a}, t)$ ist ein Paar aus Attributen $\{a_1, a_2, ..., a_m\}$, die mit $\vec{a}$ bezeichnet werden, und einem Zielattribut t. In dieser Arbeit werden die Begriffe Datenitem und Transaktion die gleiche Bedeutung haben. Das *Zielattribut* definiert die eingegebene Eigenschaft, für die die Daten erklärt werden sollen. Das Zielattribut muss binär sein, jedoch hat jedes Attribut a_m einen Wert aus einer Domäne $dom(\mathcal{A})$. Die Werte der Attribute können binär, nominal oder numerisch sein. Beispiele für Domänen sind $dom(\mathcal{A}_m) = \{0, 1\}$, $|dom(\mathcal{A}_m)| \in \mathbb{N}_0$ oder $dom(\mathcal{A}_m) = \mathbb{R}$. $\vec{d_i}$ wird das i-te Datenitem genannt. Außerdem bezeichnen $\vec{a^i}$ und t^i den i-ten Vektor der Attribute und das i-te Zielattribut. Die Größe der Datenmenge wird mit $N = |\mathcal{D}|$ bezeichnet.

Nun benötigt man die Definition einer Regel, um eine Subgruppe definieren zu können. Eine Regel ist eine Funktion $p : P(\mathcal{A}) \times dom(\mathcal{A}) \rightarrow \{0, 1\}$, wobei $P(\mathcal{A})$ die Potenzmenge der Attribute darstellt. Mit $\mathcal{P}$ bezeichnet man die Menge aller Regeln. Man sagt, eine Regel p überdeckt einen Datenitem $\vec{d^i}$ genau dann, wenn $p(\vec{a^i}) = 1$ ist. Die Attribute werden miteinander konkateniert, um $\vec{a}$ zu konstruieren. Eine Regel hat die Form:

$$\text{Bedingung} \rightarrow \text{Wert der Regel.}$$

Die Bedingung einer Regel ist die Konkatenation von Paaren (Attribut,Wert). Der Wert der Regel wird das Zielattribut darstellen.

Definition (Subgruppe) Eine *Subgruppe* G_p ist die Menge aller Datenitems, die von der Regel p überdeckt werden.

$$G_p = \{\vec{d_i} \in \mathcal{D} | p(\vec{a}^i) = 1\}$$

Das Komplement einer Subgruppe G ist $\bar{G}$ und enthält alle $\vec{d_i} \notin G$, d.h alle Datenitems, die von p nicht überdeckt werden. Mit n und $\bar{n}$ wird die Anzahl der Elemente in G und $\bar{G}$ gekennzeichnet, wobei $n = N - \bar{n}$.

Die Subgruppenentdeckung arbeitet in zwei Phasen, nämlich dem Auffinden der Kandidaten der Regeln sowie dem Bewerten der Regeln. Es werden zuerst Regeln mit einer kleineren Komplexität (allgemeine Regeln) aufgefunden, von denen im Laufe des Subgruppenentdeckungsprozesses immer komplexere (konkretere) Regeln generiert werden. Die Komplexität der Regeln ist durch die Anzahl der betrachteten Attribute bedingt.

Zuerst werden Kandidaten mit der Komplexität 1 aufgefunden. Danach werden Kandidaten mit höher Komplexität bottom-up generiert. Mit Hilfe einer *Qualitätsfunktion* werden die Regeln bewertet.

Qualitätsfunktion

Die *Qualitätsfunktion* [46, 58] spielt eine wichtige Rolle bei der Subgruppenentdeckung. Sie bestimmt die Güte der Regeln. Damit kann man die besten Regeln ausgeben.

Definition (Qualitätsfunktion) Eine *Qualitätsfunktion* ist eine Funktion φ: $\mathcal{P} \to \mathbb{R}$, die jeder Regel einen Wert (die Güte) zuweist.

Man kann die Auswahl der besten Regeln nach verschiedenen Kriterien treffen. Entweder werden die Regeln nach ihrer Güte sortiert und dann die besten k Regeln ausgegeben oder die Ausgabe wird durch einen minimalen Wert der Qualitätsfunktion beschränkt. Außerdem kann man eine minimale Menge von Regeln mit maximaler Qualität suchen. Diese Verfahren für die Auswahl der besten Regeln sollen hier nicht weiter betrachtet werden.

Es gibt viele *Qualitätsfunktionen* und es ist schwer zu sagen, welche allgemein am besten sind. Die Wahl der *Qualitätsfunktionen* wird von den Datenanalytikern getroffen. Entscheiden ist die aktuelle Aufgabe. Im folgenden Abschnitt wird eine Auswahl von Qualitätsfunktionen präsentiert.

- Coverage: liefert den Prozentanteil der Elemente der Datenmenge, die von einer Regel überdeckt sind.

  $$Cov(R) = \frac{TP+FP}{N}$$

  mit TP bezeichnet man, wie oft war eine Regel falsch war und richtig vorhergesagt

wurde. Dagegen gibt FP eine Aussage darüber, wie oft eine Regel wahr war, aber falsch vorhergesagt wurde.

- Precision: liefert den Anteil der tatsächlichen richtig vorhergesagten Regeln, wenn die Regel wahr war.
$$Pr(R) = \frac{TP}{FP+TP}$$
- Recall: liefert den Anteil aller wahren Regeln, die richtig vorhergesagt wurden, von allen wahr vorhergesagten Regeln.
$$Re(R) = \frac{TP}{TP+FN}$$
wobei FN die Anzahl der falschen Regeln ist, die falsch vorhergesagt wurden.
- Accuracy: liefert den Anteil der richtigen vorhergesagten Regeln von allen Regeln.
$$Acc(R) = \frac{TP+TN}{N}$$
- Weighted Relative Accuracy (WRAcc) [84]: Diese Gütefunktion gibt eine Aussage über die Ausgewogenheit zwischen der Überdeckung und der Genauigkeit einer Regel. WRAcc ist die am meisten verwendete Qualitätsfunktion bei der Subgruppenentdeckung.
$$WRAcc(R) = Cov(R)\left(\frac{TP+FN}{N} - \frac{TP+TN}{N}\right)$$
wobei TN die Anzahl der falschen Regeln ist, die richtig vorhergesagt wurden. Dieses Maß wird verwendet, da die einzelne Betrachtung von Accuracy zu falschen Schlüssen führen könnte.
- F1-Score [77]: das harmonische Mittel von Precision und Recall.
$$Fsr(R) = \frac{2*Pr(R)*Re(R)}{Pr(R)+Re(R)}$$

Suchstrategien

Die Anzahl der aufgefundenen Kandidaten bei der Subgruppenentdeckung kann exponentiell wachsen. Das kann beim Generieren der Regeln mit hoher Komplexität einen sehr hohen Speicher- und Rechenbedarf bedeuten. Deshalb können algorithmische Techniken eingesetzt werden, die den Suchraum verkleinern. Hierbei kann eine heuristische Suche durchgeführt werden, z.B. Beam-search [95]. Darüberhinaus kann man zwei Parameter einstellen, um den Suchraum zu beschränken oder die maximale Komplexität einer Regel festlegen. Weiterhin kann man nur bestimmte Kandidaten betrachten, beispielsweise die von einer Qualitätsfunktion am besten bewerteten Regeln.

Fazit

In diesem Kapitel haben wir uns mit maschinellen Lernmethoden, die zu dem unüberwachten Lernen gehören, beschäftigt. Vorgestellt wurden klassische Clustering und Subgruppenentdeckung Methoden. Die Methoden erzielen gute Ergebnisse auf kleinen Datenmengen. Für Big Data existieren verschiedene Ansätze, die diese Algorithmen erweitern, da-

mit sie parallel bzw. verteilt arbeiten. In den folgenden Abschnitten werden diese Ansätze erläutert.

4.3 Verteiltes Lernen

Eine Grundannahme des maschinellen Lernens ist die vollständige Verfügbarkeit des Datensatzes an einem Ort. Diese Annahme ist in vielen Fällen zutreffend, weil traditionell Datensätze auf einem leistungsstarken Computer gespeichert und auf Anfrage verarbeitet werden. Im Laufe der Zeit entstanden aber Anwendungsfälle, die maschinelles Lernen in einem verteilten Kontext betreiben und unter dem Begriff Verteiltes Lernen zusammengefasst werden. Die prominentesten Vertreter sind *wireless sensor networks* sowie Rechencluster.

Ein wireless sensor network ist ein Netzwerk aus datenerzeugenden Knoten. Üblicherweise wird angenommen, dass die einzelnen Knoten eine stark beschränkte Rechenleistung und Speicher haben. Jeder Knoten verfügt oder generiert z.B. durch Auslesen eines Sensors einen Teil der gesamten Datenmenge.

Hingegen wird bei Rechenclustern von potenziell hoher Rechenleistung und Speicherkapazität ausgegangen. Hier wird die Verteilung genutzt, um sehr große Datensätze zu bearbeiten, die auf einem einzelnen Rechner nicht praktikabel gespeichert und verarbeitet werden können.

Verteiltes Lernen kann somit als Querschnittsdisziplin aus Maschinellem Lernen und Verteilten Systemen verstanden werden und fußt auf den Grundlagen beider Gebiete. So weisen verteilte Algorithmen einige Besonderheiten auf, die bei sequentiellen Algorithmen üblicherweise nicht auftreten. Bei ihrem beim Entwurf sollten daher unter anderem folgende Aspekte [67] berücksichtigt werden:

- Rechenknoten: Gibt es verschiedene Rollen für die Rechenknoten? Ist ein gesonderter Koordinator-Knoten notwendig? Gibt es ein Minimum oder Maximum für die Zahl der beteiligten Rechenknoten?
- Nachrichtentypen: Welche Nachrichtentypen sind in welchen Phasen des Algorithmus erlaubt? Wie muss ein Knoten auf eine Nachricht in Abhängigkeit seines Zustands reagieren?
- Anforderungen an das Nachrichtentransportsystem: Ist es zum Gelingen des Algorithmus notwendig, dass Nachrichten zuverlässig ankommen?
- Konvergenz: Kann garantiert werden, dass alle Knoten ein gemeinsames Endergebnis in endlicher Zeit erreichen?
- Terminierungserkennung: Wann ist der Algorithmus beendet? Wie erkennt ein Rechenknoten die Terminierung?

- Netzwerkkosten: Wie viele Nachrichten werden im Worst-Case verschickt und wie groß ist das Gesamtvolumen der versendeten Daten?

Im Folgenden werden zwei populäre verteilte Lernalgorithmen vorgestellt, nämlich der Peer-to-Peer-K-Means (auch P2P-K-Means genannt) und Distributed random forests. Abschließend wird auf die Modellkompression mittels Fouriertransformation, ein weiteres nützliches Verfahren des Verteilten Lernens, eingegangen.

4.3.1 Peer-to-Peer-K-Means

Der P2P-K-Means von Bandyopadhyay et al. [9] ist ein verteilter Lernalgorithmus, der für wireless sensor networks konzipiert wurde. Er geht davon aus, dass der Datensatz bereits über die einzelnen Knoten verteilt und ein Verschieben der Daten zu aufwendig ist. Jeder Knoten durchläuft die folgenden Phasen:

Initialization. Ein herausgehobener Initiator-Knoten setzt den Algorithmus in Gang, indem er zufällig Startwerte für die K Zentren zieht und diese in einer Initialisierungsnachricht an alle Knoten schickt. Da angenommen wird, dass der Initiator selbst auch über eine Datenpartition verfügt, wechselt er wie alle anderen Knoten in die Computation-Phase.

Computation. Jeder Knoten führt eine Iteration des klassischen k-Means durch (siehe subsection 4.2.1) und erhält neue Positionen für die K Zentren. Dabei wird außerdem die Größe jedes Clusters in Bezug auf die eigenen Datenpunkte bestimmt. Anschließend wechselt der Algorithmus in die Polling-Phase.

Polling. Jeder Knoten zieht eine zufällige Auswahl aus allen beteiligten Knoten und erfragt deren neu ermittelte Zentren und Größen. Es wird gewartet, bis alle Antworten eingetroffen sind oder ein Timeout eintritt. Dann wird in die Merging-Phase gewechselt.

Merging. Die eigenen Zentren werden mit den Zentren aus den Antworten verglichen, wobei die jeweils mitgelieferten Clustergrößen eine Gewichtung liefern. Wenn die Zentren hinreichend nah beieinander liegen, wechselt der aktuelle Knoten in die Terminated-Phase und behält seine Zentren dauerhaft bei. Andernfalls beginnt der aktuelle Knoten mit der Computation-Phase die nächste Iteration.

Terminated. Die Terminated-Phase ist kein Terminierungszustand im engeren Sinn, da der Knoten weiterhin Poll-Nachrichten beantwortet.

In [25] wurde für den Worst Case der Speicherbedarf und das Netzwerkvolumen des Algorithmus untersucht. Sei n die Anzahl der Knoten, I die Anzahl an Iterationen, bis alle

Knoten zu Terminated gewechselt sind, L die größte Zahl an Nachbarn, die ein Knoten hat, und K die Anzahl der Zentren des k-means. Dann benötigt der P2P-K-Means auf allen Knoten zusammen $O(nI(K+L))$ Speicher und verursacht ein Netzwerkvolumen von $O(nILK)$.

4.3.2 Distributed random forests

Random forests werden im Bereich des maschinellen Lernens häufig verwendet und sind daher in vielen ML-Bibliotheken zu finden. Der Grundalgorithmus ist sequentiell, birgt aber das Potential, einige Arbeitsschritte verteilt auszuführen.

In Spark ML findet sich eine auf Rechencluster ausgelegte Implementierung, die sich folgende Ideen zunutze macht [19, 20]:

- **Trainieren ganzer Baum-Ebenen.** Es werden alle Knoten mit gleicher Tiefe gleichzeitig trainiert. Dadurch werden die Iterationen über die Trainingsdaten besser ausgenutzt und die Anzahl an benötigten Iterationen wird reduziert.
- **Quantilschätzungen.** Im sequentiellen Algorithmus werden die Daten sortiert. Dies ist bei einem verteilten Datensatz mit hohem Aufwand verbunden und wird vermieden. Stattdessen wird das benötigte Quantil des Datensatzes geschätzt und eine leichte Abweichung in Kauf genommen.
- **Vorberechnete Feature-Bins.** Kontinuierliche Features müssen in Halbräume unterteilt werden („if feature 11 <= 8.736“). Diese können zum Teil vorberechnet werden und sparen später in jeder Iteration Zeit ein.

Damit eignet sich diese Implementierung für das Ziel dieser PG, Big Data Analytics auf einem Rechencluster zu betreiben. Im chapter 14 untersuchen daher wir unter anderem, inwiefern die Random forest Implementierung von Spark ML von steigender Worker-Anzahl profitiert.

4.3.3 Kompression von Entscheidungsbäumen

Zum Bereich des Verteilten Lernen gehören neben Lernalgorithmen auch Verfahren, die den Umgang mit Modellen in Verteilten Systemen vereinfachen.

Das Kompressionsverfahren von Kargupta und Park [52] wendet die aus der Elektrotechnik bekannte Fouriertransformation auf einen gegebenen Entscheidungsbaum an. Dabei wird die Klassifizierungsfunktion durch eine gewichtete Summe von Basisfunktionen dargestellt. Der Nutzen dieses Verfahrens besteht zum Einen darin, dass die Basisfunktionen und die Gewichte sich mit weniger Aufwand im Netzwerk versenden lassen als eine ganze

Baumtopologie. Zum Anderen lässt sich aus dem Ergebnis der Fouriertransformation leicht ablesen, welche Basisfunktionen einen großen Einfluss auf die Klassifizierung haben und welche nur selten relevant sind. Dadurch kann der Nutzer entscheiden, ob Basisfunktionen mit wenig Einfluss überhaupt über das Netzwerk übertragen werden sollen. Der am Zielort rekonstruierte Entscheidungsbaum ist dann zwar keine exakte Kopie des Originals, enthält dafür aber nur die wichtigen Ebenen und ist in der Anwendung somit schneller.

4.4 Statisches und Inkrementelles Lernen

Grundlegend für das *statische* oder auch *batch* genannte *Lernen* ist, dass die Trainings-Daten vorher bekannt sind. Oftmals wird dies auch weiter eingeschränkt, indem angenommen wird, dass die Daten komplett in den Hauptspeicher passen. Da diese Annahme offensichtlich vieles vereinfacht, beruhen viele klassische Verfahren darauf.

Beim *inkrementellen* oder *online Lernen* kommen die Test-Daten nacheinander in der Reihenfolge ihres Entstehens, z.B. ihres Auftretens, Messens usw., beim Lerner an und werden dort sofort verarbeitet. Dabei wird so wenig wie möglich zwischengespeichert, was auch als *Data stream mining* bezeichnet wird.

Das auffälligste Problem beim statischen Lernen ist die Annahme, dass die Daten vollständig in den Hauptspeicher geladen werden können. Dieser ist relativ begrenzt und besonders im Big-Data-Umfeld übersteigen die Daten den zur Verfügung stehenden Platz um ein Vielfaches, z.B. umfangreiche Log-Files von großen Webseiten, Sensordaten, Internet of Things usw. Um Beschränkungen durch zu kleinen Hauptspeicher zu umgehen, gibt es auch Algorithmen bzw. Anpassung von bestehenden Algorithmen, die Sequenzen von der Festplatte lesen und auf diesen dann batch-artig lernen. Diese Klasse von Algorithmen sind zwar eine Mischung aus batch- und online-Lernen, werden aber meistens zum statischen Lernen gezählt. Wünschenswert wäre daher ein Online-Algorithmus, dessen Ergebnis äquivalent zu einem Ergebnisses eines Batch-Lerners wäre.

4.5 Concept Drift und Concept Shift

Beim kontinuierlichen Beobachten von Daten stellt man häufig fest, dass die Daten sich systematisch über einen bestimmten Zeitraum verändern bzw. *verschieben*. Dies kann durch Veränderungen in den Rohdaten an sich oder auch durch die Messgeräte verursacht werden, wenn sich diese zum Beispiel im Betrieb erwärmen und so bei gleichen Rohdaten dennoch unterschiedliche Werte liefern. Durch dieses Verschieben kann die Qualität der Klassifikation der angelernten Lernverfahren abnehmen, da die bisher verwendeten (Trainings-)Daten nun nicht mehr zu den neuen Messdaten passen.

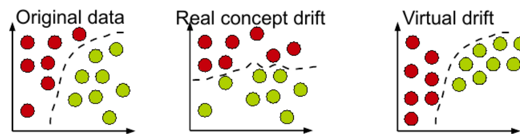


Abbildung 4.2: Unterscheidung Realer Drift vs. Virtueller Drift [37]

Daher wird sich in diesem Abschnitt etwas genauer mit *Concept Drift* bzw. *Concept Shift* beschäftigt, das heißt, die Auswirkungen dieser etwas näher erörtert, die unterschiedlichen Arten näher beschrieben und angesprochen, wie man das Verschieben erkennen kann [29].

Realer Drift vs. Virtueller Drift

Während sich die Daten verschieben, kann man im Wesentlichen zwei wichtige Fälle unterscheiden: Das Verschieben beeinträchtigt unsere Klassifikation oder es ist für die Klassifikation nicht weiter von Bedeutung. Betrachtet man alle Features über einer Menge von Rohdaten, so sind nicht immer alle Features entscheidend für die Klassifikation durch maschinelles Lernen. Oft sind die Algorithmen auch darauf ausgerichtet, eine möglichst einfache Unterscheidung, das heißt, mit möglichst wenigen Features, der Klassen zu finden. Findet nun ein Drift in einem oder mehreren Features statt, die zur Klassifikation nicht notwendigerweise gebraucht werden, ist der Drift nicht weiter relevant. In diesem Fall wird auch vom *virtuellen Drift* gesprochen (vgl. Figure 4.2 links und rechts). Verschieben sich die Daten jedoch so, dass ein zur Klassifikation nötiges Feature betroffen ist und die Daten die bisherigen Unterteilungskriterien nicht mehr erfüllen, spricht man von *realem Drift* (vgl. Figure 4.2 links und Mitte). In diesem Fall muss der Lerner angepasst oder gar neu antrainiert werden.

Auftreten von Shifts

Diese Veränderung der Daten kann zeitlich betrachtet recht unterschiedlich passieren (vgl. Abbildung 4.3):

Plötzlich (engl. *sudden / abrupt*) Ab einem bestimmten Zeitpunkt fallen die Daten einfach anders aus oder zeigen andere Charakteristika.

Schleichend (engl. *incremental*) Dies bezeichnet den Vorgang, wenn sich die Daten langsam in einen anderen Bereich verschieben.

Wiederauftretend (engl. *reoccuring concepts*) Die Daten alternieren zwischen zwei bestimmten Werten, wobei es keine festen Zeitpunkte für den Wechsel zwischen den Werten geben muss.

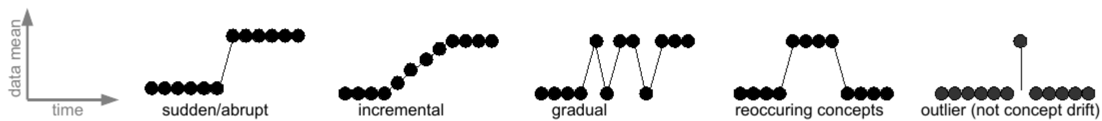


Abbildung 4.3: Schematische Darstellung vom unterschiedlichen Auftreten von Concept Drift [37]

Ausreißer (engl. *outlier*) Es können vereinzelte Datenpunkte außerhalb des erwarteten Bereiches liegen, dies ist jedoch **kein** Shift / Drift, sondern einfach eine (Mess-) Ungenauigkeit.

Erkennen von Shift

Das Erkennen von Shift verlangt ständiges Beobachten der Daten und Validieren der Klassifikationen. Plötzlich auftretende Veränderungen und auch Ausreißer lassen sich noch relativ einfach, auch durch einfache Algorithmen, erkennen. Schleichenden oder wiederauftretenden Shift zu erkennen erfordert dagegen komplexere statische Modelle oder Algorithmen. In beiden Fällen können maschinelle Lernmethoden angewendet werden, um einen möglichen Shift zu erkennen und um die Nutzer entsprechend zu informieren [37].

4.6 Learning with Imbalanced Classes

Bei vielen realen Klassifikationsproblemen geht es darum, seltene Ereignisse in einer Masse aus uninteressanten Vorkommnissen zu entdecken [36]. Beispiele hierfür sind zum Beispiel:

- Die Diagnose von seltenen Krankheiten auf Basis der Daten von größtenteils nicht betroffenen Patienten
- Die Erkennung von betrügerischen Finanztransaktionen
- Die Gamma-Hadron Separation, die ein entscheidender Teil der Analysekette in der Cherenkov Astronomie ist (siehe section 1.2)

Für die Klassifikation bedeutet dies, dass ein starkes Ungleichgewicht zwischen der Häufigkeit des Auftretens von Vertretern der unterschiedlichen Klassen besteht. Vielfach wird in diesem Zusammenhang auch von einer positiven, seltenen Minoritätsklasse und einer negativen, häufigen Majoritätsklasse gesprochen. Die damit verbundene Festlegung auf nur zwei Klassen ist ohne Beschränkung der Allgemeinheit möglich, da eine Problemstellung mit mehr Klassen immer als Klassifikationsaufgabe zwischen einer Gruppe von häufigen und einer Gruppe von seltenen Klassen gesehen werden kann. Von entscheidender Bedeutung ist hierbei das Verhältnis zwischen der Häufigkeit der beiden Klassen. Dies quantifiziert alle Aussagen, die hier getroffen werden.

4.6.1 Einfluss auf Klassifikatoren

Der Einfluss, den das Klassenungleichgewicht auf die Leistung von Klassifikatoren hat, wurde in verschiedenen Studien empirisch untersucht [51]. Die Ergebnisse lassen sich wie folgt zusammenfassen: Das Ungleichgewicht führt nicht dazu, dass Standardlerner zwangsläufig nicht mehr funktionieren, sondern sorgt vielmehr dafür, dass sich die Schwellen hinsichtlich der benötigten Menge an Trainingsdaten und der maximalen Modellkomplexität verschieben. Das Problem lässt sich also dadurch lösen, dass einfach die herkömmlichen Lerner mit zusätzlichen Trainingsdaten verwendet werden – ungünstigerweise ist das bei vielen Anwendungen aber ohnehin der limitierende Faktor.

4.6.2 Bewertung von Klassifikatoren

Ein wichtiger Punkt, der bei stark verschobenen Klassenverhältnissen bedacht werden muss, ist, wie Klassifikatoren eigentlich zu bewerten und zu vergleichen sind. Ein natürlicher Ansatz für die Darstellung der Performanz eines binären Klassifikators ist eine Wahrheitsmatrix, wie in Figure 4.4 dargestellt. Mit gegebenem Validationsdatensatz lässt sich eine solche Tabelle durch simples Zählen der Antworten des Klassifikators und der tatsächlichen Klassen befüllen. Offen ist aber, wie Leistungen verschiedener Lerner, also verschiedene Tabellen dieser Art, miteinander verglichen werden können.

	Positive Klasse	Negative Klasse
Positive Voraussage	Richtig positiv (TP)	Falsch positiv (FP)
Negative Voraussage	Falsch negativ (FN)	Richtig negativ (TN)

Abbildung 4.4: Schematischer Aufbau einer Wahrheitsmatrix

Ein verbreitetes Vergleichskriterium ist die Fehlerrate $ERR = (FP + FN)/(TP + FP + FN + TN)$, also der Anteil der Datenpunkte, die falsch klassifiziert wurden. Wenn die Minoritätsklasse nun aber sehr selten ist, können Lerner sehr geringe Fehlerraten erreichen, indem sie einfach alle Eingaben der Majoritätsklasse zuordnen. Da ein solcher Klassifikator aber vollkommen nutzlos ist, ist dieses Vorgehen bei stark verschobenen Klassenverhältnissen offensichtlich inadäquat. Dies hat damit zu tun, dass die Anzahl der falsch positiven und falsch negativen Datenpunkte in der Fehlerrate schlicht addiert werden. Da es von der negativen Klasse aber wesentlich mehr Instanzen gibt, werden die falsch positiven Datenpunkte die Fehlerrate höchstwahrscheinlich dominieren. Ein sinnvolles Vergleichskriterium muss daher die Klassifikatorleistung auf den einzelnen Klassen unabhängig von der Anzahl der jeweils vorliegenden Instanzen ins Verhältnis setzen.

Eine Möglichkeit, dies zu tun, ist, über die richtig-positiv-Rate $TP_{rate} = TP/(TP + FN)$

und die richtig-negativ-Rate $TN_{rate} = TN/(TN + FP)$, die angeben, welcher Anteil der jeweiligen Klassen richtig klassifiziert wurde. Um ein wirkliches Vergleichskriterium zu erhalten, müssen diese beiden Werte aber noch geeignet ins Verhältnis gesetzt werden. Ein Weg, die beiden Metriken zu kombinieren, ist die Visualisierung in einer Receiver Operating Characteristic (ROC) Grafik [30] wie in Abbildung 4.5.

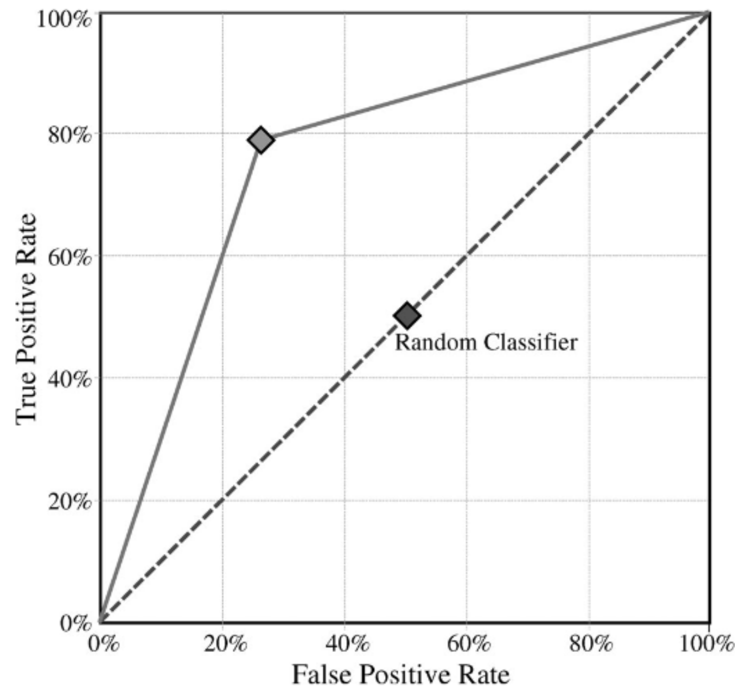


Abbildung 4.5: Eine ROC Kurve [36]

Die Klassifikatorleistung kann so als ein Punkt in diesem zweidimensionalen Raum dargestellt werden. Hierbei bedeutet ein Punkt, der sich weiter oben und weiter links befindet, einen strikt besseren Klassifikator. Mögliche daraus abgeleitete Metriken sind das arithmetische und geometrische Mittel von TP_{rate} und TN_{rate}:

$$AUC = \frac{TP_{rate} + TN_{rate}}{2}$$
$$Gmean = \sqrt{TP_{rate} * TN_{rate}}$$

Diese Metriken behandeln TP_{rate} und TN_{rate} symmetrisch. In manchen Anwendungsfällen ist die Performanz auf einer Klasse (üblicherweise der Minoritätsklasse) aber wichtiger als auf der anderen. In diesem Fall bietet es sich an, eine asymmetrische Metrik zu verwenden, etwa den Index of balanced accuracy (IBA) [38].

$$IBA = (1 + \alpha\,(TP_{rate} - TN_{rate})) * Gmean^2$$

Dieser Index führt den Asymmetriefaktor α ein, über den sich steuern lässt, wie viel stärker die TP_{rate} gegenüber der TN_{rate} gewichtet werden soll.

4.6.3 Verbesserung von Klassifikatoren

Zur Verbesserung der Performanz von Klassifikatoren auf unausgewogenen Trainingsdaten gibt es verschiedene Ansätze, die in drei Kategorien eingeteilt werden können: interne, externe, und auf Ensemble-Learning basierende Ansätze [36, 39].

Eine Möglichkeit besteht darin, den Lernalgorithmus selbst zu verändern. Denkbar wäre etwa, die Kostenfunktion anzupassen, um dafür zu sorgen, dass der Algorithmus seine Ausgabe mit Blick auf die gewählte Metrik optimiert. Diese auch als *intern* bezeichneten Ansätze stehen vor dem Problem, dass sie ein genaues Verständnis des Lernalgorithmus und des Problems erfordern. Des Weiteren beziehen sich die vorgenommenen Anpassungen jeweils nur auf einen Algorithmus und lassen sich in der Regel nicht auf andere Verfahren verallgemeinern.

Eine anderer, attraktiver Ansatz besteht daher darin, in einem Vorverarbeitungsschritt die Trainingsdaten so zu verändern, dass das Problem der unausgeglichenen Klassen geringer wird. Diese *externen* Ansätze haben den Vorteil, dass sie sich mit jedem beliebigen Klassifikator kombinieren lassen.

Over-Sampling

Eine mögliches externes Verfahren besteht darin, zusätzliche synthetische Instanzen der Minoritätsklasse in den Trainingsdatensatz einzufügen, um so den wenigen vorhandenen Datenpunkten mehr Gewicht zu verleihen. Die wird *Over-Sampling* genannt und kann durch verschiedene Strategien umgesetzt werden:

- Zufällig ausgewählte vorhandene positive Datenpunkte können repliziert werden
- Es kann zwischen vorhandenen Datenpunkten interpoliert werden, um Instanzen zu erzeugen, die neu, aber gleichzeitig konsistent mit den bisherigen Daten sind
- Andere Ansätze sind möglich, etwa kann versucht werden, Datenpunkte in der Grenzregion der Klasse zu erzeugen

Alle diese Ansätze haben ihre Vor- und Nachteile, abhängig davon, ob die über die Daten getroffenen Annahmen stimmen oder nicht. Ein übergreifendes Problem ist aber das *Overfitting*, also das Phänomen, dass ein Klassifikator die spezifische Verteilung der Trainingsdaten lernt, anstatt der dahinter liegenden Muster, und deswegen schlecht auf andere Daten generalisiert. Dadurch, dass die wenigen Datenpunkte der Minoritätsklasse beim Oversampling vervielfacht werden, wird dieses Problem verstärkt. Ein weiterer Nachteil ist der erhöhte Rechenaufwand durch die künstliche Vergrößerung des Trainingsdatensatzes.

Under-Sampling

Das dem Over-Sampling entgegengesetzte Verfahren wird *Under-Sampling* genannt und besteht darin, zufällig Instanzen der Majoritätsklasse aus dem Trainingsdatensatz zu löschen. Der Effekt ist auch hier, dass das Ungleichheitsverhältnis so künstlich verringert wird. Auch hierfür gibt es verschiedene Umsetzungsmöglichkeiten:

- Entfernen von zufälligen negativen Datenpunkten
- Entfernen von „redundanten“ Datenpunkten, also etwa solchen, in deren Nähe sich noch andere Punkte der selben Klasse befinden
- Entfernen von Datenpunkten in der Grenzregion zur Minoritätsklasse

Der große Nachteil dieser Verfahren ist, dass durch das Löschen von Datenpunkten unter Umständen wichtige Informationen verloren gehen, und die Klassifikatorleistung dadurch abnimmt. Insgesamt lässt sich aber sagen, dass beide Resampling-Varianten in aller Regel zu einer Leistungssteigerung führen. Dank ihrer universellen Einsetzbarkeit sind diese Verfahren daher sehr attraktiv.

Ensemble Learning

Ein weiterer Ansatz besteht darin, die in section 4.1 vorgestellten Ensemble Learning Verfahren zu adaptieren. Auch hierzu gibt es verschiedene Strategien, die allesamt das Ziel haben, der Minoritätsklasse ein größeres Gewicht zu verleihen. Einige Beispiele sind:

- *Over-/Under-Bagging.* Bei dieser Variante des Bagging werden die Teildatensätze nicht zufällig gezogen, sondern unter Benutzung von Over-/Under-Sampling
- *SMOTEBoost.* Diese Variante von AdaBoost (algorithm 2) generiert nach jeder Iteration durch Interpolation zusätzliche Datenpunkte und fügt diese in den Datensatz ein
- *AdaCost.* Diese andere Variante von AdaBoost verändert die Updatefunktion der Gewichte so, dass positive Datenpunkte schneller an Gewicht zunehmen als negative

Welches Verfahren das beste ist, lässt sich letztendlich nur durch den empirischen Vergleich entscheiden. Die durchgeführten Studien deuten aber darauf hin, dass Resampling-Verfahren in der Regel lohnenswert sind.

4.7 Feature Selection

Feature Selection (Merkmalsauswahl) versucht, möglichst geeignete Merkmale für ein gegebenes Vorhersageproblem zu identifizieren. Als ungeeignet betrachtete Merkmale können

ignoriert werden, wodurch sich die Dimensionalität der Daten reduzieren lässt. Dabei wird die Auswahl nur unter den Originalmerkmalen vorgenommen (die hier nicht betrachtete Merkmals-Extraktion hingegen erzeugt neue Merkmale, um die Datendimensionalität zu reduzieren).

Die Vorteile von Dimensionsreduktion und Feature Selection insbesondere werden in subsection 4.7.1 vorgestellt. Es wird eine formale Problemstellung aus den Eigenschaften abgeleitet, die ein „geeignetes“ Merkmal erfüllen sollte (siehe subsection 4.7.2). Eine Übersicht der Ansätze zur Feature Selection wird vorgestellt und Qualitätsmerkmale von Auswahl-Algorithmen werden identifiziert (siehe subsection 4.7.3). Als prominentes Beispiel wird der korrelationsbasierte Algorithmus CFS (Correlation-based Feature Selection) nach Hall [43] intensiv betrachtet (subsection 4.7.5). Dessen Erweiterung zu Fast-Ensembles wird ebenfalls vorgestellt (siehe subsection 4.7.6).

4.7.1 Vorteile

Die Reduktion der Datendimensionalität kann im überwachten Lernen sowohl die Trainingszeiten als auch die Anwendungszeiten der verwendeten Modelle reduzieren. Die trainierten Modelle sind aufgrund der geringeren Dimension kompakter und damit, falls es der Modelltyp hergibt, leichter interpretierbar. Ein besonderer Vorteil der Dimensionsreduktion ist aber, dass dem Fluch der hohen Dimension entgegengewirkt werden kann. Dieser besagt, dass hochdimensionale Modelle bei geringer Anzahl verfügbarer Beispiele stark überangepasst werden. Überangepasste Modelle generalisieren schlecht auf unbekannten Daten und resultieren daher in schlechter Vorhersage-Performanz. Dimensionsreduktion schränkt die Variabilität der Modelle ein, sodass der Informationsgehalt kleiner Stichproben besser repräsentiert und damit die Generalisierungsfähigkeit erhöht wird.

Besteht die Dimensionsreduktion aus der Auswahl von Originalmerkmalen, können weitere Vorteile gewonnen werden. So lassen sich Datenvisualisierungen auf wichtige Merkmale fokussieren, was das Verständnis der Daten erhöhen kann. Außerdem müssen bei zukünftigen Datenerfassungen nicht alle Merkmale erfasst werden, was die Kosten solcher Datenerfassungen senken kann. Natürlich werden auch, wenn pro Beispiel weniger zu speichern ist, auch die Speicheranforderungen geringer ausfallen.

Überdies hat sich Feature Selection auch als eigenständiges bzw. primäres Analysewerkzeug etabliert: Einige Probleme sind bereits dadurch gelöst, dass wichtige Merkmale identifiziert werden. Beispielsweise sollen in der Analyse von Genexpressionsdaten für Krankheiten relevante Gene ausfindig gemacht werden. Die Ausprägungen der Gene stellen Merkmale dar. Mit Krankheiten stark korrelierte Ausprägungen können ein Indiz für einen Zusammenhang sein.

Im Anwendungsfall interessiert uns die Auswahl von Features, da bestehende Analysen eine große Anzahl teils redundanter Merkmale extrahieren. Die Relevanz dieser Merkma-

le für die Gamma-Hadron-Separation und die Energy Estimation ist fraglich. Wenn wir Merkmale identifizieren können, die nicht weiter betrachtet werden müssen, können wir die Analyse beschleunigen, indem wir die Berechnung unwichtiger Merkmale überspringen. Sämtliche der oben genannten Vorteile können ebenfalls geltend gemacht werden.

4.7.2 Problemstellung

Nützliche Merkmale zeichnen sich durch zwei Eigenschaften aus: Sie sollten zum Einen für das gegebene Vorhersageproblem relevant sein, also eine gewisse Vorhersagekraft besitzen. Möglicherweise ergibt sich diese Vorhersagekraft nur durch Zusammenspiel mit anderen Merkmalen. Zum Anderen sollte die durch das Merkmal kodierte Information sich nicht mit der Information anderer Merkmale überschneiden. Selektierte Merkmale sollten also nicht redundant zueinander sein.

Es lässt sich daher nicht für jedes Merkmal isoliert entscheiden, ob es gewählt werden sollte oder nicht. Wir müssen die Qualität von Merkmalsmengen (genauer: Teilmengen der Original-Merkmalsmenge) abschätzen. Koller und Sahami [56] prägten die Vorstellung einer optimalen Merkmalsmenge wie folgt:

Definition 4.1 (Optimale Merkmalsauswahl) *Die minimale Teilmenge $G \subseteq F$ der Original-Merkmale F, so dass:*

$$\mathbf{P}(C \mid G = f_G) \text{ und } \mathbf{P}(C \mid F = f) \text{ so ähnlich, wie möglich}$$

betrachten wir als optimal, wobei $\mathbf{P}$ die wahre Wahrscheinlichkeits-Verteilung über den Klassen C, f eine Realisierung von F und f_G die Projektion von f auf G.

Damit ist die optimale Merkmalsauswahl eine minimal große Menge, welche die (wahre) Wahrscheinlichkeitsverteilung über der Zielvariable so gut wie möglich erhält. Es soll also das zu lösende Vorhersageproblem durch die Beschränkung auf eine Teilmenge der Merkmale nicht wesentlich verzerrt werden. Eine oft verwendete alternative Definition beschreibt die optimale Auswahl als die minimal große Menge, welche die Vorhersage-Performanz maximiert. Damit ist allerdings die wahre Verteilung ignoriert und das eigentliche Problem nicht korrekt wiedergegeben.

Da es bei Merkmalsauswahl um den Erhalt der wahren Verteilung geht (welche wir nicht kennen), lässt sich das Problem im Allgemeinen nicht optimal lösen. Selbst die Verwendung der alternativen Definition über die Vorhersageperformanz lässt Merkmalsauswahl nicht zu einem einfachen Problem werden: Um das Zusammenspiel aller Merkmale zu berücksichtigen, müssten wir alle möglichen Merkmalsmengen ($2^{|F|}$ Möglichkeiten) ausprobieren, was für viele Probleme schlicht nicht realisierbar ist. Daher ist allen Merkmalsauswahl-Algorithmen gemein, dass sie einige Merkmalsmengen (Kandidaten) heuristisch auswerten.

Kandidaten werden dabei durch eine Such-Strategie (z.B. Vorwärts-Suche, randomisierte Suchen, ...) im Raum der möglichen Lösungen erzeugt.

4.7.3 Arten von Algorithmen

Algorithmen zur Auswahl von Merkmalen unterscheiden sich hauptsächlich durch die von ihnen genutzte Heuristik zur Bewertung möglicher Lösungen. Oft genannte Arten von Algorithmen sind:

Wrapper nutzen die Accuracy (Anteil korrekter Vorhersagen auf Testdaten) von Modellen, die mit der betrachteten Merkmalsmenge trainiert wurden. Es wird also in jedem Suchschritt durch den Raum möglicher Teilmengen ein Modell eingepasst, was einen hohen Berechnungsaufwand mit sich führt. Durch Wrapper ausgewählte Merkmale sind allerdings nahe an der optimalen Merkmalsmenge, da die Accuracy auf unbekannten Daten eine gute Abschätzung für die Erhaltung der wahren Verteilungsfunktion darstellt.

Eingebettete Methoden verwenden interne Informationen von Modellen, die auf der gesamten Merkmalsmenge eingepasst werden. So können beispielsweise Merkmale gewählt werden, die in einem Random Forest viele oder besonders gute Splits erzeugen. Eingebettete Methoden sind effektiv, da der Raum möglicher Merkmalsmengen und Modelle zugleich durchsucht wird, verzerren die Lösung aber zum verwendeten Modell hin. Durch einen Random Forest ausgewählte Merkmale können z.B. für die Verwendung in einer SVM ungeeignet sein.

Filter agieren unabhängig von jedem Lernalgorithmus durch explizite Verwendung von Heuristiken, wie etwa Korrelationen zwischen Merkmalen. Sie sind daher besonders effektiv.

Über diese Arten hinaus existieren hybride Verfahren, die etwa Filter für eine Vorauswahl verwenden, um im Anschluss einen Wrapper die Endauswahl treffen zu lassen. Wir wollen hier Filter fokussieren, da sie das allgemein effektivste Verfahren darstellen. Durch Berücksichtigung von zusammenspielenden Features können sie bereits sehr gute Ergebnisse liefern. Die Qualität eines Algorithmus lässt sich überdies an folgenden Eigenschaften messen [80]:

Begünstigung des Lernens Die Accuracy des trainierten Modells sollte im besten Fall erhöht, aber zumindest nicht wesentlich gesenkt werden.

Geschwindigkeit Der Auswahl-Algorithmus sollte in der Anzahl der Originalmerkmale skalierbar sein.

Multivarianz Das Zusammenspiel von Merkmalen (bzgl. Vorhersagerelevanz und Redundanz) sollte berücksichtigt werden.

Stabilität Die ausgewählte Merkmalsmenge sollte robust gegenüber der Varianz der verwendeten Daten sein. Insbesondere sollten für unterschiedliche Stichproben nicht gänzlich unterschiedliche Merkmale ausgewählt werden. Nur stabile Verfahren können ein Vertrauen in die Auswahl schaffen, das es erlaubt, Feature Selection zur Wissensgenerierung zu verwenden.

4.7.4 Korrelation als Heuristik

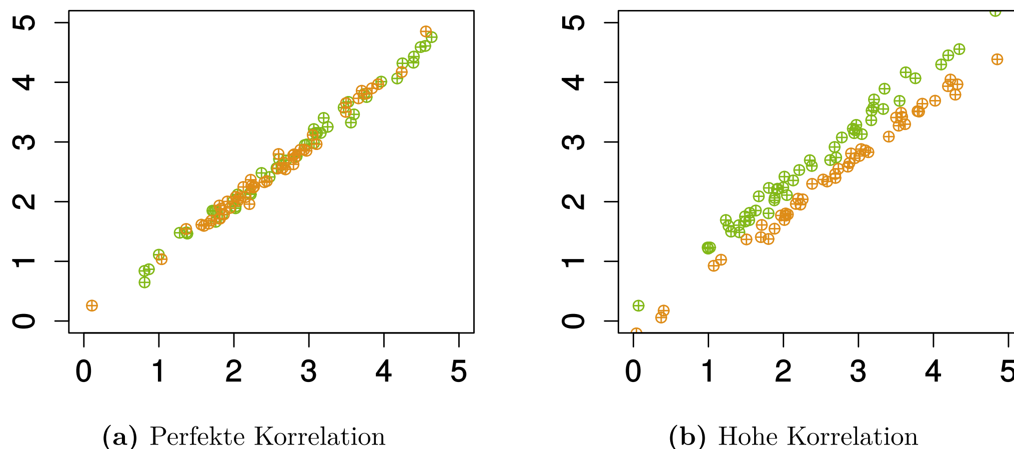
Bevor wir in subsection 4.7.5 mit CFS ein korrelationsbasiertes Verfahren zur Merkmalsauswahl kennen lernen, wollen wir zunächst die heuristische Natur von Korrelation zwischen Merkmalen bzw. Korrelation zwischen Merkmalen und der Zielvariablen als Maß für die Qualität einer Merkmalsmenge untersuchen.

Korrelation und Redundanz

Figure 4.6 zeigt zwei mögliche Verteilungen von Beispielen in $\mathbb{R}^2$. Mit den beiden Dimensionen gibt es also zwei Merkmale, von denen möglicherweise eines ausgewählt werden könnte. Wir wollen mit der Auswahl die Klasse von Beispielen vorhersagen, wobei Beispiele entweder aus der orangenen oder der grünen Klasse stammen. Bei perfekter Korrelation zwischen den Merkmalen (Figure 4.6a) ist es egal, ob wir ein Merkmal oder beide verwenden, die Klassen lassen sich nicht trennen. Damit sind die Merkmale redundant zueinander. Bei einer „lediglich“ sehr hohen Korrelation muss es jedoch nicht sein, dass beide Merkmale redundant zueinander sind: In Figure 4.6b erlaubt die Verwendung beider Merkmale eine lineare Separation der Klassen, was mit nur einem der Merkmale nicht möglich wäre. In diesem Fall hinkt die Heuristik also. Für reale Probleme funktioniert Korrelation als Heuristik aber sehr gut [42].

Korrelation und Kausalität

Weiterhin ist anzumerken, dass Korrelation nicht gleich Kausalität ist: Welches zweier Merkmale der Auslöser für die Ausprägung des anderen Merkmals ist, kann Korrelation nicht erfassen. Möglicherweise sind die Ausprägungen beider Merkmale auch gemeinsamer Effekt eines dritten Merkmals. Die Offenlegung (probabilistisch) kausaler Zusammenhänge kann tiefgehende Erkenntnisse bringen, ist aber außerhalb dieser Betrachtung von Merkmalsauswahl (für weitere Informationen, siehe [41]).



(a) Perfekte Korrelation

(b) Hohe Korrelation

Abbildung 4.6: Korrelation als Heuristik

4.7.5 CFS

Wir wollen im Folgenden einen prominenten Vertreter von korrelationsbasierten Filter-Verfahren zur Merkmalsselektion auf seine Qualität hin untersuchen, die Correlation-based Feature Selection nach Hall [43].

Idee

Die Idee von CFS ist recht simpel: In jedem Schritt $j+1$ wird das Merkmal $f \in F \setminus F_j$ mit dem besten Verhältnis von Relevanz und Redundanz zur bisherigen Auswahl F_j hinzugenommen. Damit beschreibt CFS eine Vorwärtssuche durch den Raum möglicher Merkmalsmengen. Relevanz und Redundanz werden heuristisch ermittelt, indem die Relevanz als Korrelation zwischen Merkmal f und Zielvariablen y und die Redundanz als Korrelation zwischen Merkmal f und Merkmalen $g \in F_j$ der vorherigen Auswahl F_j abgeschätzt wird:

$$F_{j+1} = F_j \cup \left\{ \underset{f \in F \setminus F_j}{\arg\max} \frac{Cor(f,y)}{\frac{1}{j}\sum_{g \in F_j} Cor(f,g)} \right\}$$

Für das Maß Cor existieren verschiedene Definitionen basierend darauf, ob die Eingabe-Merkmale numerisch oder nominal sind (siehe [41]). Diese sollen hier aber nicht weiter betrachtet werden.

Beispiel-Ablauf

Figure 4.7 zeigt einen Beispiel-Ablauf des CFS-Algorithmus: Im ersten Schritt wird für jedes Merkmal dessen Korrelation mit der Zielvariablen bestimmt. Das Merkmal mit der

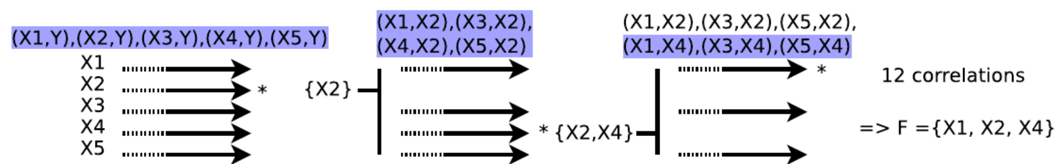


Abbildung 4.7: Beispiel-Ausführung CFS [80]

höchsten Korrelation (hier `X2`) wird gewählt. In den weiteren Schritten müssen zusätzlich die Korrelationen mit zuvor gewählten Merkmalen berechnet werden, um die Redundanz abzuschätzen. Einmal berechnete Korrelationen können gecached werden, um das Verfahren zu beschleunigen. Dies passiert hier mit dem Korrelationen `(X1,X2)`, `(X3,X2)` und `(X5,X2)`. Diese müssen kein zweites Mal berechnet werden. Das Verfahren kann bei einer festgelegten Anzahl Merkmale terminieren oder wenn keine relative Verbesserung größer als eine festgelegte Konstante erreicht wird.

Qualität

Der CFS-Algorithmus ist vielversprechend: Experimente zeigen, dass sich die Accuracy von auf den Merkmalen trainierten Modellen erhöhen lässt [43]. Durch die höchstens einmalige Berechnung der $(|F| + 1)^2$ Korrelationen zwischen Merkmalen und Zielvariablen ist der Algorithmus zudem schnell. Da er das Zusammenspiel von Merkmalen bezüglich ihrer Redundanz berücksichtigt, erfüllt er auch das Multivarianz-Kriterium. Ein Problem von CFS ist allerdings, dass alle verwendeten Maße Cor auf Varianz basieren und damit anfällig für eine hohe Varianz der Stichprobe und gegenüber Ausreißern sind. CFS ist also nicht stabil.

4.7.6 Fast-Ensembles

Um die Stabilität eines Klassifikators zu erhöhen, lassen sich mehrere Klassifikatoren zu einem Ensemble zusammenfassen (siehe section 4.1). Dieselbe Idee lässt sich auf Feature Selection übertragen, um die Stabilität der ausgewählten Merkmalsmengen zu erhöhen [79]. Dazu wird ein Merkmalsauswahl-Algorithmus auf unterschiedlichen Teilmengen der Stichprobe trainiert, wodurch mehrere Merkmalsmengen erzeugt werden. Die aggregierte Merkmalsauswahl ist die Merkmalsmenge, die aus häufig selektierten Features besteht.

Problematisch bei der Anwendung von Ensembles zur Feature Selection ist, dass im Ensemble mehrere Merkmalsmengen ausgewählt werden müssen. Damit sind Ensembles üblicherweise nicht schnell. Für CFS-Ensembles haben Schowe und Morik [80] aber ein Verfahren entwickelt, dass durch die Bildung eines Ensembles nahezu keine zusätzliche Laufzeit erzeugt. Der Fast-Ensembles genannte Merkmalsselektor besitzt damit alle Vorteile von CFS, ist aber zudem stabil (CFS wurde bereits in subsection 4.7.5 kennen gelernt).

Idee

Die Grundlegende Idee zur Beschleunigung von CFS-Ensembles ist, die Korrelations-Maße *Cor* in eine Summe aus voneinander unabhängigen Teilsummen aufzuspalten. Die Teilsummen können dann wiederverwendet werden, um alle im Ensemble benötigten Abschätzungen der Korrelation zu berechnen: Dass CFS im Ensemble ausgeführt wird, erzeugt dann kaum zusätzliche Laufzeit. Alle Abschätzungen einer Korrelation können wie im Single-CFS in einem Durchlauf über die Stichprobe erzeugt werden.

Wir wollen beispielhaft die Zerlegung des Pearson's Correlation Coefficient in unabhängige Teilsummen betrachten. Die Idee ist aber auch auf alle anderen in CFS verwendeten Maße für Korrelation anwendbar.

$$Cor_{pcc}(X,Y) \;=\; \frac{Cov(X,Y)}{\sqrt{Var(X)\cdot Var(Y)}} \qquad \text{(Pearson's Correlation Coefficient)}$$

Wobei $Cov(X,Y) \;:=\; E\big[(X-E(X))(Y-E(Y))\big] \underset{\text{displ. law}}{=} E(XY)-E(X)E(Y)$.

Wegen $Var(X) = Cov(X,X)$ beschränken wir unsere Betrachtungen im Folgenden auf Cov, welches wir anhand der gegebenen Beispiele $(x_i, y_i), 1 \leq i \leq n, x_i \in X, y_i \in Y$ schätzen wollen:

$$\begin{aligned}\hat{Cov}(X,Y) &= (\frac{1}{n}\sum_{i=1}^{n} x_i y_i) - -\underbrace{(\frac{1}{n}\sum_{i=1}^{n} x_i)(\frac{1}{n}\sum_{i=1}^{n} y_i)}_{\star} \\ &= \frac{1}{n}(\underbrace{\sum_{i=1}^{m_1} x_i y_i}_{s_1(X,Y)} + \underbrace{\sum_{i=m_1+1}^{m_2} x_i y_i}_{s_2(X,Y)} + \cdots + \underbrace{\sum_{i=m_{e-1}+1}^{n} x_i y_i)}_{s_e(X,Y)} - \star\end{aligned}$$

Wir sehen: Es lassen sich voneinander unabhängige Teilsummen $s_j(X,Y), 1 \leq j \leq e$ durch Partitionierung der Beispiele an willkürlichen Grenzen m_j erzeugen. Der mit $\star$ bezeichnete Term wird analog zum dargestellten ersten Term in die Teilsummen $s_j(X)$ und $s_j(Y)$ zerteilt. Bei der ebenfalls analogen Zerteilung der Varianz-Schätzungen $\hat{Var}$ werden zusätzlich die Teilsummen $s_j(X^2)$ und $s_j(Y^2)$ erlangt.

Um eine Menge von e Ensemble-Schätzungen zu erzeugen, brauchen lediglich für jede Schätzung die j-ten unabhängigen Teilsummen weggelassen werden. Damit ist der j-te Teil der Stichprobe im j-ten Teil des Ensembles ignoriert. Alle anderen Teilsummen werden aufaddiert, um die Gesamtsummen zu ergeben, mit denen sich die Schätzung von Cor_{pcc} berechnen lässt. Wir erhalten e unterschiedliche Schätzungen für die Korrelation zweier Merkmale bzw. eines Merkmals mit der Zielvariablen. Figure 4.8 fasst die Schätzung der Korrelation im Ensemble zusammen.

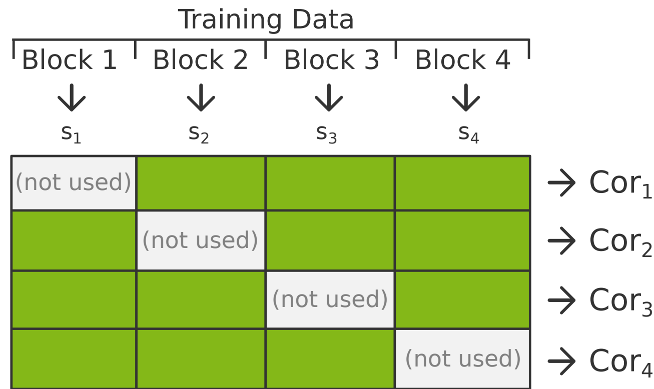


Abbildung 4.8: Berechnung von Ensemble-Korrelationen in Fast-Ensembles

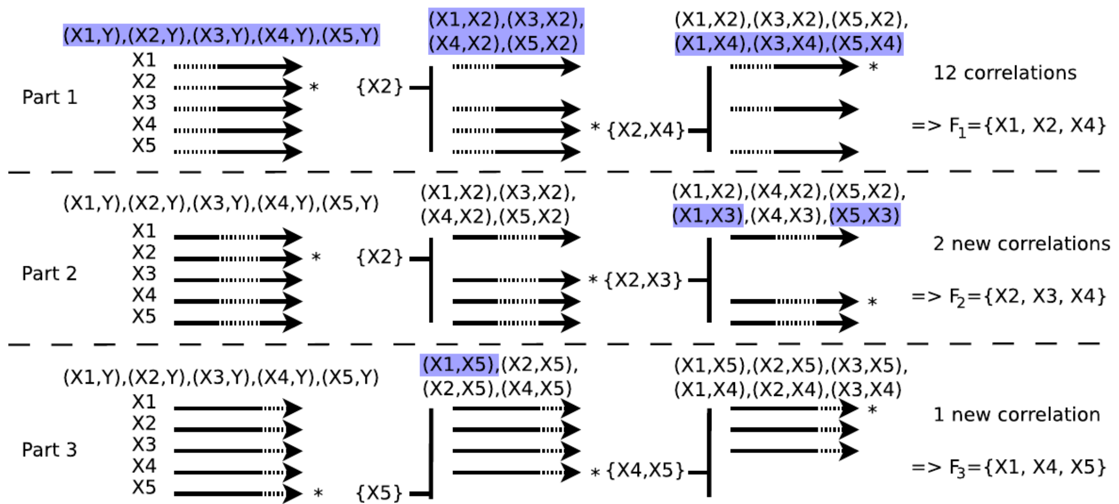


Abbildung 4.9: Beispiel-Ausführung Fast-Ensembles [80]

Beispiel-Ausführung

Es werden nun, wie im Single-CFS, einmal berechnete Korrelationen gecached, sodass sie kein zweites Mal berechnet werden müssen. Fast-Ensembles berechnen durch das oben vorgestellte Schema jedoch nicht nur eine Ensemble-Schätzung pro Korrelation, sondern gleich alle Schätzungen des Ensembles.

Figure 4.9 stellt dar, wie dadurch bei Einpassung eines Ensembles nur wenige zusätzliche Korrelationen (im Gegensatz zum Single-CFS) berechnet werden müssen. Part 1 in der Abbildung ist bereits aus subsection 4.7.5 bekannt. Part 2 und 3 müssen nun ihre Schätzungen der Korrelationen mit der Zielvariablen nicht mehr berechnen, da diese bereits durch Part 1 auf Basis der unabhängigen Teilsummen mitberechnet wurden. Auch andere Korrelationen können ohne Mehraufwand wiederverwendet werden. Da die unterschiedlichen Schätzungen unterschiedliche Entscheidungen des Algorithmus hervorrufen können, gibt es natürlich einige zusätzlich zu berechnende Korrelationen (`(X1,X3)`, `(X5,X3)` und

`(X1,X5)`). Im Gegensatz zu einer kompletten Neuberechnung aller Korrelationen stellt das Verfahren aber eine enorme Beschleunigung dar. Damit erfüllen Fast-Ensembles alle in subsection 4.7.3 vorgestellten Qualitätskriterien.

4.8 Sampling und Active Learning

Bisher haben wir uns in diesem Kapitel mit den eigentlichen Lernverfahren beschäftigt. Zum Beispiel haben wir gelernt, was ein Modell ist, wie Merkmale ausgewählt werden etc. Jetzt wollen wir zum Abschluss noch das sogenannte *Sampling* betrachten. Wollen wir einen Algorithmus verwenden, um ein Modell zu lernen, stellt sich nämlich die Frage, welche Daten wir diesem überhaupt übergeben und auf welchen Teilen des Datensatzes das Modell angelernt werden soll. Angenommen, wir haben einen Datensatz der Form $(x_1, y_1), \ldots, (x_n, y_n)$ gegeben, wobei $\hat{x}_i$ ein Merkmalsvektor und y_i die Klasse des Vektors ist. Diese Daten wollen wir nun nutzen, um unser Modell zu trainieren.

4.8.1 Der naive Ansatz

Am einfachsten bzw. logischsten erscheint es nun, den gesamten Datensatz zum Lernen zu verwenden. Schließlich bedeuten mehr Daten auch mehr Informationen und je mehr Informationen wir dem Lernverfahren geben, desto besser sollte unser gelerntes Modell sein.

Das Problem bei diesem Ansatz ist, dass wir nicht nur ein Modell lernen wollen, sondern unser gelerntes Modell auch testen müssen. Schließlich müssen wir auch herausfinden können, wie gut das Modell überhaupt ist, gerade wenn wir zwischen verschiedenen entscheiden müssen. Wir brauchen also definitiv einen Datensatz, an dem wir das Gelernte ausprobieren und testen können. Verwenden wir hierfür nämlich den bereits zum Lernen verwendeten Datensatz einfach nochmal, werden unsere gelernten Modelle zwar alle erstaunlich akkurat sein, allerdings testen wir auch nur, wie gut sie darin sind, den Datensatz, auf dem sie basieren, zu klassifizieren. Wir lernen also nicht die „wahre“ Klassenverteilung, sondern nur die Testdaten auswendig. Dieses Problem wird als *Overfitting* bezeichnet. Was wir brauchen, ist ein zweiter, unabhängiger Datensatz, auf dem wir unsere Modelle testen können. Ein besserer Ansatz wäre daher, die gegeben Daten vor dem Lernen zufällig in Test- und Trainingsdaten zu unterteilen. Eine typische Einteilung hierfür wäre, zwei Drittel der Daten zum Lernen zu nutzen und das gelernte Modell dann auf dem letzten Drittel zu testen. Und tatsächlich gibt uns dieser Ansatz erstmal die Möglichkeit, ein Modell zu lernen und es dann fair beurteilen zu können. Schade ist nur, dass jetzt ein beträchtlicher Anteil unserer Daten gar nicht zum Lernen verwendet wird und somit Informationen ungenutzt bleiben.

4.8.2 Re-Sampling

Nachdem wir die Probleme dieser simpleren Ansätze betrachtet haben, überlegen wir nun, wie diese vermieden werden können. Dazu betrachten wir das sogenannte Re-Sampling in Form der Methoden der k-fachen Kreuzvalidierung und des Bootstrappings, die uns Lösungen für diese Probleme geben können. Die Idee dieser Ansätze ist, die Daten zwar wie zuvor in Trainings- und Testdaten zu teilen, dies aber dann mehrmals zu wiederholen.

k-fache Kreuzvalidierung

Bei der k-fachen Kreuzvalidierung wird unsere Datenmenge in k Teile geteilt, von denen dann $k-1$ zum Trainieren des Klassifikators verwendet werden. Das gelernte Modell wird dann auf dem letzten Teil getestet. Dieser Vorgang wird k mal durchgeführt, wobei jeder Teil des Datensatzes einmal zum Testen verwendet wird. Schließlich wird die durchschnittliche Fehlerrate der einzelnen Modelle betrachtet, um die erhaltenen k Klassifikatoren zu bewerten. Durch diese mehrfache Ausführung haben wir erreicht, dass wir zwar immer auf unabhängigen Testdaten testen konnten, aber trotzdem jeder Teil der Daten gleich starken Einfluss auf das Modell hat.

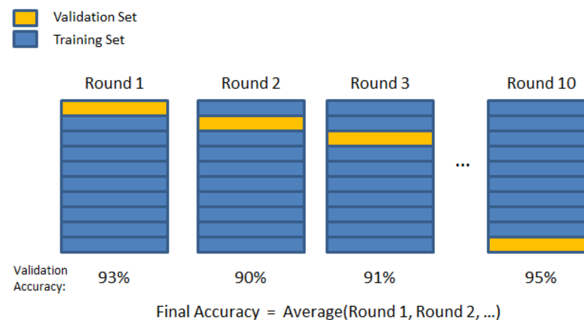


Abbildung 4.10: k-fache Kreuzvalidierung [68]

Bootstrapping

Ein alternativer Ansatz zur Kreuzvalidierung ist das sogenannte Bootstrapping. Hier wird die Datenmenge nicht in k Blöcke unterteilt, sondern es wird zufällig eine Menge von Daten mit zurücklegen aus dem Datensatz gezogen. In der gewählten Menge von Daten können nun also bestimmte Daten mehrfach auftreten, alle Daten, die nie gewählt wurden, werden wie zuvor zum Testen verwendet. Der Vorteil dieser Methode ist, dass sich bessere Rückschlüsse auf die Verteilung, die den Daten zugrundeliegt, machen lassen, allerdings werden auch deutlich mehr Durchläufe benötigt. Bootstrapping ist also in der Regel deutlich rechenintensiver.

Data: Zeiger auf große Beispielmenge $\mathcal{E}$
Größe m der Arbeitsmenge
Anzahl der Iterationen k
Resampling Intervall R
Gewichtungsregel $W : X \times Y \times \mathbb{R} \to \mathbb{R}$
Result: Modell h: $X \to \mathbb{R}$
Initialisiere Arbeitsmenge $\mathcal{E}_0$
Initialisiere Gewichte $w_{0,i} := 1$
for $t = 1, .., k$ **do**
  if $t/R \in \mathbb{N}$ **then**
    $\mathcal{E}_t := random_subset(\mathcal{E}, m)$
    $w_0, i := 1 \; \forall i \in \{1, ..., m\}$
    for $j = 1, .., t-1$ **do**
      $\forall (x_i, y_i) \in \mathcal{E}_t : w_{j,i} := w_{j-1,i} \cdot W(x_i, y_i, h_j(x_i))$
    end
  end
  else
    $\mathcal{E}_t := \mathcal{E}_{t-1}$
    if $t > 1$ **then**
      $\forall (x_i, y_i) \in \mathcal{E}_t : w_{t,i} := w_{j-1,i} \cdot W(x_i, y_i, h_{t-1}(x_i))$
    end
  end
  Trainiere neues Basismodell $h_t : X \to \mathbb{R} auf \mathcal{E}_t$
end
return $h : X \to \mathbb{R}$ mit $h(x) = h(h_1(x), .., h_k(x))$

Algorithm 3: VLDS-*Ada*^{2}*Boost* [45]

4.8.3 VLDS-*Ada*^{2}*Boost*

Als nächstes betrachten wir nun den VLDS(Very Large Data Set)-*Ada*^{2}*Boost* Algorithmus. Dieser ist eine Variation des *AdaBoost*-algorithm 2 aus dem Boosting Kapitel. Im Kontext von Big Data stellt sich nun nämlich eine völlig neue Frage. Bisher war unser Datensatz kostbar und wir haben versucht, ihn möglichst effizient zu nutzen, doch was tun wir, wenn das Gegenteil auftritt? Wie gehen wir vor, wenn unser Datensatz so groß ist, dass es unmöglich ist, alle Daten zum Lernen zu verwenden? Natürlich könnte man einfach nur einen Teil der Daten zum Lernen nutzen und die restlichen Daten ignorieren, der VLDS-*Ada*^{2}*Boost* Algorithmus zeigt allerdings eine Möglichkeit, doch noch einen Vorteil aus der großen Datenmenge zu ziehen. Betrachten wir zunächst den Pseudocode des Algorithmus aus der Diplomarbeit von Marius Helf [45]. Hierbei ist zu beachten, dass der in dem Paper behandelte Algorithmus, der *Ada*^{2}*Boost* Algorithmus, eine Variante des normalen AdaBoost-Algorithmus ist. Für den VLDS Part des Algorithmus ist dies aber nicht weiter relevant.

Die Idee dieser Version des Algorithmus ist es, alle R Durchläufe einmal den kompletten Satz an Trainingsdaten auszutauschen. Die neuen Trainingsdaten durchlaufen dann noch

einmal dieselben Schritte wie die alten, danach fährt der Algorithmus fort.

Der Else-Pfad des Algorithmus entspricht deshalb dem normalen Ada^2Boost-Algorithmus. Ein schwacher Klassifikator wird trainiert, danach werden die Datenpunkte neu gewichtet, sodass ein größerer Fokus auf schwierige Fälle gelegt werden kann. Die späteren Klassifikatoren konzentrieren sich dann häufig auf ebendiese. Am Ende wird eine gewichtete Kombination der einzelnen Lerner zum Bilden von Modellen genutzt.

Der Unterschied zum ursprünglichen Algorithmus liegt im if-Teil. Hier wird alle R Durchläufe einmal der Datensatz durch einen völlig neuen, zufälligen Datensatz aus unserer großen Datenmenge ersetzt. Die neuen Daten werden zunächst wieder mit 1 gewichtet, dann werden alle bisher verwendeten Klassifikatoren $1, \ldots, t$ noch einmal durchlaufen, um nacheinander die Daten neu zu gewichten. Die Klassifikatoren werden also auf die neuen Daten angewendet, auf denen sie allerdings nicht trainiert wurden. Wichtig ist, dass die bereits gelernten Klassifikatoren dabei nicht mehr geändert werden, nur die Gewichte der Beispiele werden bearbeitet und für den $t + 1$-ten Klassifikator angepasst.

Der VLDS-Ada^2Boost Algorithmus tauscht also regelmäßig die ihm zugrunde liegenden Daten aus und kann dadurch einen beliebig großen Teil der vorhandenen Daten zum Lernen verwenden. Wichtig ist, dass schon gelernte Klassifikatoren dabei immer wieder auf neuen Daten angewendet werden, der Algorithmus ist also nicht äquivalent zum normalen AdaBoost auf der kombinierten Datenmenge. Stattdessen testet er seine bereits gelernten Klassifikatoren immer wieder auf neuen Daten. Somit können eventuelle Tendenzen in einzelnen Datenblöcken durch Umgewichtung in späteren Klassifikatoren korrigiert werden und es gibt deutlich weniger *Overfitting*. Auch ist es aus praktischen Gründen natürlich hilfreich, dass nicht der gesamte Datensatz dauerhaft im Speicher vorhanden sein muss.

4.8.4 Active Learning

Zum Schluss beschäftigen wir uns noch mit der Idee des *active learnings*. Bisher war es immer unsere Aufgabe, mit einer begrenzten Menge an klassifizierten Daten einen Klassifikator zu trainieren. Nun stellt sich jedoch die Frage, ob dies überhaupt realistisch ist. Woher kriegen wir überhaupt diese perfekt klassifizierten Daten, auf denen wir lernen? Gerade im Kontext von Big Data erhalten wir stattdessen häufig riesige Mengen an Daten, die (noch) nicht klassifiziert sind. Wollen wir diese Daten nutzen, müssen wir sie also erst selber klassifizieren. Aber war unser Ziel nicht gerade, mit den Daten einen Klassifikator zu finden? Was nun?

Überlegungen

Häufig gibt es auch andere Möglichkeiten, die Klasse eines Datenpunkts zu erfahren. So können zum Beispiel im Fall von Diagnosen weitere Tests an einem Patienten durchgeführt

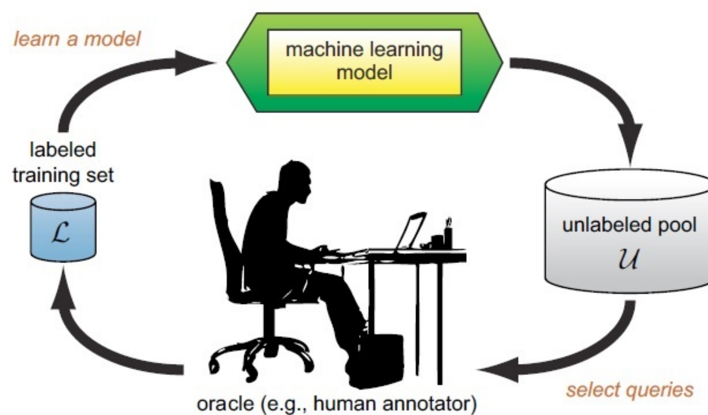


Abbildung 4.11: Active learning als Kreislauf [81]

werden, es kann ein Experte gefragt werden oder Ähnliches. Das Problem hierbei ist nur, dass dies häufig teuer und zeitaufwändig ist. Wollen wir eine sehr große Menge Daten klassifizieren, können wir nicht erwarten, dass unser Experte die Zeit hat (oder wir das Geld haben), jeden Datenpunkt einzeln zu klassifizieren. Genau deshalb soll ja ein automatischer Klassifikator gefunden werden. Es stellt sich nun die Frage, wie wir aus einer begrenzten Anzahl an Beispielen, die wir dem Experten zeigen können, möglichst viele Informationen für unseren Klassifikator erhalten können.

Querys und Experten

Genau mit dieser Frage, welche Daten lasse ich klassifizieren, um daraus zu lernen, beschäftigt sich active learning. Hierbei werden sogenannte *Querys* formuliert, die einem *Oracle*, also dem Experten, übergeben werden. Dabei verfolgen wir einen gierigen Ansatz, wir fragen uns also stets nur welche Anfrage uns genau in diesem nächsten Schritt den größten Informationsgewinn liefert.

Gehen wir davon aus, dass wir zu einem beliebigen Zeitpunkt t bereits Querys gesendet haben, haben wir dadurch auch eine Menge $\mathcal{L}$ von klassifizierten Daten. Jetzt wählen wir entweder einen einzelnen Datenpunkt oder eine Gruppe von Punkten, die wir als nächstes übergeben. Dazu brauchen wir eine Funktion, die den nützlichsten Datenpunkt, gegeben irgendwelcher Kriterien und der Menge $\mathcal{L}$, aussucht. Diesen Punkt lassen wir dann klassifizieren und fügen ihn in $\mathcal{L}$ ein.

Die eigentliche Aufgabe beim active learning ist also, eine ideale Strategie für die Auswahlfunktion zu finden. Hierzu werden häufig die zwei folgenden Kriterien betrachtet, andere sind natürlich auch denkbar.

Informativeness Wie sehr hilft der Punkt bei der Verbesserung meines Modells?

Representativeness Wie repräsentativ ist der Punkt für die Verteilung D, die ich suche?

Uncertainty Sampling

Eine Beispiel für eine sehr einfache Art, eine Query zu formulieren, ist das sogenannte *uncertainty sampling*. Hier wird immer der Datenpunkt zur Klassifikation gewählt, der für das Modell mit der bisherigen Punktemenge $\mathcal{L}$ am schwersten vorherzusagen ist. Beim Formulieren der Query wird also nur auf die Informativeness geachtet. Leider führt dieses Vorgehen wieder zu dem bekannten Overfitting-Problem, da wir unsere Klassifikatoren nur mit Ausreißern und Spezialfällen trainieren. Sie lernen also nur die Besonderheiten des aktuellen Datensatzes auswendig, lernen dabei aber wenig über die repräsentativeren Punkte. Dieses kurze Beispiel reicht aber, um zu zeigen, dass das Formulieren von Querys nicht trivial ist und dass solche einfachen Ansätze keine akzeptable Lösung sind.

Fazit

Wichtig ist, dass active Learning kein Gegensatz zu anderen Sampling-Strategien ist. Stattdessen beschäftigt es sich mit neueren Problemen, die durch die immer größere Menge an gewonnenen Daten auftreten. Active learning kann auch als eine Art Vorbereitung für das eigentliche Sampeln betrachtet werden. Hier erstellen wir aus den noch nicht klassifizierten Rohdaten einen Datensatz, auf den andere Sampling-Methoden wie die Kreuzvalidierung angewandt werden können.

Teil II

Architektur und Umsetzung

Kapitel 5

Komponenten und Architektur

Bei Betrachtung der für den Anwendungsfall zu analysierenden Daten (siehe auch chapter 6) wird deutlich, dass zur Umsetzung der Analyseziele ein Big-Data-System benötigt wird. Mehrere Eigenschaften von Big Data (vgl. chapter 2) treffen auf die Problemstellung zu:

- **Volume.** Die Menge der Daten überschreitet mit teilweise hunderten Gigabyte pro Tag das, was von herkömmlichen Systemen gestemmt werden kann.
- **Velocity.** Das FACT-Teleskop zeichnet kontinuierlich Daten auf und diese sollen idealerweise in Echtzeit verarbeitet werden.
- **Variety.** Wie in chapter 6 gesehen, werden von verschiedensten Sensoren Daten gesammelt, die anschließend in der Analyse kombiniert werden müssen.

Unser System basiert daher auf der in chapter 3 vorgestellten Lambda-Architektur für Big-Data-Systeme. Eine Übersicht über die verwendeten Software-Komponenten ist in Figure 5.1 dargestellt.

Den Kern des Systems bildet ein Apache Hadoop Cluster (vgl. subsection 3.1.1). Dieser bietet zum einen das verteilte Dateisystem HDFS, mit dem große Datenmengen redundant und effizient abrufbar gespeichert werden können. Aufgrund dieser Eigenschaften wird es von uns zur Ablage der Rohdaten, also der in chapter 6 beschriebenen FITS-Dateien, verwendet. Um diese Daten und etwaige Zwischenergebnisse allerdings durchsuchbar zu machen, müssen sie indexiert werden. Hierfür verwenden wir das dokumentenbasierte Datenbanksystem MongoDB. Unsere Lösung sowie andere von uns in Betracht gezogene Systeme werden in chapter 7 vorgestellt.

Zum Anderen bildet Hadoop auch die Grundlage für das verteilte Rechnen auf dem Cluster, da es über den Ressourcen-Manager YARN die Möglichkeit bietet, verschiedenartige verteilte Rechenaufgaben auf dem Cluster auszuführen. Hierzu verwenden wir das Cluster Computing Framework Apache Spark, welches es erlaubt, verteilte Datensätze über den Hadoop Cluster zu verarbeiten (vgl. subsection 3.1.2).

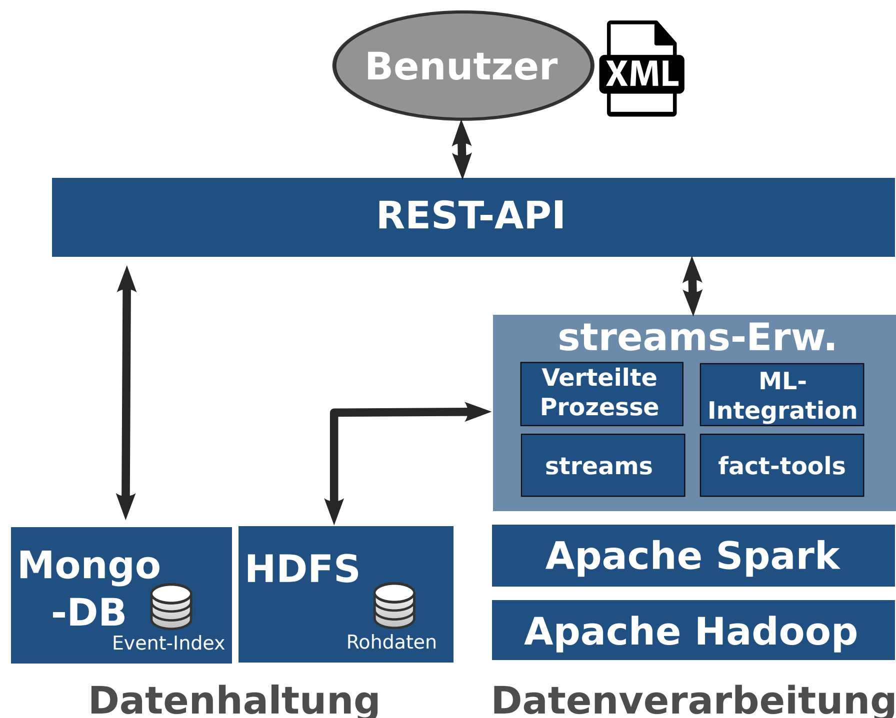


Abbildung 5.1: Überblick über die verwendeten Software-Komponenten

Um die verteilte Ausführung möglichst vieler Analyseaufgaben zu ermöglichen, erweitern wir das `streams`-Framework (vgl. subsection 3.2.4) zur Ausführung unter Apache Spark. Dieser Ansatz hat den Vorteil, dass die von `streams` vorgesehene XML-Schnittstelle zur Spezifikation von beliebigen Analyseprozessen auch für die verteilte Ausführung verwendet werden kann. Insbesondere kann die Analysekette zur Vorverarbeitung der Teleskopdaten (siehe subsection 1.4.1) mit geringen Anpassungen auf dem Cluster ausgeführt werden. Um das zu erreichen, führt unsere Erweiterung die Möglichkeit ein, Prozesse als verteilt zu definieren, sodass diese dann verteilt auf dem Cluster ausgeführt werden.

Die Ausführung kann hierbei auf zwei Arten geschehen. Eine Möglichkeit ist die Verarbeitung eines statischen Datensatzes, der bereits zu Beginn der Laufzeit vollständig vorliegt (zum Beispiel im HDFS). In diesem Fall sprechen wir von einem Batch-Prozess, weil die Daten in einem Schub verarbeitet werden. Die Implementierung dieser Funktion wird in section 9.5 beschrieben. Die andere Möglichkeit ist die Verarbeitung eines Datenstroms. Hierbei werden die Daten erst während der Laufzeit des Prozesses gelesen und fortlaufend verarbeitet. Unsere Umsetzung dieser Funktionalität basiert auf Spark Streaming (vgl. subsection 3.2.3) und wird in section 9.6 beschrieben. Diese beiden Komponenten stellen unsere Umsetzung von Batch- und Speed-Layer dar.

Zusätzlich integriert unsere Erweiterung die von Spark zur Verfügung gestellte Bibliothek für maschinelles Lernen in das `streams`-Framework. Damit lassen sich Lern- und Klassifikationsaufgaben via XML definieren, sodass auch die ML-basierte Analyse der Te-

leskopdaten (vgl. subsection 1.4.1) über dieselbe Schnittstelle spezifiziert werden kann. Näheres zur Implementierung und zu den Änderungen an der XML-Schnittstelle wird in chapter 10 erläutert.

Um dem Benutzer eine einheitliche Schnittstelle zu unserem System zu bieten, verwenden wir eine REST-API (vgl. subsection 3.3.2). Diese erlaubt es einerseits, Anfragen an die MongoDB zu stellen, um zum Beispiel bestimmte Datenpunkte zu selektieren. Andererseits bietet sie ein Webinterface, über das Jobs an den Cluster geschickt und ihre Ausführung überwacht werden kann. Weiteres zu Design und Implementierung der API wird in chapter 8 beschrieben.

Kapitel 6

Datenbeschreibung

In diesem Kapitel werden die verwendeten Daten näher beschrieben. Dazu zählt sowohl eine Einführung in das zugrundeliegende Dateiformat als auch eine etwas ausführlichere Beschreibung der logischen Struktur der Dateien und deren Inhalt.

6.1 FITS-Dateiformat

Das FITS-Format [33] wurde 1981 von der National Aeronautics and Space Administration (NASA) als Austausch- und Transportformat von astronomischen Bilddaten entwickelt. Dabei ist dieses Format modular aufgebaut und es gibt verschiedene *Extensions*, welche die eigentliche Datenrepräsentation in der Datei vorschreiben.

Eine FITS-Datei hat zunächst einen 2880 Byte großen Header-Block, den sogenannten Primary-Header, wobei dieser die weiteren Daten in der Datei beschreibt. Dazu besteht der Header aus Key-Value-Paaren, denen ein optionaler Kommentar folgen kann. Pro Key-Value-Paar stehen jedoch nur 80 Byte zur Verfügung, von denen zehn dem Schlüssel zugeteilt sind und 70 Byte sich der Wert und der Kommentar teilen. Sollte der Header nicht die kompletten 2880 Byte brauchen, so bleiben die restlichen Bytes leer. Im Primary-Header sind bestimmte Felder vorgeschrieben, zum Beispiel eine Checksumme über den Header und ob sich an den FITS-Standard gehalten wird oder nicht. Dieser Header gibt auch Auskunft darüber, ob Extensions in der Datei verwendet werden.

Nach dem Primary-Header folgt das erste Datenfeld, welches auch leer sein kann.

Hiernach folgt der Secondary-Header, der ähnlich zum Primary-Header aufgebaut ist, jedoch auch angibt, welche *Extension* verwendet wird und noch weitere Informationen für diese enthält. Als Beispiel für eine solche Erweiterung sei hier die *Extension* „BINTABLE“ erwähnt. Dafür wird im Secondary-Header auch angegeben, wie viele Zeilen diese Tabelle enthält, wie viele Spalten es gibt, wie diese Spalten heißen und welchen Datentyp sie haben. Dieser Header wird auch in 2880 Byte großen Blocks gespeichert.

Nach diesen Header-Blocks folgt dann die Datentabelle.

Darüberhinaus werden große FITS-Dateien mit GZip komprimiert und diese Dateien tragen die Endung *.fits.gz*.

6.2 Rohdaten

Die Daten des FACT werden in FITS-Dateien mit der Erweiterung „BINTABLE“ gespeichert. Dazu schreibt das Teleskop die auftretenden Events in einer Zeitspanne von etwa fünf Minuten in sogenannte *Runs*. Diese Dateien werden in einer hierarchischen Ordner-Struktur pro Nacht zusammengefasst, zum Beispiel „raw/2013/09/29/0130929_232.fits.gz“ für den *Run* mit der Nummer „232“ am 29.09.2013. Innerhalb eines *Runs* gibt es nun eine Tabelle mit etwa 3000 Zeilen, wobei jede Zeile ein Event beschreibt. Dazu zählen unter anderem die Eventnummer, der Zeitpunkt des Auftretens und die Daten der einzelnen Pixel, ein Datenfeld aus 432000 16bit-Integern.

6.3 Monte-Carlo-Daten

Monte-Carlo-Daten werden im Gegensatz zu den anderen Daten per Simulation erzeugt. Bei dieser Simulation trifft ein Teilchen von festgelegter Energie auf die Atmosphäre und erzeugt ein Cherenkov-Licht, das von einem simulierten Teleskop aufgenommen wird.

Der große Vorteil dieses Vorgehens liegt darin, dass im resultierenden Datensatz sowohl die Features der Aufnahme als auch die Energie des verursachenden Teilchens vorliegen. Deswegen werden die Monte-Carlo-Datensätze dazu verwendet, Modelle zu trainieren, die anhand der Features die Energie des zugrundeliegenden Teilchens vorhersagen.

6.4 Drs-Daten

Die analogen Signale, die an den Fotodioden der Teleskopkamera gemessen werden können, werden mithilfe von Domino-Ring-Samplern (DRS) digitalisiert. Ohne Kalibierung sind die Messungen jedoch, wie in Figure 6.1 (links) zu sehen, stark verrauscht. Dies liegt zum Einen am einfallenden Hintergrundlicht und zum Anderen an temperaturbedingen Spannungsänderungen. Um Events besser erkennen zu können, wird eine DRS-Kalibrierung durchgeführt. Diese wird in regelmäßigen Zeitabständen vor einem Run durchgeführt und dessen Ergebnisse mit den folgenden Aufnahmen verrechnet.

Die Drs-Daten, die ebenfalls im FITS-Format abgespeichert werden, beinhalten neben diversen Kalibrierungskonstanten zwei Aufnahmen: Ein Bild wird bei geschlossener Klappe

aufgenommen und eins wird vom Nachthimmel gemacht. Aus den Informationen dieser Aufnahmen kann das Hintergrundrauschen für folgende Aufnahmen zuverlässig herausgerechnet werden (s. Figure 6.1 (rechts)).

[5], [3], [4]

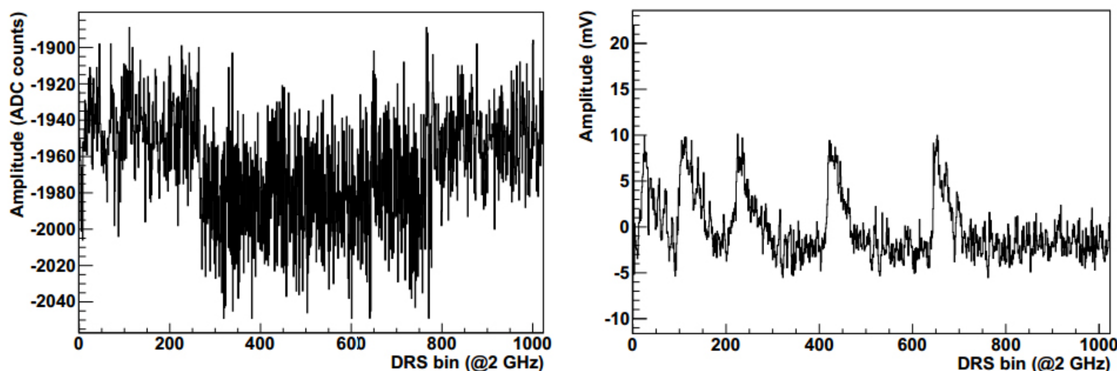


Abbildung 6.1: Event vor (links) und nach (rechts) der DRS Kalibrierung. Die Spitzen entsprechen den Signalen einer einzelnen Fotodiode [5]

6.5 Aux-Daten

Neben den eigentlichen Rohdaten werden von verschiedenen weiteren Sensoren Daten aufgenommen, die dabei helfen sollen, die Rohdaten besser zu interpretieren oder Anpassungen an dem Messvorgang zur Laufzeit durchzuführen. Diese Hilfsdaten (Auxiliary Data) werden je nach Sensor in bestimmten Intervallen im FITS-Format abgespeichert und beinhalten zum Beispiel Informationen über Wetter- und Sichtverhältnisse zum Zeitpunkt einer Aufnahmereihe. So können etwa Informationen über die Wolkendichte oder Nebel von Interesse sein, da bei dichtem Himmel, schlechten Sichtverhältnissen oder Schneefall nur ein Bruchteil des Cherenkov-Lichts am Teleskop ankommt. Weiterhin kann beispielsweise Regen einen Wasserfilm auf der Kamera hinterlassen, der eingehendes Licht reflektiert, und starker Wind kann die Lage des Teleskops verändern, sodass Anpassungen an dessen Antriebssystem gemacht werden können [69].

Für den Anwendungsfall sind die Aux-Daten insofern interessant, als dass man durch deren Indexierung in einer Datenbank eine genauere Eventselektion und Eventanalyse erreichen kann. So können zum Beispiel Anfragen der Art „Finde alle Events aus Nacht n, wo die Temperatur unter y°C liegt“ gestellt werden, um bessere Modelle für maschinelle Lernverfahren zu erzeugen. Bei Anfragen dieser Art werden geeignete Strategien benötigt, um Event-Daten und Aux-Daten zusammenzuführen, da nicht sichergestellt werden kann, dass zum Zeitpunkt t_e der Aufnahme eines Events e auch Sensordaten aufgezeichnet wurden.

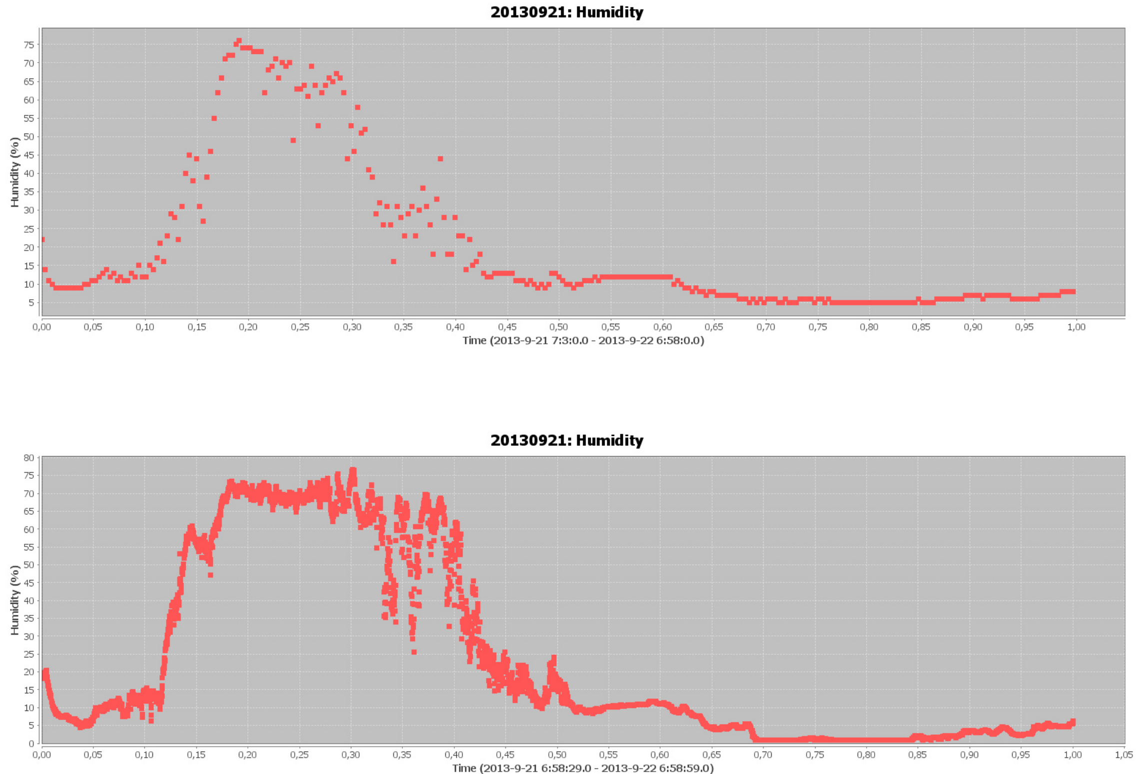


Abbildung 6.2: Statistik zur Luftfeuchtigkeit in der Nacht des 21.09.2013 aufgenommen von zwei Sensoren: TNG (oben) und MAGIC (unten)

Meistens befindet sich t_e nämlich irgendwo zwischen zwei aufgezeichneten *AuxPoints* a_i und a_j, also $t_{a_i} < t_e < t_{a_j}$. In solchen Fällen wird e mit dem AuxPoint zusammengeführt, dessen Aufnahme am nächsten an t_e liegt, um möglichst genaue Informationen zu erhalten.

Für Analysezwecke wurde von uns ein Tool *(AuxViewer)* entwickelt, mit dessen Hilfe sich Diagramme indizierter Aux-Daten für eine bestimmte Nacht generieren lassen. Eine beispielhafte Analyse der Wetterdaten ergab, dass verschiedene Sensoren unterschiedliche Aufnahmeintervalle haben, wie die Statistiken zur gemessenen Luftfeuchtigkeit einer Nacht in Figure 6.2 zeigt. Für eine genauere Eventselektion gilt es also herauszufinden, welche Sensordaten besser geeignet sind, *falls* verschiedene Sensoren das selbe Merkmal aufzeichnen.

Eine stichprobenartige Überprüfungen mehrerer Sensoren zu unterschiedlichen Nächten zeigte weiterhin, dass die Sensoren anscheinend zuverlässig arbeiten. Die Werte werden in regelmäßigen Abständen ausgelesen, Definitionslücken durch Ausfälle wurden nicht verzeichnet und Sensoren, die dasselbe Merkmal aufnehmen, liefern in etwa die selben Werte (siehe z.B. Figure 6.2).

Kapitel 7

Indexierung der Rohdaten

Der Ausgangspunkt für unsere Datenanalyse sind die vielen Hundert Gigabyte von Rohdaten, die im FITS-Format vorliegen und von uns in dem verteilten Dateisystem HDFS abgelegt wurden (vgl. chapter 6). Unser System soll dem Nutzer erlauben, anhand von Suchanfragen bestimmte Teildatensätze daraus auszuwählen, um diese dann weiterzuverarbeiten. Diese Anfragen beziehen sich nicht auf die vom Teleskop gemachten Bilder selbst, sondern auf die Metadaten zu diesen Bildern, also etwa den Zeitpunkt der Aufnahme, die Ausrichtung des Teleskops oder die Außentemperatur.

Eine effiziente Bearbeitung solcher Anfragen ist nur dann möglich, wenn diese Daten in einer für die Suche geeigneten Datenstruktur vorliegen. Andernfalls müsste für jede Anfrage der gesamte Datensatz durchlaufen werden. Aus diesem Grund indexieren wir die Metadaten mithilfe von Datenbanksystemen. Ausgenommen sind hierbei die eigentlichen Bilddaten, welche einen Großteil der Datenmenge ausmachen, jedoch für die Auswertung der Suchanfragen nicht relevant sind. Zweck der Datenbanken ist es, die Menge der aufgezeichneten Datenpunkte (Events) zu finden, die den durch den Nutzer formulierten Bedingungen genügen. Anschließend können dann gezielt die zugehörigen Bilddaten aus dem HDFS geladen und weiterverarbeitet werden.

Die drei von uns untersuchten Systeme sind die dokumentenbasierte verteilte Datenbank MongoDB, die verteilte Suchmaschine ElasticSearch und die relationale Datenbank PostgreSQL. Die Art und Weise, wie wir jedes dieser Systeme auf das Problem angewendet haben, wird im Folgenden erläutert.

7.1 MongoDB

Das Ziel, einen Index für die Rohdaten zu erstellen, kann in MongoDB (siehe subsection 3.3.1) auf sehr unterschiedliche Art und Weise erreicht werden. Eine mögliche Realisierung besteht in dem Anlegen einer Collection, die für jedes Event ein einzelnes Dokument

besitzt. Genauso gut ist es möglich, mehrere Events zu aggregieren und als ein Dokument zu speichern. Wir gehen im Folgenden auf beide Varianten ein.

Ein Dokument pro Event. Dieser Ansatz ist sehr naheliegend und nutzt die simple key-value-Struktur der JSON-Dokumente. Ein großer Vorteil liegt in dem einfachen Hinzufügen von zusätzlichen Attributen, wenn weitere Informationen zu den Events gespeichert werden sollen. Diese flache Dokumentenstruktur führt auch zu sehr übersichtlichen Suchanfragen, da eine Suchanfrage bei MongoDB ebenfalls ein JSON-Objekt ist, das dieselbe Struktur wie das Dokument besitzt.

Aggregation von mehreren Events. Ein MongoDB-Dokument darf Arrays, eingebettete Dokumente sowie Arrays von eingebetteten Dokumenten beinhalten. Daher ist es möglich, mehrere Events in einem Dokument zusammenzufassen. Dabei kann die Granularität frei gewählt werden. So können zum Beispiel für jede Sekunde alle Events, die in dieser Sekunde aufgenommen wurden, zu einem Dokument zusammengefasst werden. Durch Aggregation sinkt die Anzahl der Dokumente in der Collection, wodurch die Größe der Indices sinkt. Außerdem liegen dann die Events, die in der gleichen Sekunde aufgenommen wurden, in der gleichen Datei. Wenn also oft Events aus einem zusammenhängenden Zeitraum angefragt werden, sinkt die Anzahl der zu durchsuchenden Dokumente, was die Performanz vermutlich erhöht. Dafür steigt aber auch die Komplexität der Suchanfragen.

Beide Varianten der Indexierung wurden von uns mithilfe des `streams`-Frameworks implementiert. Bei den bisher durchgeführten Tests wurde die MongoDB bisher nur auf einem einzelnen Knoten gestartet, weshalb noch keine abschließende Beurteilung möglich ist. Es hat sich insbesondere bei der Variante „Ein Dokument pro Event“ gezeigt, dass der Job mehr Zeit in Anspruch nimmt, als es für das reine Auslesen der Ursprungsdateien nötig wäre. Dieses Problem könnte durch ein verteiltes Setup der Datenbank gelöst werden.

Darüber hinaus ist es uns gelungen, die Aux-Daten in die indexierten Meta-Daten zu integrieren. Dabei wurde die in section 6.5 erläuterte Strategie zum Finden des passenden Messwertes für ein Event eingesetzt.

7.2 Elasticsearch

Um die Performanz verschiedener Datenbanken hinsichtlich des Anwendungsfalles dieser Projektgruppe gegeneinander abwägen zu können, wurde als zweite Persistenzlösung Elasticsearch eingesetzt. Der Cluster *pg594-cluster* gliedert sich in drei Indizes, nämlich *metadataindex*, *drsindex* und *auxindex*. Der *metadataindex* enthält Dokumente des Typs *metadata*, in denen die Metadaten zu den jeweiligen Events abgelegt sind. Im *drsindex* befinden sich die Kalibrationsdaten aus den DRS-Dateien und im *auxindex* in analoger

Weise die in den AUX-Dateien befindlichen Informationen. Für den *pg594-cluster* wurde Elasticsearch lediglich auf einem einzigen Rechenknoten betrieben, eine Alternative dazu wäre jedoch gewesen, auf jedem verfügbaren Knoten des Clusters des Sonderforschungsbereiches 876 einen Elasticsearch-Node zu betreiben.

7.3 PostgreSQL

Als dritte mögliche Lösung haben wir ein PostgreSQL System aufgesetzt, also ein herkömmliches relationales Datenbankmanagementsystem (vgl. Figure 3.3.1). Dies ist unter anderem dadurch motiviert, dass die Größe der Metadaten sich in Grenzen hält. Es ist anzunehmen, dass der verbrauchte Speicherplatz pro Event selbst mit zusätzlichen Aux-Daten und berechneten Features 2 KB nicht überschreiten wird. Für die zwei Millionen Events, die aktuell den Cluster füllen, sind das gerade einmal 4 GB. Insofern ist es durchaus realistisch, die Metadaten auch auf lange Sicht in einer monolithischen relationalen Datenbank zu verwalten. Des Weiteren bietet Postgres-XL im Zweifelsfall die Möglichkeit, auf eine verteilte Lösung umzusteigen.

Eine größere Herausforderung stellt das Design eines Schemas dar, das alle in Zukunft benötigten Funktionalitäten bereitstellt. Insbesondere das Abspeichern der berechneten Features ist nicht einfach, da jederzeit neuartige Features hinzukommen können. Eine Möglichkeit, dies umzusetzen, ist, eine eins-zu-viele-Relation zu verwenden, die Events und Features verbindet. Diese würde allerdings dazu führen, dass für viele Anfragen teure Join-Operationen nötig wären, und so die Prinzipien der dimensionalen Modellierung verletzen (vgl. Figure 3.3.1). Eine andere Möglichkeit ist der Einsatz des JSON-Datentyps, den PostgreSQL anbietet. Neue Features könnten dann einfach in bestehende Tabellenzeilen eingefügt werden.

7.4 Auswahl der Datenbank

Nachdem im ersten Semester der Projektgruppe drei verschiedene Datenbanken in einer experimentellen Phase parallel zueinander verwendet wurden, entwickelte sich im zweiten Semester ein deutlicher Trend zur Nutzung der MongoDB. Dies ist zum Einen damit zu begründen, dass eine dokumentenbasierte Lösung im Gegensatz zu einem herkömmlichen relationalen Datenbankmodell für unseren Anwendungsfall zweckdienlicher ist, da Erstere es ermöglicht, jederzeit ohne großen Aufwand neue Attribute hinzuzufügen. Ein solches Vorgehen ist beispielsweise dann notwendig und relevant, wenn Berechnungen auf den Events durchgeführt werden, deren Ergebnisse längerfristig Gültigkeit haben bzw. häufig benötigt werden und somit nicht bei jedem Zugriff neu kalkuliert werden sollen. Zum

Anderen setzen auch die Physiker zur Indexierung ihrer Rohdaten eine MongoDB ein, sodass durch die Fokussierung auf diese Datenhaltungslösung ein leichter Umstieg auf die Datenbank der Physiker oder aber eine Fusionierung der Datenbanken ermöglicht wird.

Kapitel 8

RESTful API

Zum Erreichen der Analyseziele, namentlich die Normalisierung der Rohdaten, die Durchsuchbarkeit von Events und deren Analyse mittels maschineller Lernverfahren, ist eine RESTful API implementiert worden. Insbesondere soll mit ihr sowohl der Zugriff auf den Rohdaten-Index, als auch die Ausführung, die Überwachung und die Steuerung von verteilten Jobs über ein einheitliches HTTP Interface vereinfacht werden.

Für die angesprochenen Punkte sind Schnittstellen entworfen und implementiert worden, die im Laufe der folgenden Abschnitte detaillierter besprochen werden. Neben der Dokumentation der einzelnen Schnittstellen wird gleichzeitig auf die technischen Aspekte der Implementierung eingegangen. Weiterhin wird eine Webanwendung präsentiert, die zusammen mit der RESTful API ausgeliefert wird und deren Funktionalitäten über eine komfortable Oberfläche zur Verfügung stellt.

8.1 Design

Zur Umsetzung der RESTful API (vgl. subsection 3.3.2) ist es zunächst wichtig, diese Schnittstelle zu planen. Dazu werden wir die notwendigen URLs festlegen und das Format der Daten definieren. Weiterhin wird beschrieben, wie diese Informationen auch außerhalb dieses Berichts dokumentiert wurden. [66]

8.1.1 Endpunkte

Die Endpunkte der REST API sind so gewählt, dass der Zugriff auf indizierte Daten in den in der PG genutzten Datenbank MongoDB mehr anwendungsfallbezogen verläuft.

Die in Table 8.1 aufgelisteten Schnittstellen sind für den Zugriff auf Metadaten von Events konzipiert. Die Angabe der GET-Parameter ist optional. Hierbei kann über *format* das

URL	GET-Parameter
GET /api/events	format, filter
GET /api/events/count	filter

Tabelle 8.1: Schnittstellen der REST API für Metadaten

```
1  [
2    {
3      "EVENT_NUM": "4",
4      "TRIGGER_NUM": "4",
5      "NIGHT": "20130921",
6      ...
7    },
8    {
9      "EVENT_NUM": "5",
10     "TRIGGER_NUM": "4",
11     "NIGHT": "20130921",
12     ...
13   },
14   ...
15 ]
```

(a) JSON

```
1  [
2    {
3      "path": ".../hdfs/fact/raw/2013/08/21/....fits.gz",
4      "eventNums": [20, 22, 24, 50, ...]"
5    },
6    {
7      "path": ".../hdfs/fact/raw/2013/09/06/....fits.gz",
8      "eventNums": [2, 22, 120, 121, ...]"
9    },
10   ...
11 ]
```

(b) Minimal

Abbildung 8.1: Die Rückgabeformate der REST API

Rückgabeformat einer Antwort bestimmt werden (s. subsection 8.1.2). Über den Parameter *filter* lässt sich ein Filterausdruck übergeben, mit dem die Metadaten selektiert werden können (s. subsection 8.2.2).

Im Laufe des zweiten Semesters sind weitere Endpunkte erstellt worden, mit denen sich etwa komfortabel Jobs verwalten und ausführen lassen. Diese sollen hier jedoch nicht alle aufgezählt werden, da sie im Verlauf des Kapitels, in chapter 18 und in einer separaten Software-Dokumentation erläutert sind.

8.1.2 Rückgabeformate

Die Ausgabe von Anfragen, die über die REST API gestellt werden, können für verschiedene Zwecke anders formatiert werden. Das Rückgabeformat lässt sich dabei mit dem GET-Parameter *format* über die URL festlegen. Mögliche Werte für diesen Parameter sind *json* und *min*. Falls der Formatierungsparameter nicht übergeben wird, wird der Wert standardmäßig auf *json* gesetzt. Das Rückgabeformat ermöglicht so ein einheitliches Format, sodass Anfragen unabhängig von der angesprochenen Datenbank eine einheitliche Antwort erzeugen. Eine Beschreibung der unterschiedlichen Formate sowie mögliche Beispiele zur Benutzung und mögliche Ausgaben ist im Folgenden gegeben.

JSON Eine Anfrage, die den Parameter *format=json* übergibt, bekommt als Antwort eine Liste aller Events mit allen Attributen, wie sie in der Datenbank vorkommen, im JSON-Format. Dadurch wird ein direkter Zugriff auf die indexierten Metadaten ermöglicht. Eine beispielhafter Request an die API könnte wie folgt aussehen:

GET http://[...]/api/events/?filter=[...]&**format=json**

Die Antwort würde in diesem Fall aussehen, wie in Figure 8.1a gezeigt, wobei die Felder „EVENT_NUM“, „TRIGGER_NUM“ und „NIGHT“ den Namen der entsprechenden Dokumenten in der Datenbank entsprechen.

Minimal Anstatt alle Felder der Metadaten zurückzugeben, besteht der Sinn dieses Parameters darin, an die eigentlichen Rohdaten zu kommen, die zu dem in der URL gegebenen Filterausdruck passen. Wie in Figure 8.1b zu sehen, wird die Antwort ebenfalls im JSON-Format zurückgegeben. Zu jedem Event, das auf den Filter zutrifft, wird die Event-Nummer innerhalb der entsprechenden FITS-Datei in eine Liste eingefügt. Ein HTTP Request sollte nach folgendem Muster gestellt werden:

GET http://[...]/api/events/?filter=[...]&**format=min**

Dieser Parameter eignet sich insbesondere für den Fall, zu einer gestellten Anfrage die Rohdaten aus den fits Dateien zu erhalten, um diese anschließend in einem Stream zu verarbeiten. Durch der Angabe der einzelnen Event-Nummern kann im Stream innerhalb einer fits-Datei genau nach passenden Events gesucht werden.

8.1.3 Dokumentation

Da diese API nicht nur von Mitgliedern dieser PG verwendet werden soll, ist eine gute Dokumentation unerlässlich. Natürlich erfüllt dieser Bericht auch diese Funktion, jedoch wäre es wünschenswert die Dokumentation näher an die Anwendung zu bringen.

Um diese Anforderungen zu erfüllen, wurde sich für das Swagger-Projekt entschieden. Dort wurde eine Spezifikation, die mittlerweile von der Open API Initative betreut wird, entwickelt, mit der sich RESTful APIs mithilfe von JSON beschreiben lassen [74]. Rund um diese Dokumentation sind unterschiedliche Tools entstanden, z.B. der Text-Editor Swagger Editor, mit dem das JSON, welches die API beschreibt, einfach bearbeitet werden kann. Noch hilfreicher ist jedoch die Swagger UI, die aus der JSON-Definition eine dynamische Website generiert, welche die Dokumentation übersichtlich und mit einer modernen Oberfläche anzeigt. Darüber hinaus kann man die angegebenen REST-Endpunkte auch direkt ansprechen und bekommt die Anfrage- und Antwort-Informationen detailliert präsentiert (vgl. Screenshot). Diese Website kann nun mit zusammen mit der eigentlichen API auf einem Server bereitgestellt werden.

8.2 Implementierung

Zur Implementierung der REST API wurde auf Java-basierte Lösungen gesetzt, für die bereits entsprechendes Know-How unter dem Team Mitgliedern vorhanden war. Dies sollte eine schnelle und zuverlässige Implementierung ermöglichen. Im Folgenden werden die verwendeten Softwareprodukte kurz vorgestellt.

8.2.1 Spring Framework

Bei der Implementierung der RESTful API wurde das Spring-Framework verwendet. Dabei handelt es sich um ein sich aus verschiedenen, separat nutzbaren Modulen bestehendes OpenSource-Framework für die Java-Plattform. Für den Einsatz in dieser Projektgruppe wurden aus dem vielfältigen Angebot an Modulen des Spring-Frameworks *Spring Boot* sowie *Spring Data* für MongoDB ausgewählt, welche im Folgenden näher erläutert werden.

Spring Boot Spring Boot ermöglicht es, auf einfache Weise und mit minimalem Konfigurationsaufwand Stand-Alone-Anwendungen zu entwickeln. Bei mit Spring Boot entwickelten Anwendungen entfällt zum Einen jegliche über die pom.xml herausgehende XML-Konfiguration sowie zum Anderen die Notwendigkeit, die Anwendung als War-File zu deployen, da Spring Boot bereits einen Application-Server - wahlweise Tomcat, Jetty oder Undertow - mitliefert, sodass die Anwendung nur noch gestartet werden muss.

Zur Einbindung von Spring Boot müssen lediglich die benötigten Dependencies zur Projektkonfigurationsdatei des entsprechenden Dependency-Management-Systems hinzugefügt werden.

```
1
2  <parent>
3      <groupId>org.springframework.boot</groupId>
4      <artifactId>spring-boot-starter-parent</artifactId>
5      <version>1.3.3.RELEASE</version>
6  </parent>
7  <dependencies>
8      <dependency>
9          <groupId>org.springframework.boot</groupId>
10         <artifactId>spring-boot-starter-web</artifactId>
11     </dependency>
12 </dependencies>
```

Listing 8.1: Einbindung von Spring Boot mittels Maven durch Hinzufügen der Dependencies zur pom.xml

Das Herzstück einer mit Spring Boot entwickelten Anwendung ist die Application-Klasse, die im Falle der REST API folgendermaßen aussieht:

```
1
2 @SpringBootApplication
3 public class Application {
4     public static void main(String[] args) {
5         SpringApplication.run(Application.class, args);
6     }
7 }
```

Listing 8.2: Application-Klasse bei Spring Boot

Die Annotation `@SpringBootApplication` deklariert die Anwendung als Spring Boot Application und ermöglicht den Einsatz folgender weiterer Annotationen:

- Durch die Annotation `@Configuration` wird eine annotierte Klasse als mögliche Quelle für Bean-Definitionen im Application-Context erkannt.
- Die Annotation `@EnableAutoConfiguration` ermöglicht, wie der Name bereits erkennen lässt, eine automatisierte Spring-Konfiguration, im Zuge welcher Beans auf Basis von Classpath-Settings generiert sowie diverse weitere Einstellungen vorgenommen werden. Über das vollständige Funktionsspektrum klärt die Projekt-Homepage von Spring Boot auf.
- Falls von Spring Boot eine entsprechende Dependency in der Projektkonfigurationsdatei des Dependency-Management-Systems erkannt wurde, wird die Anwendung automatisch als Web-Anwendung gekennzeichnet.
- Durch Einsatz der Annotation `@ComponentScan` sucht Spring Boot automatisiert nach weiteren Komponenten, Services sowie Konfigurationsdateien.

Die `main()`-Methode der Application-Klasse nutzt Spring Boots `SpringApplication.run()`-Methode, um die Anwendung zu starten, welche den Application-Context und somit auch alle automatisiert und manuell erstellten Beans zurückgibt.

Spring Data Bei Spring Data handelt es sich um ein Modul des Spring-Frameworks, mittels dessen Boilerplate-Code beim Datenbank-Zugriff durch Nutzung sogenannter CRUD-Repositories reduziert werden kann. Dieses wird nachfolgend in Kapitel 1.5.2.2 näher in Augenschein genommen.

8.2.2 Filterung

Der Ansatz der Implementierung einer Schnittstelle mit Hilfe von REST ressourcenbasiert auf der Überlegung bestimmte Funktionen zu kapseln und als Services bereitzustellen,

die von anderen Teilen der Anwendung oder von außerhalb angesprochen werden können, um z.B. die Metadaten der Events bereitzustellen, die wiederum zur Selektion von Events genutzt werden können, die bestimmten Kriterien genügen. Im Falle der Events handelt es sich bei den Kriterien um eine Vielzahl von Attributen, die jedes Event inne hat.

Herausforderungen Bei der Implementierung der Filterung stellen sich einem mehrere Herausforderungen. Die Filterung muss in der Lage sein, eine Anfragesprache (engl. *domain specific language (DSL)*) verarbeiten und interpretieren zu können, sodass auch komplexere Anfragen an das System gestellt werden können. Es wäre noch verhältnismäßig leicht gewesen, die Selektion von Events zu implementieren, deren Attribut exakt den vorgegebenen Werten entsprechen. Womöglich möchte der Anwender aber den Wertebereich eines Attributs nicht auf einen bestimmten Wert, sondern auf ein Intervall eingrenzen und womöglich sollen einige Datensätze prinzipiell ausgeschlossen werden. Und vielleicht soll ein Wert nicht nur innerhalb eines, sondern zweier Intervalle liegen. Der Komplexität einer Anfrage sind je nach Anwendungsfall also keine Grenzen gesetzt und die Implementierung eines geeigneten Interpreters ein anspruchsvolles Unterfangen gewesen. Es wird also eine Anfragesprache verlangt, die zum Einen hinsichtlich der Ausdruckskraft z.B. der Datenbanksprache SQL nahekommt und zum Anderen vom Anwender leicht anzuwenden und somit möglichst nah an die natürlichen Sprache angelehnt ist.

SQL (engl. *Structured Query Language*) ist eine Anfragesprache, die auf der relationalen Algebra basiert und den Umgang mit den Daten eines relationalen Datenbankmanagementsystems ermöglichen. Eine wichtige Komponente der SQL ist die sog. *Query*, die der Beschreibung der gewünschten Daten dient und vom Datenbanksystem interpretiert wird, um die gewünschten Daten bereitzustellen. Listing 8.3 stellt eine solche SQL-Anfrage beispielhaft dar, die den Pfad (*event_path*) aller Events ausgeben soll, deren Eventnummer (*event_num*) entweder zwischen 5 und 10 oder zwischen 50 und 100 liegt und deren Triggernummer (*trigger_num*) größer als 10 ist. Bei der vorliegenden Anfrage ist die *WHERE clause* von Interesse, da diese beschreibt, welche Eigenschaften die gewünschten Events besitzen sollen, und nach diesen Kriterien gefiltert wird.

```sql
1 SELECT event_path FROM events WHERE (
2   (event_num >= 5 AND event_num <= 10) OR
3   (event_num >= 50 AND event_num <= 100)
4 ) AND trigger_num > 10
```

Listing 8.3: Beispiel für eine SQL-Anfrage

Die Ausführung einer übergebenen SQL-Anfrage wäre möglich, aber bringt mehrere Nachteile mit sich. Die Persistierungsebene wird nicht abstrahiert und der Anwender ist gezwungen mit dieser insofern direkt zu interagieren, als dass er sich unnötigerweise mit

dem Aufbau des Datenbankschemas vertraut machen muss. Wie eingangs erwähnt, werden mehrere Systeme zur Datenhaltung eingesetzt, die nicht allesamt auf SQL als Anfragesprache setzen. MongoDB setzt ganz im Gegenteil auf ein JSON-basiertes Anfrageformat, dessen Pendant zum o.g. SQL-Ausdruck in Listing 8.4 dargestellt wird.

```
{ $and: [
    { $or: [
        { event_num: { $gte: 5, $lte: 10 }},
        { event_num: { $gte: 50, $lte: 100 }}
    ]},
    {
      trigger_num: { $gt: 10 }
    }
]}
```

Listing 8.4: Beispiel für eine Anfrage an eine MongoDB Datenbank

Da die REST API JSON-basiert ist und die Anfragesprache von MongoDB alle benötigten Eigenschaften einer ausdrucksstarken Anfragesprache in Form eines JSON-Dokuments mitbringt, liegt der Gedanke nahe, diese Syntax zur Filterung der Events zu übernehmen. Die Problematik bestünde jedoch darin, diese Anfrage in das jeweilige Anfrageformat der anderen Systeme (Elasticsearch und PostgreSQL) übersetzen zu müssen, was einen gewaltigen Overhead an zusätzlicher Programmierarbeit zur Folge hätte.

Es wird also eine Lösung benötigt, um die Anfrage über den Filter möglichst automatisiert in eine kompatible Anfrage für die jeweilige Engine zu übersetzen.

Architektur Architektonisch besteht die Filterung aus drei Schichten: Schnittstelle, Service-Layer und Persistierungs-Layer. Wie in subsection 8.1.1 erwähnt, steht jeweils ein Endpunkt für jede Engine zur Verfügung, der einen Filterausdruck über die aufgerufene URL entgegennimmt. Jeder Endpunkt bzw. jede Engine, die durch diesen repräsentiert wird, verwendet einen eigenen Service, der die Geschäftslogik für die jeweilige Engine implementiert. Über die Geschäftslogik der Services wird schließlich auf den Persistierungs-Layer zugegriffen, welcher den Zugriff auf die persistierten Daten ermöglicht.

Der Kern des Spring-Frameworks, welches in subsection 8.2.1 eingeführt wurde, kann um das Modul *Spring Data JPA* erweitert werden, welches auf der *Java Persistence API* (JPA) aufbaut und die Zuordnung zwischen Java-Objekten und den persistierten Daten vereinfacht. Man spricht hier auch von einem bidirektionalen Mapping, sodass Veränderungen der Daten auf die korrespondierenden Java-Objekte übertragen und gleichzeitig Änderungen der Attribute der Java-Objekte in den Daten reflektiert werden. Die

grundlegende Idee besteht darin, sog. *Repositories* bereitzustellen, die als Interfaces umgesetzt wurden und über die grundlegende Methoden zur Datenverarbeitung (CRUD - Create, Read, Update, Delete) zur Verfügung gestellt werden. Ebenso wird über die Repositories der Datentyp festgelegt, der für das Mapping zwischen Daten und Objekten genutzt werden soll.

Da JPA mit den verschiedensten Datenbanktreibern kompatibel ist und die Repositories für alle drei Datenbankengines genutzt werden können, wurde der Zugriff auf die Persistierungsebene vereinheitlicht. Diese Vereinheitlichung stellt auch die Grundlage für eine einheitliche Lösung zur Filterung von Eventdaten dar.

Um die Events filtern zu können, wird das Framework *QueryDSL* eingesetzt, das typsichere, SQL-ähnliche Anfragen an unterschiedliche Datenquellen, wie JPA, MongoDB, SQL, Java Collections u.v.m. ermöglicht. Dabei ist das Format der Anfrage unabhängig von der verwendeten Datenquelle und somit die Anwendung des Filters vereinheitlicht.

Implementierung Für jedes Datenbanksystem steht ein dedizierter Service zur Verfügung, der die Businesslogik kapselt. Dabei soll die Filterung der Events unabhängig vom verwendeten System sein bzw. jedes System die Filterung unterstützen. Zu diesem Zweck implementieren alle Services ein Interface, welches die Methode zur Filterung der Events definiert (vgl. Listing 8.5).

```
1 public interface EventService {
2   Iterable<Metadata> filterEvents(String filterExpression);
3 }
```

Listing 8.5: Service Interface

Dem Rückgabewert der Methode `filterEvents(...)` ist ein `Iterable` des Datentyps `Metadata`. `Metadata` ist ein sog. *POJO* (Plain Old Java Object), welches die Metadaten der Events aus der Datenbank als Java-Objekt repräsentiert. Somit ist die Klasse `Metadata` auch diejenige Klasse, die von QueryDSL modifiziert wird, um entsprechende Anfragen an eine Liste mit Instanzen dieser Klasse stellen zu können. Eine Anfrage könnte beispielsweise wie in Listing 8.6 aussehen.

```
1 ((
2   eventNum.gte(5).and(eventNum.lte(10))
3 ).or(
4   eventNum.gte(50).and(eventNum.lte(100))
5 )).and(
6   triggerNum.gt(10)
```

```
7 )
```

Listing 8.6: Anfrage

Hier repräsentieren `eventNum` und `triggerNum` Attribute der Klasse `Metadata`, die aber in dem POJO als Integer definiert sind und somit nicht über die Methoden `gte()`, `lte` o.ä. verfügen. Mittels eines Präprozessors wird beim Bauen des Projekts eine Klasse *QMetadata.class* erzeugt, die die Attribute der Klasse um die entsprechenden Methoden erweitert, die Anfragen, wie die o.g. erlauben. Ebenso wird durch das Beispiel ersichtlich, dass es sich hierbei um Methodenaufrufe auf einem Java-Objekt handelt, jedoch der Anfrage zur Filterung der Events als String übergeben wird (vgl. Listing 8.5).

Der Ausdruck muss also zur Laufzeit in ausführbaren Java-Code übersetzt werden, was mittels der Ausdruckssprache *MVEL* erreicht wird. Diese Ausdruckssprache ist an die Java-Syntax angelehnt, sodass der String mit dem Filterausdruck äquivalent zu Java-Code ist. Um nun ein `Predicate`-Objekt zu erhalten, welches vom QueryDSL-Framework benötigt wird, um die Abfrage an die Datenbank zu stellen, wird eine Java-`HashMap` erstellt, der als Schlüssel gültige Variablennamen übergeben werden, die in dem Ausdruck vorkommen dürfen, sowie deren entsprechendes Klassenattribut als Wert, wie man es beispielhaft in Listing 8.7 nachvollziehen kann. MVEL wertet den Ausdruck aus, ordnet die Variablen im Ausdruck denen der Zielklasse zu und erzeugt das gewünschte Objekt, in diesem Fall das `Predicate`.

```
1 public static Predicate toPredicate(final String
     filterExpression){
2     Map<String, Object> vars = new HashMap<>();
3     vars.put("eventNum", QMetadata.metadata.eventNum);
4     vars.put("triggerNum", QMetadata.metadata.triggerNum);
5     ...
6     return (Predicate) MVEL.eval(filterExpression, vars);
7 }
```

Listing 8.7: Evaluation der Anfrage

Nach der Erzeugung des `Predicate` Objekts kann dieses an das entsprechende Repository übergeben werden, wie es beispielsweise in Listing 8.8 umgesetzt wurde. Die Methode `findAll(...)` dient der Suche aller Events (bzw. Metadaten), die dem Prädikat genügen.

```
1 @Override
2 public Iterable<Metadata> filterEvents(String
     filterExpression) {
```

```
3     return metadataRepository.findAll(Metadata.toPredicate(
        filterExpression));
4 }
```

Listing 8.8: Service Implementierung

Für gewöhnlich akzeptiert diese Methode des Spring-Repositorys kein `Predicate`-Objekt als Parameter. Daher muss das Repository insofern angepasst werden, als dass es ein weiteres Interface (`QueryDslPredicateExecutor<Metadata>`) implementiert, das von *QueryDSL* bereitgestellt wird und dem Repository die Fähigkeit verleiht, Prädikate zur Filterung von Datenbankeinträgen zu nutzen. Damit der `QueryDslPredicateExecutor` das Prädikat für das jeweilige Datenbanksystem ausführen kann, muss lediglich die entsprechende Maven Dependency eingebunden werden, die die nötige Logik enthält. Eine solche Depedency ist für die populärsten Systeme vorhanden, sodass eine Integration problemlos und schnell umgesetzt werden kann.

Ein Spring-Repository zeichnet sich dadurch aus, dass es ein Interface ist, dessen definierte Methoden zur Übersetzungszeit des Projekts automatisch vom Spring-Framework implementiert werden, wie dem Beispiel in Listing 8.9 zu entnehmen ist. Durch diesen Mechanismus garantiert die Einbindung des `QueryDslPredicateExecutors`, dass die benötigten Methoden wie `findAll(Predicate predicate)` ohne zusätzliche Arbeit implementiert werden.

```
1 public interface MetadataRepository extends MongoRepository<
    Metadata, String>,
2         QueryDslPredicateExecutor<Metadata>
3 {
4 }
```

Listing 8.9: Metadata Repository für die MongoDB

Fazit Mit der Kombination verschiedener Frameworks und Bibliotheken ist es gelungen, einen Ansatz zu entwickeln, der den Zugriff auf die Persistierungsebene und die Auswertung der Anfragen vereinheitlicht und sich somit generisch an verschiedenste Datenbanksysteme anpassen lässt. Der Vorteil dieses Ansatz liegt insbesondere in der Wartbarkeit, Anpassbarkeit und der Reduktion des Codes zur Implementierung der benötigten Features. Im Vordergrund steht hierbei insbesondere die automatisierte Auswertung komplexerer Anfragen zur Filterung der persistierten Daten.

Bisher wurde jedoch nur von dem Fall ausgegangen, dass der Filter korrekt angewandt wurde. Durch eine fehlerhafte oder absichtlich böswillige Query könnte Schadcode injiziert werden, was bisher nicht überprüft wird, sodass der aktuelle Fortschritt eher als

HTTP-Methode	URL	Beschreibung
POST	/jobs/start	Starten einen im Body übergebenen Job.
GET	/jobs/active	Gibt eine Liste aller aktiven Jobs im Cluster zurück.
POST	/jobs	Speichert einen Job als Template.
GET	/jobs	Gibt eine Liste aller gespeicherten Jobs zurück.
GET	/jobs/id	Gibt einen gespeicherten Job zurück.
DELETE	/jobs/id	Löscht einen gespeicherten Job.

Tabelle 8.2: Schnittstellen der REST API für Jobs

Proof of Concept bezeichnet werden kann. In einer weiteren Iteration müsste überprüft werden, ob der übergeben Ausdruck tatsächlich in ein Prädikat übersetzt werden kann und die Eingabe auf die Prädikatausdrücke beschränkt werden. Im Fehlerfall muss mit einer Exception o.ä. reagiert werden.

8.2.3 Jobs

Dies RESTfull API soll jedoch nicht nur das Durchsuchen von Events ermöglichen, sondern insbesondere auch die Interaktion mit dem Apache Spark Cluster vereinfachen. Daher wurden in der Schnittstelle auch Wege umgesetzt um *Jobs* zu starten, verwalten, speichern und auch Ausführungen zu planen. Durch diese Abstraktion des Clusters kann mit diesem interagiert werden, ohne dass spezielle Software auf dem Client installieren werden muss oder besondere Einstellungen getroffen werden müssen.

Ein *Job* bezeichnet dabei eine durch ein Stream-XML definierte Aufgabe, die per Streams auf Apache Spark ausgeführt werden soll. Neben dem XML können noch weitere Ausführungsparameter wie die Anzahl von Cores, der zuverwendende Arbeitsspeicher usw. festgelegt werden. Dies hilft, die zur Verfügung stehenden Resourcen effektiv zu nutzen, oder beim Testen der Skalierbarkeit.

Apache Spark bzw. YARN stellen ihrerseits schon umfangreiche und teilweise auch REST-basierte Werkzeuge bereit. Die im folgenden beschriebene API soll diese nicht ersetzten oder in Konkurrenz zu diesen sein, sondern ist mehr eine Ergänzung, die auf die Aufgabenstellung zugeschnittene Optionen anbietet. Bei manchen Endpunkten werden im Hintergrund auch die mitgelieferten Tools verwendet.

Eine Übersicht der Endpunkte ist in Table 8.2 aufgelistet.

Starten

Das Starten von Jobs erfolgt per HTTP-POST-Aufruf von */api/jobs/start* mit einer Job-Beschreibung als JSON im Body. Diese Job-Beschreibung enthält einen eindeutigen Namen für den Job, das Streams-XML, die Anzahl von Spark Executors, der zu verwendende Haupstpeicher für die Excecutors und die Spark Driver sowie die Anzahl der Kerne für die Executors und für die Driver.

Neben diesen Parametern für den Spark-Kontext kann noch optional eine Jar-Datei übergeben werden. Dies ermöglicht es, neue Prozessoren, Operatoren etc. auszuprobieren, ohne die REST API verändern zu müssen. Diese Idee entstand aus dem Wunsch, dass unterschiedliche Arbeitsgruppen der PG verschiedene Funktionen testen wollten.

Auf dem Server, auf welchem die REST API gestartet wurde, liegt eine Standard-Jar-Datei, die verwendet wird, wenn keine explizit übergeben wird,

Nachdem versucht wurde, den Job zu starten, wird eine Antwort zurückgegeben, die Auskunft über den Erfolg oder Misserfolg enthält (vgl. Listing 8.10).

```
1 {
2     "name": "Test Job",
3     "error": false,
4     "message": "The job started successfully.",
5     "startDate": "20130921"
6 }
```

Listing 8.10: Beispeil Antwort

Die Implementierung dieser Endpunkte wurde auch über das Spring Framework umgesetzt. Die Kommunikation mit Spark erfolgt jedoch nicht über eine spezielle Java-API, sondern verwendet ein externes Shell-Skript. Das übergebene XML wird in eine temporäre Datei geschrieben und aus den anderen Parametern der entsprechende Skript-Aufruf generiert.

Verwalten

Das Webinterface bietet auf der Startseite als Hauptmerkmal eine einfache Möglichkeit zum Starten von Jobs und erleichtert somit die Interaktion mit dem Cluster. Nach dem Starten eines Jobs möchte der Nutzer naturgemäß über den Zustand seines ausgeführten Jobs im Bilde bleiben, um beispielsweise entsprechend reagieren zu können, wenn dieser erfolgreich beendet oder mit einem Fehler abgebrochen wurde. Dazu bietet YARN, welches die Ressourcen des Clusters dynamisch für die verschiedenen Jobs verwaltet, unterschiedliche Informationen über sein eigenes Webinterface an. Jedoch wird bei kontinuierlicher Nutzung des Webinterfaces recht deutlich, dass ein Wechsel zum Interface des YARN-Clusters umständlich ist und das Interface oft mehr Informationen anbietet als grundsätzlich von Interesse sind. Daher lag es nahe, eine entsprechende Übersicht mit den laufenden und bereits terminierten Jobs in das Webinterface zu integrieren und die gewünschten Informationen tabellarisch darzustellen.

Als Untermenge der von YARN bereitgestellten Informationen bot sich u.A. die ID des laufenden Jobs sowie sein Name an. Der Name wird beim Starten des Jobs vergeben und an YARN übermittelt, während die ID von Nutzen ist, um weitere Informationen über einen Job von YARN anzufordern. Für den Abruf (weiterer) Informationen stellt YARN entsprechende Endpunkte über eine Schnittstelle zur Verfügung. Die bereitstehenden Informationen werden im Folgenden erläutert, während die Details zur Kommunikation mit der Schnittstelle im Abschnitt 8.3.2 näher betrachtet werden.

Zudem enthält die Tabelle Einträge, die den Startzeitpunkt eines Jobs dem Anwender offenbaren und der aktuelle Fortschritt des Jobs wird als Prozentzahl ausgedrückt. Über diesen Wert kann in Kombination mit dem Startzeitpunkt im laufenden Betrieb insbesondere die Geschwindigkeit der Verarbeitung des Jobs durch den Anwender abgeschätzt werden und stellt somit eine sinnvolle Metrik dar. Der Zustand („state“) kann folgende Werte annehmen: „NEW, NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED und KILLED“. Die interessantesten Attribute sind hier sicherlich *RUNNING*, *FINISHED* und *FAILED*. Mit ihnen wird ausgegeben, ob der jeweilige Job ausgeführt wird, beendet wurde oder die Ausführung fehlgeschlagen ist. Dabei bezieht sich dieser Zustand auf die Ausführung des Jobs durch den Ressourcemanager. Auch ein erfolgreich ausgeführter Job kann im Endeffekt fehlgeschlagen sein, wenn das auszuführende Programm, welches durch den Job ausgeführt wurde, mit keinem erfolgreichen Resultat beendet wurde. Dies wird durch den finalen Status („finalStatus“) ausgedrückt, der den Status des Jobs ausgibt, der von dem auszuführenden Programm bzw. dem Job zurückgegeben wurde. Der Status kann die Werte „UNDEFINED, SUCCEEDED, FAILED und KILLED“ annehmen.

Der Vollständigkeit halber wird in der Tabelle 8.3 auf weitere, mögliche Attribute eingegangen. Aus dieser Tabelle wird schnell ersichtlich, dass die wichtigsten Werte für den jeweiligen Job bereits im Webinterface einzusehen sind.

Neben den jobspezifischen Parametern stehen auch Metriken zum Clusterzustand zur Verfügung. Darunter fällt beispielsweise das Attribut *äctivesNodes¨*, welches die Anzahl der aktiven Knoten ausgibt oder die Attribute *l̈ostNodes¨* und *ünhealthynodes¨*: Sie weisen auf möglicherweise ausgefallene Knoten hin, die dem Cluster nicht mehr zur Verfügung stehen oder auf Knoten, die sich in einem Zustand befinden, in dem sie nicht in der Lage sind Teile eines Jobs auszuführen. Besonders im Hinblick auf die Verlässlichkeit eines Clusters könnte anhand dieser Parameter entschieden werden, ob dem Cluster neue Jobs zugeführt werden sollten oder eine verringerte Verarbeitungsgeschwindigkeit der Jobs auf einen fehlerhaften Cluster zurückzuführen sind. Somit wäre die Einbindung dieser Attribute im Webinterface eine potentielle Verbesserung.

Im Folgenden werden mögliche Verbesserungen konzeptuell diskutiert, die das Entfaltungspotential und die Notwendigkeit einer differenzierten Jobverwaltung über das Webinterface hervorheben und unterstreichen.

Momentan wird die Untermenge an dargestellten Informationen bzw. Attributen eines Jobs vorgegeben. Vorstellbar wäre, dass ein Anwender die für ihn relevanten Attribute auswählt und die Tabellenansicht entsprechend angepasst wird. So wird kann die Anwendung benutzerspezifischer gestaltet werden.

Die Tabelle, die die Jobs und ihre jeweiligen Attributen beinhaltet, wird lediglich durch das Neuladen der Browserseite oder durch das Drücken des Buttons am Kopfende der Seite aktualisiert. Um den Komfort zu erhöhen, wäre ein automatisiertes Aktualisieren

id	Eine eindeutige Job ID
user	Der Name des Nutzers, der den jeweiligen Job ausgeführt hat
name	Der Name der Applikation, der über das Interface mitgegeben wird
Application Type	Die Art der Applikation
queue	Die Warteschlange, in der der Job eingereiht wurde
state	der Zustand der Applikation im Ressourcemanager
finalStatus	Der Status, der von der Applikation bzw. dem Job zurückgemeldet wurde
progress	Der Fortschritt der Verarbeitung des Jobs als Prozentzahl
trackingUI	Ort der Logs: Application Master oder History Server, auf den die Logs ausgelagert wurden
trackingUrl	Die URL zu dem YARN Interface, über die der Zustand des Jobs bereitgestellt wird
diagnostics	Detaillierte Diagnoseinformationen
clusterId	Die ID des Clusters, auf dem der Job ausgeführt wird
startedTime	Die Zeit, zu der der Job gestartet wurde
finishedTime	Die Zeit, zu der der Job beendet wurde
elapsedTime	Die Zeit (in ms), die vergangenen ist, seitdem der Job gestartet wurde
amContainerLogs	Die URL der Container Logs
amHostHttpAddress	URL des Application Servers
allocatedMB	Der Speicherplatz, der für die Container zur Ausführung des Jobs reserviert wurde
allocatedVCores	Die Anzahl virtueller Kerne, die reserviert wurden
runningContainers	Die Anzahl der Container, die für den Job ausgeführt werden
memorySeconds	Den Speicherplatz, den die Applikation/der Job belegt hat
vcoreSeconds	Die Anzahl der CPU Ressourcen, die die Applikation belegt hat

Tabelle 8.3: Abrufbare Attribute eines YARN Jobs

der Seite wünschenswert. Noch besser wäre es, wenn die Seite lediglich dann aktualisiert würde, wenn tatsächlich eine Änderung stattgefunden hätte. Vorstellbar wäre hier eine feingranulare Beobachtung der Attribute hinsichtlich jeder Änderung wie beispielsweise dem Fortschritt oder eine grobgranularere Beobachtung, bei der die Seite lediglich hinsichtlich von Status und/oder Zustandsänderungen eines Jobs aktualisiert wird, um die Anfragen an den YARN Server zu verringern.

Um den Nutzer während einer Ausführung länger andauernder Jobs über den aktuellen Status im Bilde zu halten, wäre ein Benachrichtigungssystem vorstellbar, welches auf verschiedene, womöglich vom Nutzer definierte, Attributsänderungen mit einer Benachrichtung des Nutzers reagiert. Diese Benachrichtigung könnten klassisch per E-Mail erfolgen oder über sog. *Push Notifications*, die von modernen Browsern wie Safari oder Chrome angeboten werden. Dabei handelt es sich um Echtzeitbenachrichtigungen, die von einem Server an Nutzer gesendet werden können, die sich für solche Benachrichtigungen interessieren. Diese Art von Benachrichtigungen sind bereits von mobilen Endgeräten wie Smartphones oder Tablets bekannt, die den Eingang einer kurzen Nachricht für gewöhnlich am oberen Bildschirmrand für einen kurzen Zeitraum einblenden und daraufhin verschwinden. Ähnlich funktioniert es am Computer im Browserumfeld: Selbst wenn der Nutzer sich nicht auf der Website befindet, über die der Nutzer die Benachrichtigungen abonniert hat, so werden die Nachrichten dennoch ähnlich wie beim Smartphone in einem systemnativen Fenster in der rechten Ecke am oberen Bildschirmrand eingeblendet. Im Fall des Safari Browsers muss der Browser nicht einmal aktiv sein, damit der Nutzer weiterhin über Neuigkeiten benachrichtigt wird. Somit könnte ein Anwender zeitnah über beendete oder fehlgeschlagene Jobs informiert werden und wäre von der Pflicht entbunden, in regelmäßigen Abständen selber nach dem Status des Jobs schauen zu müssen.

Auch wenn sich diese Technik womöglich langfristig durchsetzen wird, so sind aktuelle Lösungen meist browserspezifisch und an einen Anbieter (z.B. Apple) gebunden. Um Push Notifications nutzen zu können, muss zunächst ein Zertifikat (beispielsweise von Apple) angefordert werden, welches die Anwendung eindeutig identifiziert. Im zweiten Schritt wird ein sog. *Push Package* auf dem Webserver generiert, welches mit dem Zertifikat signiert und an den Browser des Nutzers ausgeliefert wird. Dieses Push Package wird an den Nutzer ausgeliefert. Mit Hilfe des Push Packages wird Safari auf das Empfangen von Push-Nachrichten der entsprechenden Webanwendung vorbereitet. Möchte der Betreiber der Anwendung nun eine Benachrichtigung an seine Nutzer senden, so wird die Nachricht signiert an den *Apple Push Notification Server* gesendet, welcher wiederum eine Verbindung zum Client aufbaut und für das Auslösen der entsprechenden Benachrichtigung beim Nutzer verantwortlich ist.

So interessant und geeignet dieses Verfahren für die vorgestellte Applikation ist, so umständlich ist es, dieses für mehrere Browser einzurichten. Auch werden lediglich aktuelle Browserversionen unterstützt, da es sich dabei um ein verhältnismäßig neues Feature

HTTP-Methode	URL	Beschreibung
POST	/tasks	Plant einen Job.
GET	/tasks	Gibt eine Liste aller geplanten Jobs zurück.
PUT	/tasks	Aktualisiert einen geplanten Job.
DELETE	/tasks/id	Löscht einen geplanten Job.

Tabelle 8.4: Schnittstellen der REST API für geplante Jobs

handelt. Im Abschnitt 8.3.2 wird jedoch beispielhaft auf die Implementierung von *Local Notifications* unter Berücksichtigung des Safari Browsers eingegangen. Diese sind insofern eine vereinfachte Ausführung von Push Notifications, als dass sie durch ein JavaScript der aufgerufenen Website ausgeführt bzw. an den Anwender gesendet werden. Der Nachteil hierbei ist jedoch, dass die Webseite und der Browser definitiv geöffnet sein müssen, damit der Nutzer die Nachrichten empfangen kann. Jedoch stellt dieses Verfahren eine passable Alternativlösung zu den Push Notifications dar. Insbesondere im Bezug auf den reduzierten Implementierungs- und Verwaltungsaufwand.

Speichern Neben dem einfachen Starten und Überwachen von Jobs gibt es Aufgaben, die, vielleicht nur leicht verändert, öfters ausgeführt werden sollen. Denkbare wäre hier die Standard-Analyse-Kette, die im Wesentlichen unverändert bleibt, jedoch auf unterschiedlichen Daten aufgerufen werden soll. Ein anderes Beispiel ist das Testen der Skalierbarkeit, bei dem ein Job mit unterschiedlichen Spark-Parametern aufgerufen wird oder das Ausprobieren von verschiedenen Implementierungen von Prozessoren, bei der sich nur die PG-Jar-Datei ändert.

Daher wurde die Möglichkeit geschaffen, Jobs zu Speichern und diese gespeicherten Jobs zu verwalten. Eine Liste der dazugehörenden Endpunkte ist ebenfalls in Table 8.2 aufgelistet. Diese sind im wesentlichen Klassische CRUD-Operationen, die mit dem Spring Framework und der MongoDB als Datenbank umgesetzt wurden.

Scheduling

Als Erweiterung von vordefinierten Jobs bietet es sich an, diese auch automatisch zu bestimmten Zeitpunkten zu starten. Dies Ermöglicht eine bessere Auslastung des Clusters, da die Jobs nun von den klassischen Arbeitszeiten der Nutzer entkoppelt sind.

Ein geplanter *Job* oder *Task* wird dabei über einen Namen definiert und enthält die Id des Jobs, welcher gestartet werden soll, ob der Task aktiviert ist oder nicht und zu welchen Zeitpunkten dieser ausgeführt werden soll, wenn er aktiv ist. Zum Definieren der Zeitpunkte wurde sich dabei an der Syntax des aus der Unix-Welt bekannten Cron-Deamon orientiert, welche Zeitpunkte über Minuten, Stunden, Tag im Monat, Monaten und Wochentag beschreibt, wobei auch Wildcards erlaubt sind.

Die Endpunkte wurden dabei auch wieder mit dem Spring Framework und der MongoDB als Datenspeicher umgesetzt. Eine Übersicht der Endpunkte findet sich in Table 8.4.

8.3 Ein Beispiel-Client: Die Web-UI

Wie weiter oben in der Einführung zu REST APIs beschrieben, bietet eine RESTful API den Vorteil, dass diese auf das weit verbreitetet HTTP aufbauen und damit im Allgemeinen einfach verwendet beziehungsweise in andere Anwendungen integriert werden können. Jedoch ist die Kommunikation für Menschen über HTTP nicht wirklich intuitiv. Es gibt allgemeine Tools, die speziell für die Interaktion mit unterschiedlichen REST APIs ausgerichtet sind, wobei diese natürlich nicht auf spezielle Anforderungen der verschieden Anwendungsfälle eingehen können.

Daher wurde neben der eigentlichen API auch ein Client entwickelt, der die API verwendet und dabei speziell auf die Analyse-Anforderungen zugeschnitten wurde. Um den Vorteil der Plattformunabhängigkeit und Portabilität nicht zu verlieren, wurde sich für eine Web-UI entschieden, da diese nur einen modernen Browser bei den Anwendern voraussetzt, welcher bei den meisten Systemen im Allgemeinen vorhanden ist (vgl. chapter 18). Darüber hinaus läuft die REST API schon auf einem Server, sodass für das Bereitstellen der Web-UI kein großer zusätzlicher Aufwand auf der Server-Seite betrieben werden muss.

8.3.1 Single Page Applications

Ein populäres Konzept für solche Web-UIs sind sogenannte Single Page Application. Wie der Name es vermuten lässt, liefert der Server nur eine HTML-Seite aus, die dann beim Client im Browser dynamisch über JavaScript angepasst wird und so auch den Eindruck von mehreren Seiten erzeugen kann. Während der erste Aufruf der Seite also wahrscheinlich etwas länger dauert als bei klassischen Webservices mit mehreren Seiten, so ist die Interaktion mit der Single Page Application danach deutlich schneller, da die zum Anzeigen benötigten Daten schon beim Client sind und nur noch die Daten vom Server nachgeladen werden müssen. Dieses Nachladen kann dabei natürlich auch durch spezielle Warte-Animationen verdeutlicht werden, was für Benutzer ein insgesamt flüssigeren Nutzererlebnis erzeugt. Weiterhin wird damit die Last des Servers reduziert, da dieser nur eine statische HTML-Seite und die Daten über API ausliefern, jedoch nicht das Anzeigen der Daten bearbeiten muss. In Kombination mit einer RESTful API bedeutet dies auch, dass der Zustand der Nutzer-Sitzung beim Client gespeichert werden kann.

8.3.2 Implementierung

Für Singe Page Applications gibt es viele JavaScript-Frameworks. Für die Web-UI wird das von Google unterstütze *AngularJS* verwendet.

Zunächst sei erwähnt, dass wir den Node Package Manager (NPM) verwenden, um *AngularJS* und die anderen benötigten Abhängigkeiten zu verwalten. Dies erlaubt es uns, die verwendeten Bibliotheken in der richtigen Version mittels des Befehls *npm install* herunterzuladen, ohne dass wir mit diesen unser Projektrepository überladen.

Weiterhin wird *Grunt* als Build Managment Tool verwendet, um die Web UI zu bauen. *Grunt* ist selber in JavaScript geschrieben und kann auch mittels NPM verwaltet werden.

Um ein modernes Design zu erreichen, wurde das CSS Framework *Bootstrap* verwendet. *Bootstrap* wurde ursprünglich von einem Team von Twitter entwickelt, jedoch kommt es nun, auch wegen seiner OpenSource Lizenz, überall bei unterschiedlichsten Projekte zum Einsatz und ist eines der populärsten CSS Frameworks der Welt. Seine Stärke liegt in unterschiedlichsten, vordefinierten Komponenten, die durch einfache (Klassen-)Annotationen an bestehendes HTML angehängt werden können.

Um die unterschiedlichen Anforderungen wie Jobs zu starten oder zu verwalten auch in der Oberfläche zu trennen, wurde das sogenannte *Rounting* von *AngularJS* verwendet. Dabei wird jedem Zustand der Oberfläche eine eigene Sub-URL nach dem Muster *http://...app.html#/jobs* gegeben. Was zunächst wie ein Widerspruch zu einer Single Page Application aussieht, hilft jedoch, die Vorteile von Webseiten mit vielen Seiten in diese zu integrieren. Da sich nur der Teil der URL hinter der Raute verändert, bleibt es bei einer Single Page Application. Jedoch ist es auch so möglich, bestimmte Bereiche direkt anzusprechen, zum Beispiel als Link oder Bookmark.

Ein wesentlicher Aspekt der Web UI wird es auch das Angeben der Streams-XML-Definitionen sein. Daher wurde auf das Projekt *ACE* als Editor-Komponente zurückgegriffen. Anders als ein einfaches, großes Texteingabefeld ermöglicht ACE, mittels einer Syntax-Definition wichtige Schlüsselwörter hervorzuheben oder Fehler in der Eingabe direkt zu markieren. Dadurch wird die Spezifikation des XML-Texts für Streams einfacher und weniger fehleranfällig.

Abschließend wurde eine eigene Upload-Komponente entwickelt, um die PG-Jar-Datei an die REST API als BLOB zu übergeben.

Jobs verwalten

Die Verwaltung der Jobs verteilt sich in Hinblick auf die Implementierungsdetails auf zwei Bereiche. Zum Einen bedarf es der visuellen Darstellung der Informationen im Webinterface (Frontend) und zum Anderen des Abrufens, Parsens und Bereistellung der benötigten Informationen (Backend).

Frontend AngularJS implementiert das MVVM (Model-View-ViewModel) Pattern. Das *View* ist hierbei die Tabelle, welche für die Anzeige der Daten verantwortlich ist, die von einem Controller bereitgestellt werden.

Die Tabelle wird dabei als Template in der Single-Page-Application eingebunden. Das Template ist in einem `<script>`-Tag eingebettet und repräsentiert regulären HTML Code mit zusätzlichem Markup. Dieser wird von AngularJS beim Start der Applikation verarbeitet und in eine übliche DOM-Struktur überführt, die vom Browser interpretiert werden kann.

Der Controller stellt die benötigten Methoden in einem isolierten Scope bereit, auf den vom Template aus zugegriffen werden kann. Eine solche Methode ist z.B. das Abrufen der benötigten Informationen vom Backend über ein asynchrones *XMLHttpRequest*, welches von AngularJS durch den `$http`-Service ausgeführt werden kann. Dem `$http`-Service wird ein Konfigurationsobjekt übergeben, mit dem eine HTTP-Anfrage generiert wird und das schließlich in der Rückgabe eines Promise-Objekts resultiert. Dieses repräsentiert im Prinzip einen Proxy für einen Wert, der zum Zeitpunkt seiner Erstellung noch nicht (unbedingt) bekannt ist. Im Prinzip versteht sich das Objekt als Platzhalter für den tatsächlichen Wert und verspricht somit implizit, dass dieser Wert nicht unbedingt zum momentan Zeitpunkt, aber irgendwann in der Zukunft bereitstehen wird. Durch die Verwendung eines solchen Proxyobjekts wird ein asynchroner Programmablauf in einem sonst synchronen Code ermöglicht. Die Asynchronität hat zur Folge, dass die Applikation während der Anfrage an den Server weiterhin genutzt werden kann und nicht blockiert. Ein Promiseobjekt kann sich in den Zuständen *pending* (initialer Status), *fulfilled* (Ausführung der Operation war erfolgreich) oder *rejected* (Operation ist gescheitert) befinden. In diesem Fall bedeutet der Zustand *pending*, dass der HTTP-Request abgesendet, aber noch nicht beantwortet wurde. Wird der Zustand *fulfilled* erreicht, so hat der Endpunkt der REST-API eine Antwort geliefert, die in diesem Fall die Informationen zu den jeweiligen Jobs umfasst und über den isolierten Scope bereitgestellt wird.

In AngularJS werden *Components* als Alternative zu den sonst üblichen Direktiven eingesetzt, um eine Single-Page-Application besser zu strukturieren. Dazu werden Components über die *module*-Methode des globalen AngularJS-Objekts in die Applikation eingebunden und kapseln das View sowie seine Daten und den assoziierten Controller. Der *Job History Component* stellt dem View beispielsweise die Jobs und ihre entsprechenden Attribute als Model zur Verfügung.

Backend Das Backend, sprich die REST-API, stellt einen Endpunkt zur Verfügung, der auf einen HTTP-GET-Request mit den Jobinformationen im JSON-Format antwortet. Die Informationen müssen vom YARN Cluster bezogen werden, der selbst wiederum eine REST-API anbietet, um die entsprechenden Informationen anzufragen. Der erste Gedanke wäre die YARN API direkt vom Frontend aus anzusprechen und diese über Controller-Logik des *Job History Components* zu verarbeiten. Jedoch scheitert dieser Versuch an der *Same-Origin-Policy*, die ein browserseitiges Sicherheitsfeature ist, die den Zugriff auf andere Objekte (JavaScript, Grafiken, CSS Stylesheets u.v.m.) und Websites

anderen Ursprungs untersagt. Damit sollen Angriffe durch das Einbinden fremder Skripte unterbunden werden, jedoch führt dies auch leider zu den o.g. Einschränkungen.

CORS und JSONP Unter Cross-Origin Resource Sharing (CORS) wird eine Methode verstanden, die die auferlegten Restriktionen der Same-Origin-Policy des Browsers aufheben lässt. Damit ein solcher Cross-Origin-Request erfolgreich durchgeführt werden kann, muss der Server mit der abweichenden Domain den Header seiner HTTP-Response anpassen. Durch das Hinzufügen des *Access-Control-Allow-Origin* Schlüssels wird der Aufruf von Seiten ausnahmsweise erlaubt, die unter die Domain fallen, die als Wert des Schlüssels gesetzt wird. Die Funktionalität dieser Lösung ist wie so oft bei der Frontendentwicklung von der Art und Version des eingesetzten Browsers abhängig. CORS wird auch von YARN Schnittstelle unterstützt, muss jedoch gesondert konfiguriert werden und steht standardmäßig nicht zur Verfügung, sodass ohne weitere Maßnahmen kein Zugriff auf diese Schnittstelle direkt aus dem Frontend heraus erfolgen kann.

Als Alternative zu CORS existiert außerdem JSONP (JSON mit Padding), der eine Ausnahme in der Same-Origin-Policy zum Nachladen weiterer Inhalte eines anderen Servers ausnutzt. Browser erlauben nämlich die Referenzierung beliebiger URLs im `src` Attribut eines `<script>` Tags. Somit wird eine Übermittlung von Daten auch über Domaingrenzen hinweg ermöglicht. Wird auf diese Weise ein Endpunkt referenziert, der JSON-basierte Daten zurückliefert, hat dies jedoch keinen Effekt und kann nicht vom Entwickler ausgenutzt werden, um die Daten in sein Model (o.Ä.) einzubinden. Hier kommt der Zusatz *Padding* ins Spiel: Der URL im `src` Attribut wird als Query String (bzw. GET Parameter) der Name einer JavaScript-Funktion, die bereits im Frontendcode vorhanden ist, hinzugefügt. Es handelt sich also um eine Art Callback-Funktion. Der Server wiederum antwortet mit JavaScript Code, der die angefragten Informationen (JSON-Daten) kapselt und über das `script`-Tag eingebunden wird. Der vom Server übertragene JavaScript-Code ruft schließlich die Funktion als Callback auf, deren Namen zuvor übertragen wurde, und übergibt die gekapselten Daten über einen Paramter dieser Callback-Funktion.

Der `$http`-Service von AngularJS unterstützt JSONP-Anfragen über die gleichnamige `jsonp`-Methode, jedoch hat dies im Zusammenspiel mit der YANR REST API nicht funktioniert. Schlussendlich schien nach der Abwägung der Vor- und Nachteile,die Implementierung eines zusätzlichen Endpunkts, der sich um die Anfrage und Verarbeitung der Informationen des YARN Clusters kümmert, am sinnvollsten.

Kommunikation mit YARN Der YARN RessourceManager erlaubt es, clusterbezogene Informationen über eine eigene REST API zu beziehen. Dabei kann der Zustand des Clusters, Metriken zum Cluster, Scheduler und Node Informationen sowie Informationen zu den ausgeführten und auszuführenden Jobs erfragt werden.

Unter der URL `http://<rmhttpaddress:port>/ws/v1/cluster/apps` wird ein verschachteltes JSON-Objekt zurückgeliefert, welches unter dem Wurzelschlüssel `apps` einen weiteren Schlüssel `app` enthält, der wiederum als Wert ein Array aus Objekten annimmt, die die einzelnen Jobs repräsentieren, die vom RessourceManager verwaltet wurden und werden. In den Job-Objekten stehen alle Informationen bereit, die bereits in der Tabelle 8.3 aufgeführt wurden.

Zum Aufruf der YARN API wird der `jersey-client` genutzt, über den der Header der Anfrage an die YARN API insofern manipuliert wird, als dass der Wert des Schlüssels `Accept` auf `application/json` gesetzt wird, um die Schnittstelle zur Ausgabe von JSON-basierten Daten zu zwingen. Mithilfe der von Google bereitgestellten `gson`-Bibliothek werden die JSON Daten geparst, das Job-Array traversiert und die gewünschten Informationen extrahiert. Im Anschluss wird ein neues JSON-Objekt aus einem POJO (Plain Old Java Object) erzeugt, welches als Attribute die extrahierten Informationen enthält.

Notifications Um den Anwender über Job-Status-Updates zu informieren, sind zwei Kommunikationswege vorstellbar. Die simplere Variante verfolgt den Ansatz des Pollings, bei dem das Frontend in vorgegebenen Zeitintervallen Anfragen an den Endpunkt der REST API sendet, um über neue Updates in Kenntnis gesetzt zu werden. Neben dem Polling ist aber auch eine Socketverbindung vorstellbar, die beispielsweise mit der *socket.io*-JavaScript-Bibliothek umgesetzt werden könnte. Das Frontend baut eine dauerhaft bestehende Verbindung zum Backend auf, über die sowohl Frontend als auch Backend Nachrichten senden können, die wiederum zeitnah beim Eintreffen der Nachricht verarbeitet werden. Ein Vorteil der o.g. Bibliothek ist auch, dass sie Fallbackstrategien anbietet, sofern Socketverbindungen vom Browser nicht unterstützt werden und der implementierte Socketendpunkt im Backend diese auch implementiert hat. Leider bietet YARN lediglich eine REST API an und lässt keine Socketverbindungen zu, sodass zumindest im Backend Polling betrieben werden müsste, um aktuelle Informationen über Jobs abzurufen. Im Endeffekt würde sich also leider der Mehraufwand für die Implementierung einer Socketverbindung zwischen Frontend und Backend nicht lohnen. Dadurch, dass Jobs aber i.d.R. eine ganze Weile laufen und sich der Status i.d.R. nicht in kurzen Abständen ändert, ist das Polling eine akzeptable Umsetzung.

Um Änderungen des Models im Frontend feststellen zu können, muss bei jeder periodischen Anfrage der Daten am Backend das aktuell vorhandene Model mit den eingehenden JSON-Daten verglichen werden. Die JavaScript-Bibliothek *Lodash* bietet hierfür die Methode `isEqual(value, other)` an, die einen Vergleich zweier JavaScript-Objekte mit beliebiger Tiefe ermöglicht. Im betrachteten Fall werden zwei Arrays bzw. die Schlüssel-Wert-Paare der beinhalteten Job-Objekten verglichen. Bei der Änderung eines Wertes wird (beispielhaft im Falle des Safaribrowsers) ein Notification-Objekt erstellt, dem ein Titel und eine Nachricht übergeben wird. Damit jedoch Notifications durch das Notificationcenter angezeigt werden, wird zunächst eine Erlaubnis über die Methode `Notification.requestPermission()`

erfragt. Wenn der Nutzer Notifications durch die Website erlaubt, werden diese in Zukunft rechts am oberen Bildschirmrand eingeblendet. Dafür muss die Website lediglich in einem Browsertab geöffnet sein, allerdings nicht aktiv genutzt werden.

Events zählen

Mit dem *Reststream* wurde ein Stream entwickelt, der u.a. als Parameter einen Filterausdruck entgegennimmt und die FITS-Dateien zu denjenigen Events zurückliefert, deren Metadaten dem gegebenen Filterausdruck genügen, wie es bereits im Abschnitt 8.2.2 erläutert wurde. Das Webinterface wurde um die Funktionalität erweitert, einen solchen Filterausdruck vor dem Einsatz in der Analysekette auszuführen, um zum Einen die Korrektheit der Syntax überprüfen zu können und zum Anderen a priori eine Rückmeldung darüber zu bekommen, wie viele Events von der gegebenen Anfrage betroffen sind und ob eine Analyse somit überhaupt lohnenswert ist.

Das Userinterface für diese Funktion ist bewusst einfach gehalten und besteht aus einem Eingabefeld für den Filterausdruck und einer Ausgabe mit der Anzahl der gezählten Events, die von dem Filterausdruck erfasst werden. Dabei erscheint die Ausgabe erst, sobald das Ergebnis, sprich die Anzahl der gezählten Events zur Verfügung steht. Dafür wird auch an dieser Stelle von dem zuvor vorgestellten Promise-Objekt des `$http`-Services Gebrauch gemacht. Beim Klick auf den Button wird die zugehörige Beschriftung solange geändert bis ein Ergebnis vorliegt bzw. die übergebene Callbackfunktion vom Promise-Objekt aufgerufen wird.

Im Backend wird die Anfrage ähnlich ausgewertet, wie es auch im Abschnitt 8.2.2 erläutert wird. Lediglich zum Zählen der Events wird im Spring Repository auf die `count()`-Methode zurückgegriffen, die dynamisch vom Spring Framework für alle Datenbankanfragen zur Verfügung gestellt wird und eine effiziente Zählung ermöglicht.

Kapitel 9

Verteilung von Streams-Prozessen

Das für unsere Erweiterung verwendete Apache Spark verteilt die Last der Datenverarbeitung über ein Rechencluster. Dieses Konzept skaliert sehr gut horizontal, d.h., die Performanz lässt sich durch Anbindung weiterer Cluster-Knoten steigern. Da horizontale Skalierbarkeit eine Schlüsseleigenschaft von Big-Data-Anwendungen darstellt (siehe section 2.3), wollen wir die Verarbeitung der Daten im Streams-Framework geeignet mit Spark verteilen.

9.1 Nebenläufigkeit der Verarbeitung

Im Streams-Framework werden Daten in sogenannten Prozessen verarbeitet. Ein Prozess besteht dabei aus einer Kette von Prozessoren, die jeweils Datenelemente transformieren oder Seiteneffekte erzielen (wie z.B. Speicherung von Elementen oder Logging). Jedes Datenelement durchläuft diese Verarbeitungs-Kette sequentiell. Prozessoren sind üblicherweise stateless, wodurch die Verarbeitung jedes Datenelementes unabhängig von der Verarbeitung anderer Datenelemente ist (siehe subsection 3.2.4).

Teilt man die eingehenden Datenelemente in disjunkte Teilmengen (Partitionen) auf, so lässt sich jede dieser Partitionen unabhängig von den anderen verarbeiten. Damit erlaubt die Unabhängigkeit der Datenelemente zueinander eine beliebig nebenläufige Verarbeitung der Daten. Mit Ausnahme der Zusammenführung der Teilergebnisse ist überdies keine Synchronisation zwischen nebenläufigen Verarbeitungspfaden notwendig. Das Gesamtergebnis wird durch die Vereinigung der verarbeiteten Partitionen dargestellt.

Eine verteilte Ausführung eines Streams-Prozess lässt sich also wie folgt umsetzen:

- Datenelemente werden in Partitionen aufgeteilt
- Die Partitionen werden auf Worker-Nodes verteilt verarbeitet
- Die verarbeiteten Partitionen werden zum Gesamtergebnis vereinigt

Den hier vorgestellten Ansatz zur Verteilung der Last werden wir zur Demonstration mit Spark und Spark Streaming umsetzen (siehe section 9.5 und section 9.6). Die zur Nebenläufigkeit der Verarbeitung gewonnenen Erkenntnisse beschränken sich allerdings nicht auf diese Plattformen: Die Partitionierung der Datenströme kann für die Verteilung auf beliebigen Infrastrukturen verwendet werden.

9.2 XML-Spezifikation verteilter Prozesse

Zur Spezifikation verteilter Streams-Prozesse empfiehlt es sich, möglichst nahe an üblichen XML-Konfigurationen für Streams zu bleiben. Dies ermöglicht Anwendern einen schnelleren Einstieg in Streams auf Spark und kann auch den Implementierungs-Aufwand senken. Wie wir sehen werden, müssen bestehende XML-Konfigurationen nur minimal verändert werden, um unsere Erweiterung zu nutzen.

Dazu verwenden wir die bestehenden Tags `stream`, `sink` und `processor` aus dem Streams-Framework wieder, sie verhalten sich damit komplett identisch zu den Framework-Tags. Die einzig neuen Tags zur Verteilung von Streams-Prozessen sind `distributedProcess` (für Streaming-Prozesse) und `batchProcess` (für Batch-Prozesse). Sie unterscheiden sich von Default-Prozessen lediglich durch die Verteilung der Verarbeitung auf Worker-Knoten im Cluster.

Damit ein Prozess verteilbar ist, erwarten wir einen MultiStream als Input. MultiStreams sind Teil des Streams-Frameworks und werden verwendet, um mehrere innere Streams zusammenzufassen, sie z.B. sequentiell abzuarbeiten. Für die Verteilung von Prozessen stellt der MultiStream für uns die Partitionierung der Daten dar (vgl. section 9.1). Jeder innere Stream kann unabhängig von den anderen inneren Streams verarbeitet werden. Wird kein MultiStream als Eingang verwendet, so besteht keine vernünftige Partitionierung und der Prozess wird auf dem Driver (ohne eine Verteilung vorzunehmen) als Standard-Prozess ausgeführt.

Listing 9.1 stellt die Konfiguration einer verteilt ausgeführten Streams-Applikation in XML beispielhaft dar. Es lässt sich gut erkennen, wie wenig sie sich von einer üblichen Streams-Spezifikation unterscheidet: Der Input-MultiStream, die Senke und die Prozessoren sind beliebig. Insbesondere können sämtliche bestehenden Streams, Senken und Prozessoren in einer verteilten Ausführung auf Spark verwendet werden.

Im BigData-Umfeld ist es üblich, ausgesprochen viele Streams zu erzeugen, wie etwa zur Verarbeitung hunderter .fits-Dateien. Weiterhin kann es sinnvoll sein, die Ausgabe verteilt auszuführen. Wir haben dazu einige Implementationen der Sink- und Source-Interfaces entwickelt, die eine verteilte Ein- und Ausgabe ermöglichen (siehe chapter 11).

```
<stream id="IN" class="...">    <!-- arbitrary multistream -->
    <stream id="s1" class="..." />
    <stream id="s2" class="..." />
</stream>

<sink id="OUT" class="..." />    <!-- arbitrary sink -->

<distributedProcess id="PP" input="IN" output="OUT">
    ...    <!-- arbitrary processors -->
</distributedProcess>
```

Listing 9.1: Beispiel-XML für die Nutzung eines DistributedProcess

9.3 Verarbeitung der XML-Spezifikation

Damit die neuen Tags `distributedProcess` und `batchProcess` verwendet werden können, mussten wir Handler für XML-Elemente dieser Tags schreiben. Der bestehende Parser erzeugt Objekte solcher Elemente, welche dann von den neuen Handlern verarbeitet werden. Die Handler haben je eine Factory aufzurufen, die verteilte Streaming- oder Batch-Prozesse erzeugt.

Für die Implementierung der beiden Factories reichte es aus, Methodenaufrufe an die Default-Factory weiterzudelegieren und die Rückgaben anzupassen. Es musste also keine Factory von Grund auf neu implementiert werden. Zunächst erzeugt die Default-Factory Prozess-Konfigurationen, die sich anpassen lassen. So konnten wir den Namen der Klasse, von der ein Prozess-Objekt erzeugt werden soll, in diesen Konfigurationen ändern. In einem zweiten Schritt erzeugt die Default-Factory aus den (geänderten) Konfigurationen Prozess-Objekte. Mit den korrigierten Konfigurationen zeigt diese Erzeugung bereits das gewünschte Verhalten: Es werden Objekte der Typen DistributedProcess und BatchProcess erzeugt.

Für die Umsetzung von Streams, Senken und Prozessoren ist weder ein Handler noch eine Factory erforderlich. Die Angabe des Klassennamens im XML-Element (`class="..."`) realisiert die Erzeugung von Objekten der genannten Klasse bereits. Dieses Verhalten haben wir durch die komplette Wiederverwendung der Tags erzielt.

9.4 Verteilung der Daten

Wie bereits erwähnt, gehen wir davon aus, dass die Daten bereits partitioniert vorliegen als eine Menge von Streams. Offen ist aber die Frage, wie die einzelnen Teile auf die Clusterknoten verteilt werden sollen. Dies ist für die Performanz aus verschiendenen

Gründen entscheidend. Zum Einen sollte die Verteilung so gleichmäßig wie möglich sein, damit die einzelnen Clusterknoten die ihnen zugeteilte Arbeit möglichst gleichzeitig abschließen und nicht aufeinander warten müssen. Zum Anderen sollten die Daten nach dem Code-to-Data Prinzip möglichst auf denjenigen Knoten verarbeitet werden, auf denen sie auch gespeichert sind, um Netzwerkressourcen zu sparen und Verzögerungen zu vermeiden. Es besteht daher das Potential, die Performanz der Analyse gegenüber der trivialen Reihum-Verteilung signifikant zu verbessern. Unsere Architektur bietet die Möglichkeit, basierend auf Speicherort und Größe der Daten eigene Verteilungsstrategien zu definieren. Solche Strategien zu entwickeln und zu untersuchen ist ein Ansatz, um die Performanz des Systems in Zukunft weiter zu verbessern.

Um den Arbeiterknoten mitzuteilen, welche Daten für sie bestimmt sind, werden die IDs der betreffenden Streams übermittelt. Zusätzlich wird die ID des auszuführenden Prozesses und die Beschreibung des Jobs als XML-Objektbaum an alle Knoten gesendet. Jeder Knoten hat so die Möglichkeit, seine Daten aufzurufen und den auszuführenden Prozess zu bestimmen.

9.5 Verteilte Batch-Prozesse

Als ersten Ansatz zur Verteilung von Streams-Prozessen mit Apache Spark verwenden wir das „reine" Spark, in Abgrenzung zu Spark-Streaming. Die Core-Engine von Spark zeichnet sich insbesondere dadurch aus, dass sie ausschließlich Batch-Verarbeitung adressiert. Diese Eigenschaft stellt sich als problematisch heraus, wenn wir mit Datenströmen arbeiten wollen. Es lässt sich jedoch bereits bei diesem Ansatz ein hoher Performanzgewinn gegenüber einer nicht-verteilten Ausführung feststellen.

Wir diskutieren die Umsetzung von verteilten Streams-Prozessen mit der Spark Core-Engine, evaluieren dessen Performanz und nutzen die auftretenden Probleme als Motivation für den Einsatz von Spark-Streaming.

9.5.1 Daten- und Kontrollfluss

Um mit Spark verteilte Operationen auf einem Datensatz durchführen zu können, muss dieser zunächst auf die Arbeiterknoten verteilt werden. Hierzu stellt Spark den Datentyp RDD zu Verfügung, mit dem benutzerdefinierte Operationen auf verteilten Daten ausgeführt werden können. Der erste Schritt des Batch-Prozesses besteht daher darin, die Liste der gegebenen Streams mit Hilfe der `parallelize`-Methode auf die Worker zu verteilen, um so eine RDD von Streams zu erhalten. Hierbei können benutzerdefinierte Verteilungsstrategien verwendet werden, wie in section 9.4 erläutert. Dieser und die folgenden Schritte sind in Figure 9.1 dargestellt. Anschließend muss auf den Arbeiterknoten

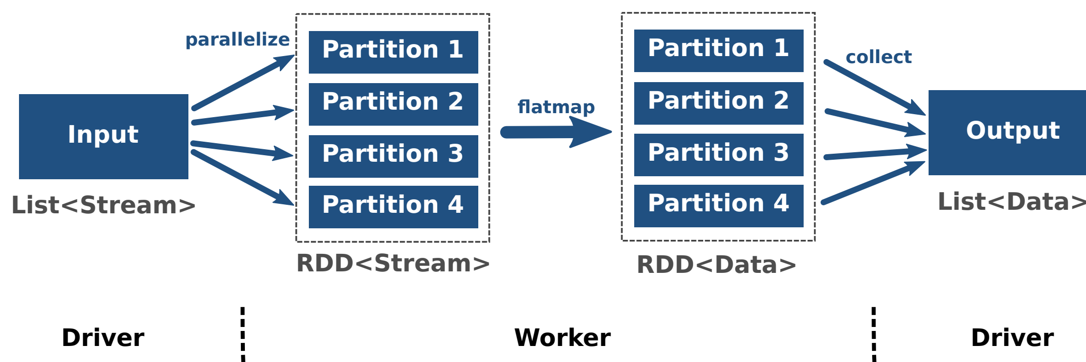


Abbildung 9.1: Datenfluss bei verteilten Batch-Prozessen

der eigentliche Prozess ausgeführt werden, also die Daten aus den Streams gelesen und durch die Prozessoren verarbeitet werden. Da aus einem verteilten Datensatz ein neuer, ebenfalls verteilter Datensatz errechnet werden soll, kommt hier eine der von Spark angebotenen RDD-Transformationen zum Einsatz. In diesem Fall wird jedem Stream (Eingabe) eine Menge von Datenelementen (Ausgabe) zugeordnet, weshalb es sich um eine `flatmap`-Transformation handelt. Was genau auf jedem Knoten passiert, um diese Transformation durchzuführen, wird im kommenden Abschnitt erläutert.

Als dritter Schritt müssen die Ergebnisdaten an den Driver geschickt werden, um dort eventuell auf herkömmliche Art weiterverarbeitet zu werden. Die über die Knoten verteilten Inhalte der Ergebnis-RDD müssen also eingesammelt und zusammengeführt werden. Dies geschieht durch die `collect`-Operation. Hat der Driver die Ergebnisse erhalten, kann er sie einfach in die im XML-Dokument spezifizierte Ausgabe des verteilten Prozesses schreiben.

Eine andere Möglichkeit, die von uns zwischenzeitlich zur Rückführung der Ergebnisse genutzt wurde, ist die Spark-Datenstruktur `Accumulable`. Diese erlaubt es, Daten von allen Knoten zu sammeln, und im Driver auszulesen. Allerdings wird von Spark empfohlen, für den Hauptdatenfluss RDDs zu benutzen, und Accumulables nur für Zusatzinformationen wie etwa Logs zu verwenden. Da sich außerdem experimentell gezeigt hat, dass RDDs performanter sind als Accumulables, haben wir von deren Nutzung Abstand genommen.

9.5.2 Instanziierung von Streams in den Workern

Wie oben gesehen, besteht die Aufgabe eines jedes Knotens darin, einen gegebenen statischen Datensatz zu verarbeiten. Diese Aufgabe wird durch ein XML-Dokument, die ID des auszuführenden Prozesses, und die IDs der zu verwenden Streams spezifiziert, und muss im Rahmen der oben beschriebenen `flatmap`-Operation geschehen. Um dieses Verhalten umzusetzen, ist es vorteilhaft, die `streams`-Klassen zur Verarbeitung des XML-Dokuments wiederzuverwenden. Andernfalls müssten große Teile des `streams`-Codes zur

Erzeugung der Ausführungsumgebung reimplementiert werden. Irgendwie muss allerdings gewährleistet werden, dass nicht *alle* Prozesse, sondern nur der mit der gegebenen Prozess-ID ausgeführt wird, und dass dieser die korrekte Eingabe bekommt. Daraus ergeben sich im Wesentlichen folgende drei Möglichkeiten der Implementierung:

Reimplementierung von `ProcessContainer` Die Klasse `ProcessContainer` ist bei `streams` dafür zuständig, die im XML-Dokument spezifizierten Prozesse in einer Liste zu sammeln und ihre Ausführung anzustoßen. Eine Möglichkeit wäre gewesen, diese so zu reimplementieren, dass sie statt ihres aktuellen Verhaltens nur einen Prozess ausführen. Aufgrund des großen Umfangs der Klasse und der Menge an Code, die schlicht hätte kopiert werden müssen, erschien diese Option nicht ratsam.

Manipulation des XML-Dokuments Eine weitere Option wäre gewesen, das XML-Dokument so zu manipulieren, dass alle Prozesse außer dem gewünschten gelöscht werden und dessen Input entsprechend umgeleitet wird. Es ist allerdings schwierig, sicherzustellen, dass dieser Ansatz für beliebige legale XML-Eingaben korrekt arbeitet. Außerdem ist er unflexibel, für den Fall, dass sich die XML-Spezifikation einmal ändern sollte. Ein weiteres Problem besteht darin, dass für jeden Knoten ein eigenes XML-Dokument erstellt werden muss, was Overhead verursacht.

Manipulation der vom ProcessContainer erstellten Objekte Wir haben uns daher dafür entschieden, den regulären ProcessContainer auf dem gegebenen XML-Dokument zu initialisieren. Damit werden für die entsprechenden Tags Prozess-, Stream-, Prozessor-Objekte usw. erzeugt. Über unseren ElementHandler sorgen wir dafür, dass der verteilte Prozess in jedem Worker wie ein regulärer Prozess behandelt wird. Dessen Eingabe wird aber von dem im XML-Dokument spezifizierten Multistream auf die jeweilig gewünschten Substreams geändert. Es werden damit die nicht gewünschten Substreams ignoriert. Weitere Prozesse im XML, die nicht der aktuellen Lastverteilung unterliegen, werden aus dem ProcessContainer entfernt. Diese Manipulationen sorgen dafür, dass die gewünschte Semantik durch einen schlichten `execute()`-Aufruf beim ProcessContainer erreicht wird: Eine Instanz des verteilten Prozesses läuft auf jedem Worker und bearbeitet genau die jeweilige Partition der Daten.

Ein Problem dieser Lösung besteht in der Initialisierung der Streams. Bei der Verarbeitung großer Datenmengen kann es leicht passieren, dass das gleichzeitige Öffnen aller gegebenen Streams auf einer Maschine nicht möglich ist, etwa weil dadurch zu viel Arbeitsspeicher verbraucht wird. Weiterhin existiert eine Beschränkung der Anzahl Dateien, die gleichzeitig aus dem HDFS gelesen werden können. Aus diesem Grund haben wir den Zeitpunkt der Stream-Initialisierung so verändert, dass diese erst passiert, wenn auch wirklich von dem Stream gelesen werden soll. Das hat zur Folge, dass auf dem Driver gar keine Streams

initialisiert werden und auf den Arbeiterknoten jeweils nur die, die auch verarbeitet werden sollen.

9.6 Verteilte Streaming-Prozesse

Als Alternative zur Batch-Verarbeitung bietet unsere Erweiterung die Möglichkeit, Daten als kontinuierlichen Strom verteilt zu verarbeiten. Dieser Ansatz hat im Wesentlichen zwei Vorteile. Zum Einen erlaubt er die Analyse von Echtzeit-Daten, also von Daten, die erst während der Laufzeit des Prozesses verfügbar werden. So kann zum Beispiel ein IP-Port als Eingabestrom genutzt werden, sodass dort ankommende Daten direkt weiterverarbeitet werden. Streaming-Prozesse können also genutzt werden, um einen Speed-Layer umzusetzen (vgl. section 3.2).

Zum Anderen gewährleistet die kontinuierliche Verarbeitung, dass zu jedem Zeitpunkt nur ein kleiner Teil der Daten im System ist. Dies steht im Gegensatz zur Batch-Verarbeitung, bei der sämtliche Ergebnisse gleichzeitig an den Driver geschickt werden. Dadurch ist dort die maximale Datenmenge, die verarbeitet werden kann, durch die Größe des Arbeitsspeichers des Drivers limitiert. Mit unserer Streaming-Lösung hingegen werden kontinuierlich kleine Teile der Daten gelesen, verarbeitet und weggeschrieben, sodass es möglich ist, die Datenmenge und die Anzahl Worker ohne harte Limits zu skalieren. Es kann also sinnvoll sein, Streaming-Prozesse auch für Daten zu verwenden, die bereits vollständig vorliegen. Unsere Lösung basiert auf Spark-Streaming und wird im Folgenden im Detail erläutert.

9.6.1 Datenfluss

Wie bereits in subsection 3.2.3 vorgestellt, basiert Spark Streaming darauf, dass der Eingabestrom in eine Sequenz von Minibatches zerstückelt wird, die dann durch herkömmliche Spark-Transformationen verarbeitet werden können. Es erscheint daher naheliegend, Streaming-Prozesse wie in Figure 9.2 dargestellt zu implementieren: Der ankommende Datenstrom wird von einem Receiver entgegengenommen, der nichts weiter tut als ankommende Daten zu speichern und in einem regelmäßigen Intervall als Minibatch-RDD weiterzugeben. Gegebenenfalls können auch mehrere Receiver verwendet werden, die reihum Daten aus dem Datenstrom nehmen und jeweils eine RDD-Sequenz erzeugen. Die so entstehenden RDDs können dann wie beim Batch-Prozess verarbeitet werden, indem in einer Spark-Transformation das `streams`-Framework instantiiert wird, um den spezifizierten Prozess auszuführen. Anschließend können die Ergebnis-RDDs zunächst durch die `union`-Operation zusammengeführt und dann mittels `collect` im Driver gesammelt werden.

Leider hat diese Lösung einige schwerwiegende Probleme. Das größte besteht darin, dass das System zusammenbricht, wenn über längere Zeit Daten schneller eingelesen als ver-

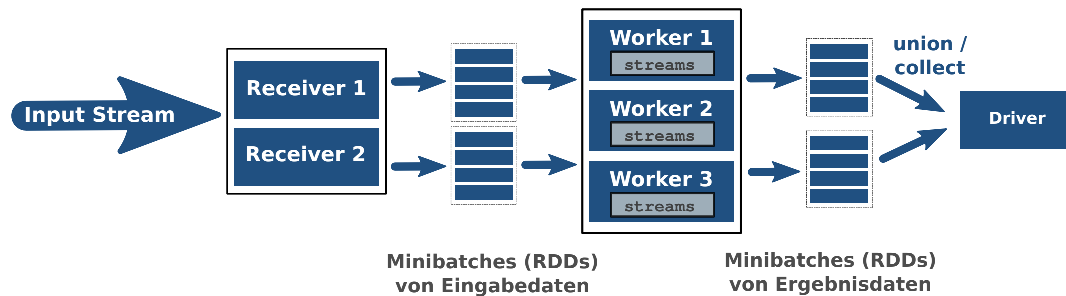


Abbildung 9.2: Datenfluss nach dem Konzept von Spark Streaming

arbeitet werden. Für den Fall, dass Echtzeitdaten verarbeitet werden, ist das schwer zu ändern - schließlich macht die Echtzeit-Analyse nur dann Sinn, wenn die ankommenden Daten auch in Echtzeit verarbeitet werden können. Eine der Motivationen für Streaming-Prozesse war ja aber, damit auch Daten verarbeiten zu können, die bereits vorliegen (z. B. im HDFS), um eine bessere Skalierbarkeit zu ermöglichen. In diesem Fall pumpen die Receiver ungebremst Daten ins System, was früher oder später den Arbeitsspeicher der Knoten füllt und das System zum Erliegen bringt. Der einzige Weg, das System auf diese Weise zu betreiben, ist, aufwändig auszutesten, mit welcher Rate Daten verarbeitet werden können und die Leserate der Receiver künstlich auf einen geringeren Wert zu limitieren. Daher ist dieser Ansatz zur Analyse von Bestandsdaten kaum praktikabel.

Ein weiteres Problem dieser Lösung besteht darin, dass die Eingabedaten, nachdem sie von den Receivern verarbeitet wurden, noch einmal durch das Netzwerk geschickt werden müssen, bevor sie verarbeitet werden. Das verschwendet unnötig Ressourcen, und fällt bei unserem Anwendungsfall besonders ins Gewicht, da hier die Eingabedaten um Größenordnungen größer als die Ausgabedaten sind. Zusätzlich hat diese Lösung das praktischen Problem, dass für jede RDD das `streams`-Framework neu initialisiert werden muss, da jeweils ein neuer Spark-Task erzeugt wird. Dies sorgt für einigen Overhead und führt zu einer unteren Schranke für das Batch-Intervall.

Aus diesen Gründen haben wir uns entschieden, von dieser Architektur Abstand zu nehmen. Stattdessen führen wir die Verarbeitung der Daten bereits im Receiver durch, wie in Figure 9.3 dargestellt. Dies eliminiert die oben genannten Probleme: Dank der Pull-Semantik des `streams`-Frameworks werden Daten nur so schnell eingelesen wie sie auch verarbeitet werden, die Eingabedaten müssen nicht nochmal verschickt werden und das `streams`-Framework kann dauerhaft initialisiert bleiben. Möglich wird diese Lösung dadurch, dass das `streams`-Framework bereits von sich aus für die Verarbeitung von kontinuierlichen Datenströmen ausgelegt ist. Es kann sehr natürlich in den Receivern instantiiert werden und die vorherige Stückelung in Minibatches ist unnötig.

Dadurch, dass auf diese Weise fast die ganze Arbeit in den Receivern erledigt wird, sollte

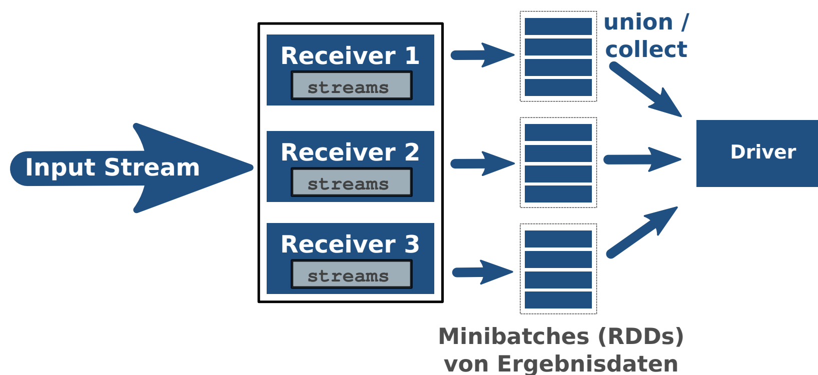


Abbildung 9.3: Datenfluss von unserer Streaming-Lösung

auch ein Großteil der verfügbaren Ressourcen für die Receiver alloziert werden. Es wird allerdings weiterhin eine geringe Zahl Worker benötigt, um die ankommenden RDDs zusammenzuführen. Dadurch, dass diese Operationen wenig Overhead pro RDD verursachen, kann das Batch-Intervall klein gehalten werden. Wir haben daher ein Intervall von 500ms gewählt. Die Implikationen für die Performanz werden in section 13.2 analysiert.

9.6.2 Arbeitsweise der Receiver

Zentral zur Umsetzung dieser Lösung ist es, einen benutzerdefinierten Receiver zu implementieren, der einen `streams`-Prozess ausführt. Hierzu wird bei Initialisierung des Receivers das `streams`-Framework instantiiert, genau wie bereits in subsection 9.5.2 beschrieben. Anders als bei den Batch-Prozessen müssen die Ergebnisse des Prozesses allerdings in die Ausgabe des Receivers umgeleitet werden. Dort werden sie dann von Spark automatisch in RDDs verpackt. Hierzu verwenden wir eine Senke, die Daten automatisch entsprechend weiterleitet, wenn sie vom Prozess ausgegeben werden.

Etwas problematisch ist, zu entscheiden, wann der Streaming-Prozess beendet ist. Spark Streaming ist zur Verarbeitung von Echtzeitdaten ausgelegt, die prinzipiell endlos in das System hineinströmen. Sollen jedoch statische Daten verarbeitet werden, ist es wünschenswert, dass die Analyse stoppt, wenn alle Daten abgearbeitet sind. Dazu bieten wir die Möglichkeit, den Prozess zu beenden, sobald eine gewisse Anzahl der eingelesenen RDDs in Folge leer waren. Ist der Eingabestrom des Prozesses versiegt, wird das auf jeden Fall passieren und die Analyse wird wie gewünscht stoppen. Um zu verhindern, dass Schwankungen im Datendurchsatz zum unerwünschten Herunterfahren des Systems führen, bieten wir dem Benutzer die Möglichkeit, die maximale Anzahl leerer RDDs über den Parameter `maxEmptyRDDs` festzulegen.

Kapitel 10

Einbindung von Spark ML

Zur Arbeit unserer Software wird unter anderem die Gamma-Hadron-Separation und die Energieschätzung gehören. Beide Aufgaben beinhalten maschinelle Lernverfahren, sodass wir eine Möglichkeit finden mussten, die Spark-MLlib-Methoden in `streams` zur Verfügung zu stellen. Dazu gehören nicht nur die Klassifikation und Regression, sondern auch die Merkmalsextraktion, die Vorverarbeitung der Daten und die Evaluation der gewählten Lernverfahren. Momentan werden Vorverarbeitung und Merkmalsextraktion von dem im vorherigen section 9.5 vorgestellten *BatchProcess* durchgeführt. Mit unserer Erweiterung soll es jedoch auch möglich sein, MLlib-Methoden zu nutzen, wenn dies gewünscht ist.

Bevor das Design unserer Erweiterung erläutert wird, soll Spark MLlib noch einmal genauer beleuchtet werden. Die grundlegenden Konzepte wurden in Figure 3.1.2 beschrieben, nun soll näher betrachtet werden, welches Paket aus Spark MLlib für unsere Projektgruppe das bessere ist. Dabei soll im Folgenden genauer auf die Unterschiede zwischen den Paketen eingegangen werden.

10.1 Spark ML vs. MLlib

In einer zweiwöchigen Experimentierphase zu Beginn der Projektgruppe beschäftigten wir uns mit der Frage, welches Paket der Spark-MLlib-Bibliothek besser für unsere Zwecke geeignet sein würde, entweder die ältere Version MLlib oder die neuere ML, welche auch noch aktiv weiterentwickelt wird. Zunächst wählten wir einige Datensätze aus dem UC Irvine Machine Learning Repository [60] aus, anhand welcher die Modelle trainiert und evaluiert werden sollten. Diese Datensätze waren leicht zu beschaffen und sollten eine erste Basis für die Experimente darstellen. Im späteren Verlauf der Experimentierphase verwendeten wir außerdem einen Ausschnitt der Monte-Carlo-Simulationsdaten (siehe section 6.3), welche auch in der endgültigen Software den Trainingsdatensatz bilden werden. Einen guten Einstieg bildet der Spark Machine Learning Library Guide [6], welcher nicht nur jedes

einzelne Verfahren detailliert erklärt, sondern auch die Grundlagen der Spark MLlib Bibliothek darstellt und einige Beispielimplementierungen liefert. Dank dieser erzielten wir recht schnell Ergebnisse, stießen jedoch auch auf einige Probleme, die im Folgenden kurz geschildert werden sollen.

Zuerst informierten wir uns, welche Algorithmen von den einzelnen Paketen implementiert werden. Unsere Ergebnisse sind in der nachfolgenden Tabelle zu sehen und entsprechen dem Stand von Apache Spark 1.6.0 (4. Januar 2016):

	MLLib	ML
Feature Extraction, Transformation and Selection		✓
Lineare SVM	✓	
Entscheidungsbaum	✓	✓
RandomForest	✓	✓
GradientBoosted Trees	✓	✓
Logistische Regression	✓	✓
Naive Bayes	✓	
Methode kleinster Quadrate	✓	
Lasso Regression	✓	✓
Ridge Regression	✓	✓
Isotonic Regression	✓	
Neuronales Netzwerk		✓

Die von den Physikern bereits genutzten Entscheidungsbäume und Zufallswälder sind in beiden Paketen enthalten. Dennoch fällt in der Übersicht auf, dass ML einen entscheidenden Vorteil bietet, nämlich die Möglichkeiten zur Merkmalsselektion, -transformation und -extraktion. Dies ist für unseren Anwendungsfall wichtig, da eine Aufgabe unter anderem darin besteht, die für das Training und die Klassifikation besten Merkmale zu finden.

Bei der Implementierung war es zunächst problematisch, Datensätze einzulesen, welche nicht dem des MLLib-Paketes bevorzugten Einleseformat LIBSVM [22] entsprachen. Dementsprechend sollten die Daten wie folgt organisiert sein:

```
label feature1:value1 feature2:value2 ...
```

Die dem Repository entnommenen Datensätzen entsprachen leider nicht dem gewünschten Format, sodass wir Methoden schreiben mussten, die die von uns ausgewählten Dateien analysierten und in JavaRDDs konvertierten. Generell kann zwar jedes beliebige Dateiformat eingelesen werden, doch das Parsen muss bei Verwendung des Pakets MLLib selbst übernommen werden. Das Paket ML hingegen arbeitet auf Grundlage von DataFrames. Diese können unter anderem aus Datenbanken oder JSON-Dateien gelesen werden. Da uns das `streams`-Framework bereits die Möglichkeit zum JSON-Export bot, konnten wir

einfach einen Ausschnitt der Monte-Carlo-Simulationsdaten (siehe section 6.3) als JSON-Datei exportieren und in unseren Tests als DataFrame importieren. Dies ist ein entscheidender Vorteil des ML-Paketes.

Auf ein weiteres Problem stießen wir bei dem Versuch, ein Modell mit Daten zu trainieren, deren Attribute nicht ausschließlich numerischer Natur waren. Bei Nutzung des MLLib-Paketes gingen die Algorithmen von Daten in Form eines *LabeledPoint* aus. Dieser besteht aus einem numerischen Label und einem Vektor numerischer Features. Nutzt man die Methoden aus dem Paket ML, gibt es zwar beim Ablegen von nominalen Attributen in einem DataFrame keine Probleme, jedoch gibt es Klassifikationsalgorithmen, welche nur mit numerischen Merkmalen trainieren und klassifizieren können. Das Problem der Transformation blieb also bestehen. Das Paket MLLib bietet keine Möglichkeiten, um diese Transformation durchzuführen, bei ML fanden wir sehr schnell die benötigten Methoden.

Auch die Label unterliegen einer Einschränkung. Sie sollen beginnend von Null durchnummeriert werden, sollen also nicht nominal sein oder mit +1 und -1 gekennzeichnet sein, wie es bei binären Klassifikationen oft der Fall ist. Es stellte sich ebenfalls heraus, dass ML uns Arbeit durch Bereitstellung geeigneter Methoden abnehmen konnte, MLLib jedoch nicht.

Für unseren Anwendungsfall ist es wichtig, dass sich Modelle abspeichern, im HDFS hinterlegen und nach Belieben wieder laden lassen. Außerdem sollen gespeicherte Modelle gestreamt werden können. Das Paket ML bietet bereits einige Methoden, um Pipelines abzuspeichern. Dabei muss darauf geachtet werden, dass in der Pipeline ein Modell trainiert oder genutzt wird, für welches diese Speichermethoden bereits funktionieren. Generell scheint es jedoch kein Problem zu sein, Modelle abzulegen und wiederzuverwenden, was ein großer Vorteil des ML-Paketes ist.

Insgesamt stellte sich heraus, dass die Spark MLlib Bibliothek sehr konkrete Annahmen über Eingabeformate und die Formatierung der Daten macht. Nutzt man das Paket ML, treten dabei jedoch keine Nachteile auf. Wir wollen primär aus Datenbanken lesen oder die Trainingsdaten, welche als JSON-Datei vorliegen, importieren. Für die Vorbereitung und Formatierung der Daten für den Trainings- und Klassifikationsablauf stellt das Paket ML viele Methoden bereit. Es scheint nicht nur komfortabler, primär auf das Paket ML zu setzen, die Nutzung wird von Apache sogar ausdrücklich empfohlen, da das Paket MLLib gar nicht mehr weiterentwickelt wird. Obwohl es auch noch unterstützt wird, haben wir uns daher entschieden, auf die Pipeline-Struktur von ML aufzubauen und die in diesem Paket enthaltenen Methoden zur Vorverarbeitung und Klassifikation unserer Daten zu nutzen. Außerdem funktioniert das Speichern und Laden von Modellen, welche wir dann problemlos streamen können. Nachdem die Entscheidung für das ML-Paket gefallen ist, folgt nun die detaillierte Beschreibung unserer Einbindung.

10.2 XML-Spezifikation

Beim Design unserer Erweiterung stand vor allem im Fokus, dass das Spark-ML-Paket auf *DataFrames* arbeitet. Während in der Basisvariante des `streams`-Frameworks die zu verarbeitenden Daten in *Data*-Items gestreamt werden, mussten wir einen Weg finden, diese in *DataFrames* zu konvertieren oder die Daten direkt in *DataFrames* zu laden, damit diese dann an die Spark-MLlib-Methoden weitergegeben werden können. Außerdem spielt die Pipelinestruktur, welche im ML-Paket von Spark MLlib verwendet wird, eine zentrale Rolle in unserer Spezifikation. Sie ähnelt stark der Prozess-und-Prozessoren-Struktur des `streams`-Frameworks. Während Prozesse diverse Prozessoren enthalten können, durch die die Daten sequentiell durchgereicht werden, können die in Spark ML verwendeten Pipelines diverse Stages enthalten. Auch dort werden die Daten sequentiell von Stage zu Stage weitergereicht. Wir entschieden uns dieses Konzept in unsere XML-Spezifikation zu übernehmen, schließlich soll die Anwendung für die Physiker, welche bisher nur das `streams`-Framework kennen, einfach zu erlernen sein. Durch den ähnlichen Aufbau integriert sich unsere Erweiterung nicht nur optisch, sondern auch inhaltlich gut in das Framework. Die Spezifikation und die Implementation der neu eingeführten Tags soll in den folgenden Unterkapiteln näher erläutert werden.

XML-Spezifikation von input

Ein `input`-Tag dient dazu, eine Datenquelle zu spezifizieren, die einen DataFrame (siehe Figure 3.1.2) zurückgibt. Im Gegensatz zu einem `<stream>` müssen die Daten also nicht zeilenweise, sondern als ganze Tabelle zurückgegeben werden.

Als Datenquelle kann jede Unterklasse von `stream.io.DataFrameStream` verwendet werden. Jeder `input` muss ein Attribut `id` mit einem eindeutigen Wert besitzen. Ein `input`-Tag muss auf der obersten Ebene eines Containers stehen. Ein Beispiel hierfür findet sich in Listing 10.1.

XML-Spezifikation von task & operator

Das Task-Tag wird genutzt, um neue Arbeitsabläufe zu modellieren. Es befindet sich innerhalb des *container*-Tags, zusammen mit den *input*-Tags. Ein Task hat die Argumente *ID=...* und *input=...*. Letzteres erlaubt ihm, auf die vorher verwendeten *input*-Tags Bezug nehmen. Dann führt er den in ihm spezifizierten Arbeitsablauf auf den im Input angegeben Daten aus. Dazu kann der Nutzer innerhalb des Tasks eine Kombination der Tags *pipeline* und *operator* verwenden, um die Daten zu bearbeiten, Modelle zu lernen und anzuwenden, Ergebnisse anzuzeigen etc. Dafür muss jeder Operator eine Unterklasse

von `stream.runtime.AbstractOperator` angeben, die die Arbeitsschritte auf dem DataFrame enthält. Operatoren und Pipelines werden sequentiell ausgeführt und der jeweils resultierende DataFrame an den Nachfolger weitergereicht.

Interessant ist hierbei, dass Task bzw. Operator genau dem Prozess bzw. den Prozessoren von *Streams* entsprechen. Da wir allerdings die *SparkML*- bzw. *SparkMLib*-Bibliothek verwenden wollen, müssen wir, wie bereits erwähnt, die Daten in Form von *DataFrames* anstelle der von Streams verwendeten *Data*-Klasse speichern. *Task* und Operatoren tun genau dies, sie sind also äquivalent zu den jeweiligen Streams-Klassen, arbeiten aber auf einem anderen Typ von Daten. Dies ermöglicht es uns, die Algorithmen der Spark-Bibliothek zu verwenden, ohne dass sich an der Struktur des XMLs viel ändert.

Eine wirkliche Neuerung stellt also nur das Pipeline-Tag, mit dem Pipelines der *Spark* Bibliotheken verwendet werden können, dar. Es dient dazu, komplexere Abläufe in der Datenvorverarbeitung einmalig zu modellieren, die so modellierte Pipeline kann dann von den auf sie folgenden Operatoren verwendet werden.

```
1  <container>
2    <input id="1" class="someInput" />
3
4    <task id="2" input="1">
5      <pipeline modelName="model">
6        ...
7      </pipeline>
8
9      <operator class="ApplyModelOperator" modelName="model" />
10     <operator class="PrintDataFrameOperator" />
11   </task>
12 </container>
```

Listing 10.1: Ein Beispiel XML - Mehr Informationen zu den einzelnen Tags sind in den folgenden Abschnitten zu finden

XML-Spezifikation von pipeline

Wie bereits erwähnt, wurde das `<pipeline>`-Tag eingeführt, damit die von Spark ML bereitgestellt Pipeline-Struktur als XML-Format definiert werden kann. Dazu wird das Tag innerhalb eines Tasks definiert und kann dann durch Spezifizierens eines Namens im weiteren Verlauf verwendet werden (Listing 10.1).

```
1  <task ...>
2    <pipeline modelName="model">
3      <stage class="MyStage" />
4      <transformer ... />
5      <transformer ... />
6      <estimator ... />
7        ...
8    </pipeline>
9
10   <operator class="ExportModelOperator" exportURL="..."
         modelName="model" />
11 </task>
```

Listing 10.2: Beispiel-XML einer reduzierten Pipeline innerhalb einer Task

Listing 10.2 stellt beispielhaft dar, wie eine Pipeline innerhalb eines Task erstellt werden kann, um dann später im `ExportModelOperator` wieder abgerufen zu werden. Dazu muss lediglich der Name der zu exportierenden Pipeline im Parameter `modelName` angegeben werden. Durch die Einführung eines Namens wird es zeitgleich ermöglicht, mehrere definierte Pipelines innerhalb eines Task voneinander zu unterscheiden. Dabei sei allerdings anzumerken, dass eine Pipeline überschrieben wird, sollte derselbe Name später wieder verwendet werden.

Innerhalb einer Spark ML Pipeline existieren zwei unterschiedliche Komponenten: `Estimator` und `Transformer`, welche im Allgemeinen als Stages bezeichnet werden. Die Beschreibung ihrer XML-Spezifikation folgt im nächsten Unterabschnitt.

XML-Spezifikation von stages

Nachdem der *pipeline*-Tag genauer ausgeführt wurde, soll es nun um die *Estimator* und *Transformer* gehen, deren Überbegriff *Stage* ist. Sie bilden das Herzstück der Pipeline und legen fest, welche Arbeitsschritte in der Pipeline auf den Daten ausgeführt werden sollen.

Ein *Estimator* ist eine Klasse, welche einen *DataFrame* bekommt und basierend auf einem Lernalgorithmus ein Modell erzeugt. In Spark ML stehen dafür zahlreiche Klassifikations- und Regressionsmethoden, aber auch Methoden für die Mermalsextraktion und das Clustering zur Verfügung. Um diese Funktionalität nutzen zu können, spezifizierten wir einen *estimator*-Tag. Die gewünschte Klasse soll im Parameter *stage* angegeben werden, danach können beliebig viele Parameter für genau diese Klasse folgen.

```
1   <estimator stage="RandomForestRegressor" numTrees="20"
       labelCol="label" featuresCol="features" />
```

Listing 10.3: Beispiel-XML für die Verwendung des estimator-Tags

In Listing 10.3 wird beispielsweise ein *Estimator* der Klasse *RandomForestRegressor* erzeugt, wobei die Attribute *numTrees*, *labelCol* und *featuresCol* gesetzt werden.

Ein *Transformer* ist eine Klasse, welche einen *DataFrame* bekommt und verändert, meistens durch Anfügen einer neuen Spalte. Damit können Vorverarbeitungsschritte oder auch eine Klassifikation, also eine Anwendung eines erlernten Modells, gemeint sein. Analog zum *estimator*-Tag erstellten wir einen *transformer*-Tag, wobei im Parameter *stage* die gewünschte Klasse angegeben werden soll. Danach können wiederum beliebig viele Parameter folgen, um die gewünschten Attribute zu setzen.

```
1   <transformer stage="Binarizer" inputCol="Length" outputCol=
       "newLength" threshold="2" />
```

Listing 10.4: Beispiel-XML für die Verwendung des transformer-Tags

In Listing 10.4 wird beispielsweise ein *Transformer* der Klasse *Binarizer* erzeugt, wobei die Attribute *inputCol*, *outputCol* und *threshold* gesetzt werden.

Insgesamt kann man auf diese Weise alle von Spark ML bereitgestellten *Estimator* und *Transformer* in einer Pipeline instantiieren. Wichtig ist, dass diese beiden Tags nur innerhalb einer *pipeline*-Umgebung stehen, denn sie werden in Spark ML immer als Teil einer großen Pipeline ausgeführt. Die Reihenfolge der Ausführung wird mit der Reihenfolge der Tags im XML festgelegt und die Stages werden sequentiell durchlaufen. Außerdem können pro Pipeline mehrere Modelle trainiert werden. Es ist auch möglich, dass nach einem *estimator*-Tag wieder *transformer*-Tags folgen, beispielsweise um im weiteren Verlauf der Pipeline ein Modell auf Grundlage eines noch weiterverarbeiteten *DataFrames* zu trainieren. Weiterhin gibt es keine Limitierung für die Anzahl von Stages. Nachfolgend steht ein abschließendes Beispiel für den Aufbau einer Pipeline durch Transformer und Estimator:

```
1 <container>
2   <input id="1" class="stream.pg594.example.MCInput"/>
3 
4   <task id="2" input="1">
5     <pipeline modelName="RFRegressor">
```

```
6         <transformer stage="VectorAssembler" inputCols="Length
            ,Width,Delta,Distance,Alpha,Disp,Size" outputCol="
            features"/>
7          <!-- arbitrary transformers and estimators -->
8         <estimator stage="VectorIndexer" inputCol="features"
            outputCol="indexedFeatures" maxCategories="10"/>
9         <estimator stage="RandomForestRegressor" numTrees="20"
             labelCol="MCorsikaEvtHeaderfTotalEnergy"
            featuresCol="indexedFeatures"/>
10      </pipeline>
11    </task>
12  </container>
```

Listing 10.5: Beispiel-XML für die Verwendung der estimator- und transformer-Tags innerhalb einer Pipeline

10.3 Umsetzung

In diesem Abschnitt werden die Schritte der Umsetzung näher erläutert. Dazu werden die Klassen zur Instantiierung und Verarbeitung von Spark-ML-Aufgaben beleuchtet.

Implementierung von task & operator

Nun wird die Implementierung der gerade beschriebenen XML-Elemente skizziert. Dabei ist es das Ziel, die *streams*-Architektur zu erhalten und lediglich an einigen Stellen zu erweitern.

Das task-Element soll wie das process-Element auf der obersten Hierarchie-Ebene eines streams Container stehen. Deshalb muss zuerst ein `TaskElementHandler` beim XML-Parser registriert werden.

Aufgrund der syntaktischen Äquivalenz von task und process ist es möglich, den Code von process wiederzuverwenden. Hierzu müssen die task-Datentypen von den process-Datentypen erben. Auf diesem Weg entfällt das Problem, den streams Scheduler anzupassen, da die task-Blöcke von der streams Laufzeitumgebung automatisch wie process-Blöcke ausgeführt werden.

Das bedeutet aber auch, dass der Inhalt eines task-Blocks kompatibel zu den Inhalten eines process-Blocks sein muss. Dies wird erreicht, indem der Operator-Datentyp von dem

Processor-Datentyp erbt. Dazu muss jeder Operator eine Methode `Data process( Data input )` implementieren. Dies steht scheinbar im Widerspruch zum Konzept, dass jeder Operator einen DataFrame erhält, diesen bearbeitet und den veränderten DataFrame zurückgibt.

Diese beiden Anforderungen können zusammengeführt werden, indem das Bearbeiten des eigentlichen DataFrames in eine abstrakte Methode ausgelagert wird, die einen DataFrame erhält und den veränderten DataFrame wieder zurückgibt. Diese abstrakte Methode wird dann von jedem einzelnen Operator anwendungsspezifisch überschrieben. Hingegen wird die `Data process( Data input )` für alle Operatoren einheitlich implementiert. Sie liest den DataFrame aus dem gegebenen Data-Objekt aus, lässt ihn von der operatorspezifischen Methode bearbeiten und schreibt den veränderten DataFrame zurück in das Data-Objekt. Auf diesem Weg verhält sich ein Operator aus der Sicht von streams wie ein Processor, bietet aber dem Nutzer die neue Schnittstelle zur Bearbeitung von DataFrames an.

Implementierung von input

Das `<input>`-Tag wurde eingeführt, damit `DataFrame` Objekte in die bisherige `streams`-Architektur eingepflegt werden konnten. Dazu wurden zwei neue Klassen entwickelt: `DataFrameInput` und `DataFrameStream`. Abbildung 10.1 stellt die einzelnen Klassen dar, die bei der Verarbeitung von Input-Elementen beteiligt sind.

Zunächst ist anzumerken, dass für die Verarbeitung von `DataFrame`-Instanzen im `streams`-Framework die instantiierten Objekte an die zugehörigen Prozesse gesendet werden müssen. Nativ wird dies vom `streams`-Framework ermöglicht, sofern eine neue Klasse als Spezialisierung von `Source` definiert wird.

Aufgrund der Ähnlichkeit zu normalen Datenstreams wurde hier eine direkte Spezialisierung zur Klasse `AbstractStream` hergestellt. Jedoch sollte vermerkt werden, dass für eine bessere Abgrenzung von normalen Datenstreams eine Spezialisierung zur Schnittstelle `Stream` hergestellt werden sollte. Dies war jedoch für den ersten Prototypen keine Priorität.

Das Interface `DataFrameInput` wurde erstellt, damit eine bessere Abgrenzung von normalen Datenstreams ermöglicht wird. Der Vorteil einer solchen Schnittstelle findet sich schnell, wenn die Instanziierung der Klassen betrachtet wird. Derzeit wird noch der vom `streams`-Framework bereitgestellt `StreamElementHandler` genutzt, um Input-Elemente zu erstellen. Jedoch wäre es angebrachter, hier ein eigenständigen `InputElementHandler` zu implementieren, welcher nur Streams erzeugt die eine Spezialisierung von `DataFrameInput` darstellen, sodass eine bessere Abgrenzung zu dem bereits vorhandenen `<stream>` Tag ermöglicht wird.

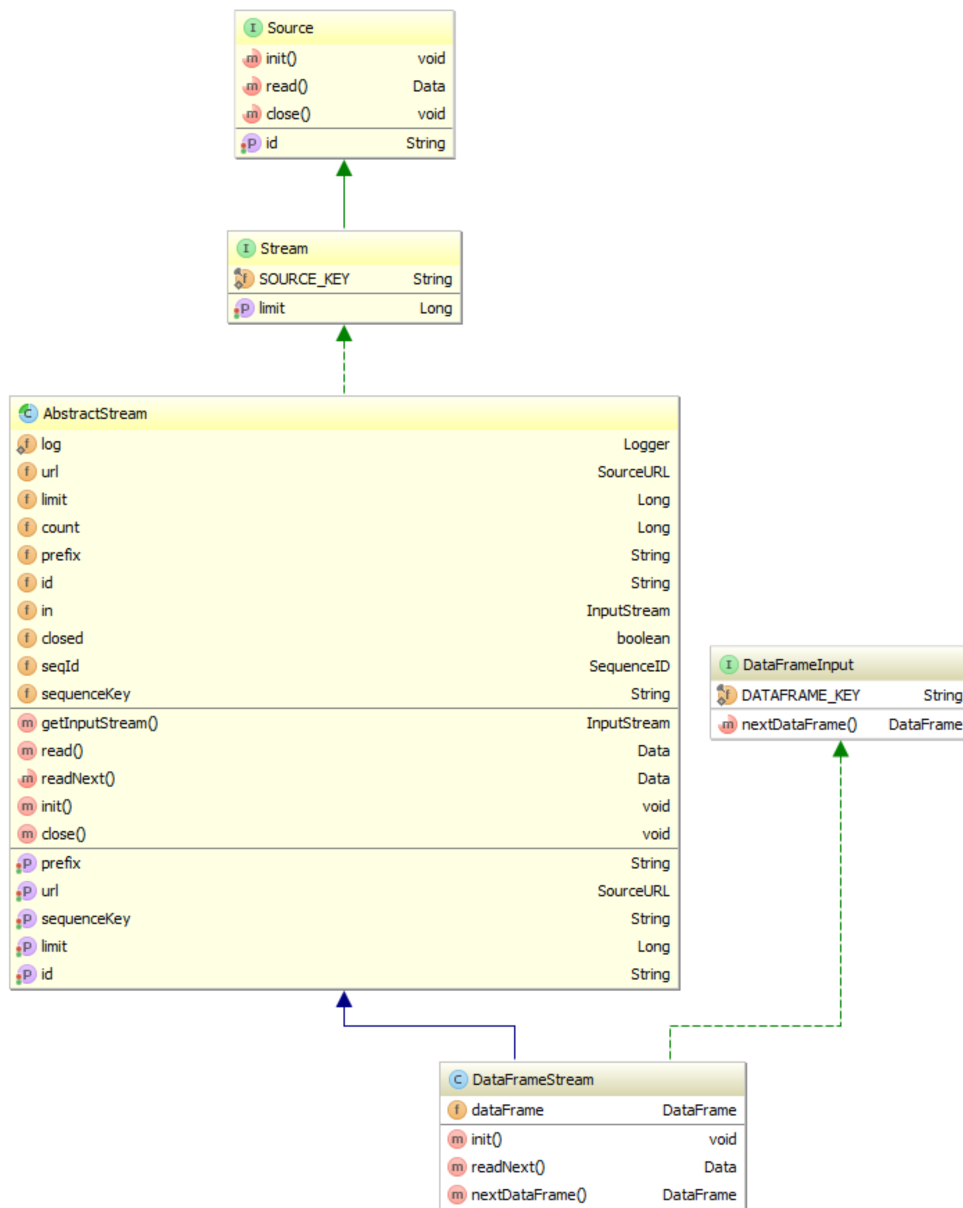


Abbildung 10.1: Klassendiagramm mit zugehörigen Klassen für `DataFrameInput` und `DataFrameStream`

Die Klasse DataFrameStream bietet die Möglichkeiten eines normalen Streams und erweitert diesen, um die von der DataFrameInput-Schnittstelle bereitgestellten Methode nextDataFrame(). Ziel dieser Methode ist es, dem DataFrameStream zu ermöglichen, eine Reihe von DataFrame Instanzen abzuarbeiten. Dazu muss zunächst ein EOF für einen Stream von DataFrames definiert werden, sodass beim Erreichen dessen der Stream endet. In der readNext() Methode wird dann jedes Mal nextDataFrame() aufgerufen und solange der Stream noch nicht den EOF Status erreicht hat, wird ein neues Data-Objekt erstellt, welchem das nächste Dataframe hinzugefügt wird. Auf diese Weise können Dataframes als Datastream im streams-Framework weitergeleitet und bearbeitet werden. Auch hier sei anzumerken, dass der derzeitige EOF Status noch nicht vollständig definiert und implementiert wurde, weshalb nur ein einziges DataFrame-Objekt in einem Input-Element erzeugt wird. Dies kann allerdings durch Implementieren von spezialisierten Klassen umgehen werden, indem die Methode nextDataFrame() überschrieben wird.

Implementierung von pipeline und stages

Mithilfe von einer <pipeline> können die aus SparkML bereitgestellten Pipelines genutzt werden. Damit diese Klassen im erweiterten streams-Framework abgerufen werden können, mussten Klassen zur Erstellung (Abb. 10.2) und Verarbeitung (Abb. 10.3) bereitgestellt werden.

Zur Erstellung von Spark ML Pipelines wurden im Wesentlichen zwei Factories implementiert. Die PipelineFactory erzeugt AbstractPipeline-Instanzen, für jedes spezifizierte <pipeline> Tag. Mittels der Methode createNestedStage() werden die definierten Stages erzeugt und der Pipeline zugewiesen. Hierbei wurden ein Ansatz über eine Erstellung über ObjectCreator gewählt. ObjectCreator sind Bestandteile der ObjectFactory, welche Teil des streams-Frameworks ist. Der ObjectFactory wird das zu erstellende XML-Element übergeben, welche dann innerhalb der Erstellung überprüft, ob ein ObjectCreator existiert, der dieses Element bearbeitet.

Abbildung 10.3 zeigt eine Übersicht der so erstellten Pipeline und Stage-Instanzen. Hierbei sei anzumerken, dass während der Entwicklung des Prototypen verschiedene Ansätze verfolgt wurden und sich diese Variante als intuitiv sinnvollste herausgestellt hat, da durch die Weiterverwendung der streams-Prozessoren wenig Änderungen an der Konstruktion dieser durchgeführt werden mussten.

Eine instantiierte Pipeline, wie beispielsweise eine DefaultPipeline, verarbeitet *DataFrame*-Objekte, weswegen sie als Spezialisierung des AbstractOperator implementiert wurde. Da Pipelines spezialisierte Prozessoren sind, kann die streams-Implementierung genutzt werden, um Daten zu verarbeiten und im Fall der Pipeline Dataframes. In der

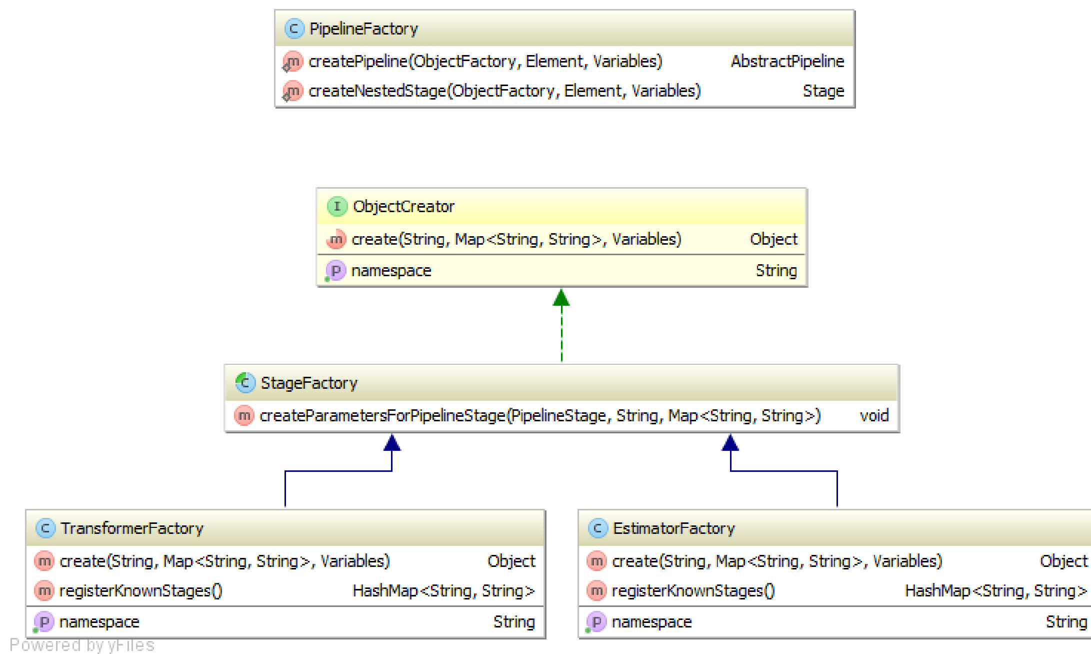


Abbildung 10.2: Übersicht der Klassen zuständig für die Erstellung von Pipelines und Stages

`DefaultPipeline` wird so für jeden Bearbeitungsphase ein neues Model trainiert und dem Datenstream übergeben. Damit die erstellten Modelle weiter genutzt werden können, muss jeder Pipeline über den Parameter `modelName` ein Name zugewiesen werden, womit die weitergegebenen Modelle identifiziert werden.

Pipelinestages werden über eine extra Ebene abstrahiert, um die aus `streams` bekannte Struktur, d.h. Prozesse (Pipelines) besitzen Prozesseoren (Stages), beizubehalten. Da Pipelines bereits als Prozessoren instanziiert werden, musste eine zustäzliche Ebene erstellt werden, um Stages einzubinden. Zugleich können diese Klassen genutzt werden, um unter anderen von Spark unabhängige Pipelinestages zu verfassen.

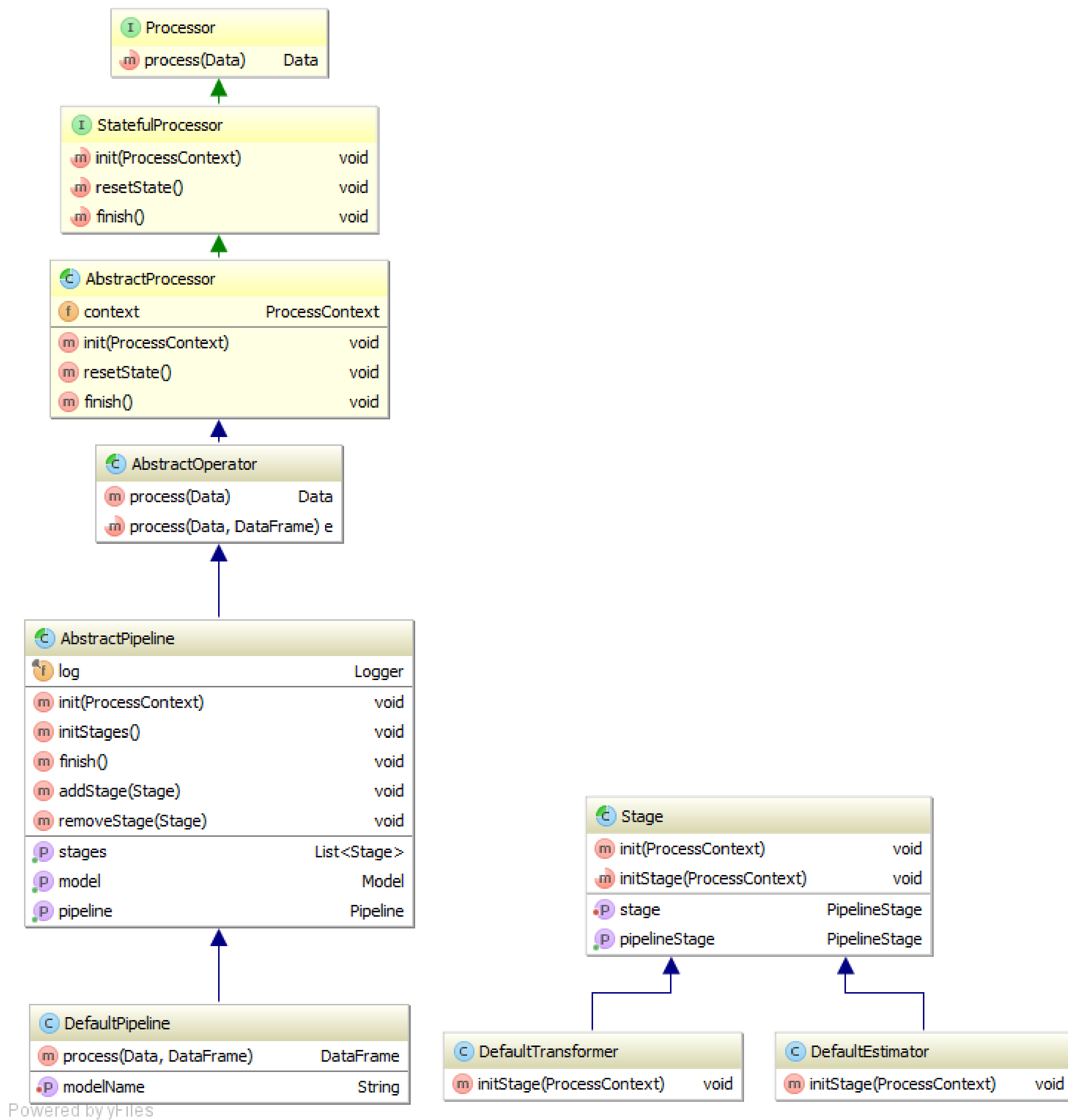


Abbildung 10.3: Übersicht der Klassen zuständig für die Verarbeitung von Pipelines und Stages

Kapitel 11

Verteilte Ein- und Ausgabe

Zur verteilten Ausführung von Prozessen ist es nötig, die zu verarbeitenden Daten aufzuteilen, sodass jeder Worker eine Teilmenge der Gesamtdaten verarbeiten kann. Nach Abschluss der Datenverarbeitung müssen die einzelnen Ergebnisdaten wiederum zu einer Gesamtheit zusammengeführt werden. Zudem soll die Möglichkeit bestehen, Ergebnisse (verteilt) im CSV-Format zu persistieren, um dieses Output als Basis für weitere Arbeitsschritte zu nutzen. Im Folgenden werden die zu diesem Zwecke erarbeiteten Lösungen der Projektgruppe vorgestellt.

11.1 MultiStream-Generatoren

Die verteilte Ausführung der Prozesse erfordert die Verfügbarkeit von mehreren Datenströmen. Damit können die Prozesse mehrere Datenströme gleichzeitig verteilt verarbeiten. Dafür wurde der `MultiStreamGenerator` implementiert.

Der `MultistreamGenerator` ist eine Erweiterung des `Streams-Framework MultiStreams`. Er wird zum Erzeugen von Datenströmen für eine verteilte Verarbeitung verwendet. Außerdem ist er auch in der Lage, mehrere Mengen von Datenströmen zu erzeugen. Da der `MultiStreamGenerator` eine Erweiterung der Klasse `SequentielMultiStream` des Streams-Frameworks ist, ist man so in der Lage, zwischen einer lokalen (nicht verteilten) und verteilten Datenstromverarbeitung zu unterscheiden.

Durch den `MultiStreamGenerator` ist es möglich, verschiedene Datenstromgeneratoren zu implementieren, zum Beispiel wurde im Rahmen der PG ein `FitsStreamGenerator` verwendet, der aus fits-Dateien Datenströme generieren kann. Aus einer Ordnerstruktur, die aus vielen Dateien besteht, werden verschiedene Datenströme erzeugt. Der Vorteil ist, dass es genügt, den Pfad des Oberordners anzugeben. Außerdem kann man durch die Eingabe regulärer Ausdrücke nicht erwünschte Dateien filtern. Will man für Testzwecke

oder aufgrund von Speichermangel die Anzahl der generierten Datenströme begrenzen, erlaubt der `FitsStreamGenerator` dies durch das Setzen der Parameter `streamLimits` und `maxNumStreams`. `streamLimits` definiert die Länge der einzelnen Datenströme. `maxNumStreams` setzt die Anzahl der generierten Datenströme fest.

Durch Erweiterung von `MultiStream` durch den `MultiStreamGenerator` ist man also in der Lage, mit einer Zeile mehrere Datenstromquellen zu definieren (siehe Listing 11.1).

```
1 <application>
2         <stream id="fact" class="stream.io.multi.
             FitsStreamGenerator" url="${infile}"
3         regex=".*\.fits\.gz" maxNumStreams="1000" />
4 </application>
```

Listing 11.1: Beispiel Multistream Eingabe

11.2 REST-Stream

Das REST-Interface 8 stellt Schnittstellen zur Verfügung, um FITS Dateien ausfindig zu machen, die Beobachtungen umfassen, deren Metadaten bestimmten Kriterien 8.2.2 entsprechen. Um die gefilterten Dateien bzw. die Beobachtungen im Sinne der Prozesskette verarbeiten zu können, wurde der REST-Stream entwickelt und wird nun im Folgenden vorgestellt.

11.2.1 `RestFulStream`

Der Stream ist so aufgebaut, dass dieser zwei Parameter (`url`, `filter`) annimmt. Mittels `url` wird die Url zu der entsprechenden Schnittstelle des REST-Interfaces übergeben und über den Paramter `filter` können die Kriterien zur Filterung der einzelnen Metadaten bzw. Dateien als einfacher String festgelegt werden. In 11.2 wird die wesentliche Verwendung des Streams verdeutlicht. Der Stream ist für die Verbindung zur REST-Schnittstelle verantwortlich und liefert als Ergebnis die einzelnen Events zurück, deren Metadaten den Kriterien des Filters genügen. Diese Events können innerhalb des `process`-Tags mit Prozessoren des `streams`-Frameworks weiterverarbeitet werden. Die Events werden mit Hilfe des Paramteres `input` und dem Wert *events* referenziert.

```
1
```

```
2  <application>
3
4      <!-- Name of the stream and url to the input file -->
5      <stream id="events"
6              class="edu.udo.reststreams.stream.
                  RestfulEventStream"
7              url="http://ls8cb01.cs.uni-dortmund.de:6060/api/
                  events/"
8              filter="night.eq(20130801).and(eventNum.lt(10)).
                  and(eventNum.gt(0))" />
9
10     <process input="events">
11       <!--  Process filtered events applying processors -->
12     </process>
13
14 </application>
```

Listing 11.2: Beispiel RestFullStream Eingabe

11.2.2 RestFulMultiStream

Die aus der REST-API zu Verfügung gestellten Daten, können als `Mulltistrem` mit dem `RestFulEventMultiStream` für den Input geliefert werden. `RestFulEventMultiStream` ist eine Erweiterung der Klasse `SequentiellMultiStream`. Aus den von der REST-API gelieferten JSON-Dateien werden die URLs der gewünschten Daten gelesen. Anhand dieser URL werden die einzelnen `FitsStream` erstellt und der Multistream angebunden. Ein Beispiel für die Verwendung eines `RestFulEventMultiStream` ist in Listing 11.3 zu finden.

```
1  <application>
2
3      <!-- Name of the stream and url to the input file -->
4      <stream id="events"
5              class="stream.io.multi.RestfullEventMultiStream"
                  streamLimits="5" maxNumStreams="3"
6              url="http://ls8cb01.cs.uni-dortmund.de:6060/api/
                  events/"
7              filter="night.eq(20130801).and(eventNum.lt(10)).
                  and(eventNum.gt(0))" />
```

```
8
9
10      <distributedProcess  id="DO" input="events"  >
11
12                <!-- Do something  -->
13
14      </distributedProcess >
15
16  </application>
```

Listing 11.3: Beispiel RestFullMultiStream Eingabe

Man kann wie auch beim einzelnen Stream einen Filter verwenden. Außerdem ist es möglich, die Anzahl der gelieferten Datenströme sowie die Anzahl der gelieferten Events per Datenstrom einzustellen. In dem Beispiel werden drei Datenströme geliefert mit jeweils fünf Events.

Mit dem `RestFulEventMultiStream` ist eine effiziente verteilte Verarbeitung der Daten, die auch aus der REST-API geliefert werden, möglich.

11.3 Verteilte CSV-Ausgabe

Um ein Modell trainieren zu können, müssen die Daten davor entsprechend vorbereitet werden. Man muss in der Lage sein, aufbereitete Daten exportieren können, um sie später für ein Modelltraining zu verwenden oder um einfach die Daten nach einer Feature-Extraction anzusehen. Aus diesem Grund ermöglicht unser Framework einen Export der von den Workern verarbeiteten Daten sowie von den Dataframes als `CSV-Dateien`

Export der Daten

Bei der Aufbereitungsphase werden entweder die Rohdaten anhand des `FitsstreamGenarator` gelesen oder über den `REST-API` geliefert. Anhand des `DistributedCsvWriter` können die Daten, die von den einzelnen Workern verarbeitet wurden, direkt nach der Aufbereitungsphase in `CSV-Dateien` geschrieben werden. Die einzelnen Dateien bekommen als Namen die IDs der einzelnen Worker. Da ein interner Zugriff auf den IDs der Workern nicht möglich ist, werden die eingegebenen Daten mit der Worker-ID versehen. Der Parameter `WorkerIdKex` in dem Input-Tag bekommt das Attribut `@worker` zugewiesen. In diesem Attribut wird die ID der für die Verarbeitung dieser Daten zuständigen Worker gespeichert und von dem `DistributedCsvWriter` verwendet. Außerdem muss man einen Link eines Ordners eingeben, in dem die Dateien gespeichert werden sollen. Ein

Beispiel ist in Listing 11.4 der XML zu sehen. Dabei werden die Daten mit Hilfe des `RestfulEventMultiStream` aus der MongoDB gelesen und von den einzelnen Workern in die CSV-Dateien geschrieben.

```
1  <application>
2
3      <!-- Name of the stream and url to the input file -->
4      <stream id="events"
5              class="stream.io.multi.RestfulEventMultiStream"
                  streamLimits="3" maxNumStreams="3"
6              url="http://ls8cb01.cs.uni-dortmund.de:6060/api/
                  events/"
7              filter="night.eq(20130801).and(eventNum.lt(10)).
                  and(eventNum.gt(0))" />
8
9
10     <BatchProcess  id="DO" input="events"  workerIdKey="
          @worker"  maxEmptyRDDs="0">
11
12                 <stream.io.DistributedCsvWriter url="hdfs://
                      ls8cb01.cs.uni-dortmund.de:9000/user/

                      hadoop/CsvWriterTest/" workerIdKey="
                      @worker" />
13     </BatchProcess >
14
15 </application>
```

Listing 11.4: Beispiel von der Anwendung eines DistributedCsvWriter

Export von `Dataframe`

Man kann nicht nur die extern gelesenen Daten als CSV exportieren sondern auch die schon in Dataframes gespeicherten Daten. Dafür ist der Operator `ExportDataframe` zuständig. Für den Export muss einen CSV-Ordner eingegeben, in den die einzelnen Dateien geschrieben werden sollen. Man ist auch in der Lage, die Anzahl der ausgegebenen Dateien zu definieren. In dem Parameter `numFiles` wird die Anzahl der ausgegeben Dateien eingegeben. In Listing 11.5 ist eine Beispiel-XML zu sehen, bei der ein Operator ein Dataframe in zehn Dateien schreiben soll.

```
1 <application>
2
3         <queue class="stream.io.RddQueue" id="queue"/>
4
5
6     <task id="Export" input="queue" persistDataFrameIn="
         MEMORY_ONLY">
7         <stream.pg594.operators.ExportDataFrame url="hdfs://
             ls8cb01.cs.uni-dortmund.de:9000/demo/  features.
             csv" numFiles="10"/>
8
9
10    </task>
11
12 </application>
```

Listing 11.5: Beispiel von der Anwendung eines DistributedCsvWriter

Mit der verteilten Ausgabe ermöglicht das Framework ein verteiltes Wiedereinlesen der Daten. Das kann von Vorteil sein, wenn man schon verarbeitete Daten oder vorhergesagte Daten wieder braucht.

Kapitel 12

Organisation

Das umzusetzende Projekt der Big-Data-Analyse auf FACT-Teleskopdaten besitzt eine Laufzeit von zwei Semestern und wird durch uns, ein Team aus 12 Studentinnen und Studenten, umgesetzt. Damit besitzt das Projekt unter Organisations-Aspekten eine gewisse Komplexität: Wie lässt sich die Arbeit sinnvoll zergliedern? Wie gehen wir mit Abhängigkeiten zwischen den Arbeitspaketen um? Wie strukturieren wir die Arbeit so, dass wir unsere Ziele bestmöglich umsetzen können?

Damit die Beantwortung solcher Fragen nicht zum Problem wird, ist es wichtig, sich bereits im Vorhinein auf Methoden zu einigen, die sinnvolle Antworten festlegen. Vorgehensmodelle und andere Projektmanagement-Praktiken geben Teams solche Methoden an die Hand.

Wir haben zu Beginn der PG eine kleine Auswahl agiler Verfahren kennengelernt, die wir in section 12.1 einführen wollen. Unsere konkrete Umsetzung dieser Verfahren wird in section 12.2 vorgestellt. Eine Retrospektive dieser Umsetzung findet sich übrigens in section 15.3.

12.1 Agiles Projektmanagement

Agile Projektmanagement-Verfahren können den Arbeitsablauf optimieren, indem sie einige der Probleme klassischer (also nicht-agiler bzw statischer) Verfahren vermeiden. Wir diskutieren hier zunächst einige dieser Probleme (siehe subsection 12.1.1), und wie das agile Manifest sie adressiert (siehe subsection 12.1.2). Als kleine Auswahl agiler Verfahren stellen wir Scrum und Kanban vor (siehe subsection 12.1.3 und subsection 12.1.4).

12.1.1 Probleme Nicht-Agiler Verfahren

Klassische Verfahren reagieren in der Regel nur unzureichend auf Änderungen in Anforderungen und Terminen, da die zugrundeliegenden Pläne für den gesamten Entwicklungsprozess erstellt werden. Da klassische Verfahren Planänderungen nicht im Entwicklungsprozess vorsehen (oder für sie ein bürokratisch aufwändiges Teilverfahren definieren), wird die Notwendigkeit solcher Änderungen gerne verkannt.

Häufig stellen sich die zu Beginn des Projektes erstellten Pläne als nicht-optimal heraus, weil sie später erworbene Informationen oder Änderungsbedarf nicht vorhersehen konnten. Daher eignen sich klassische Verfahren insbesondere nicht, um Projekte zu managen, deren Anforderungen zu Beginn unklar sind. Leider lässt sich die Klarheit der Anforderungen nicht immer sofort entscheiden.

Ein weiteres Problem ist, dass die in klassischen Verfahren geforderte Vielfalt an Dokumenten oft nur pro forma erstellt wird. So gibt es Dokumente, die nur beinhalten, was ohnehin bereits abgestimmt wurde, oder die zu einem Zeitpunkt gefordert waren, an denen noch keine ideale Lösung zu finden war. Solche Dokumente werden möglicherweise nie gelesen oder veralten, bevor sie einen Nutzen darstellen konnten.

Prominente Vertreter klassischer Projektmanagement-Verfahren sind das Wasserfallmodell, sowie die Modelle V und VXT. Sie alle basieren auf dem Prinzip, zunächst alle Anforderungen festzulegen, basierend darauf Entwürfe zu erstellen, und zuletzt Implementierungsarbeiten aufzunehmen. Im Wasserfallmodell werden Tests erst am Ende durchgeführt, was im V-Modell durch Testen auf jeder Entwicklungsstufe verbessert wurde. Das VXT-Modell erweitert V durch Ausschreibungen und Einbettung in übergeordnete Projekte. Durch diesen weiten Horizont entsteht aber ein enormer Umfang an Rollen und Artefakten, wodurch Projekte auch behindert werden können.

12.1.2 Das Agile Manifest

Das agile Manifest stellt die Grundprinzipien jedes agilen Projektmanagement-Verfahrens dar. Es korrigiert dabei die Annahmen klassischer Verfahren und leitet daraus explizite Regeln ab. Das agile Manifest lautet wie folgt [10]:

Reagieren auf Änderungen ist wichtiger, als einem Plan zu folgen. Pläne fokussieren die nahe Zukunft, da langfristige Planungen nur vorläufig sein können und möglicherweise notwendigen Änderungen unterliegen.

Funktionierende Software ist wichtiger als eine umfangreiche Dokumentation. Dokumentation sollte nicht pro forma erstellt werden, sondern einen Zweck erfüllen.

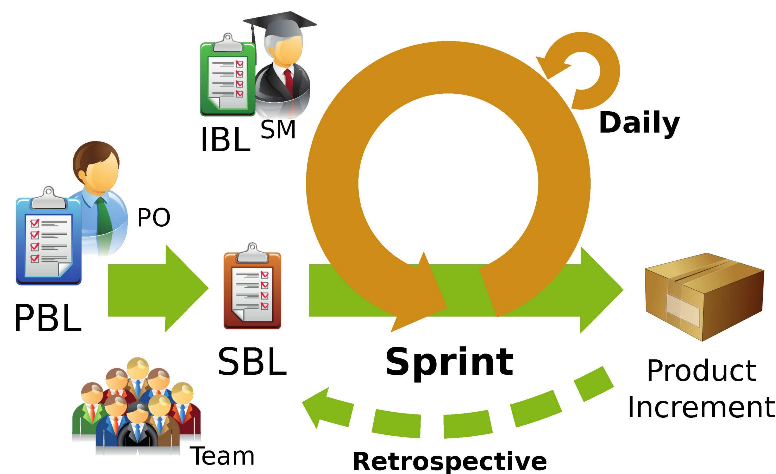


Abbildung 12.1: Der Sprint in Scrum

Individuen und Interaktionen ist ein höherer Stellenwert einzuräumen als Prozessen und Tools. Unzureichende Interaktionen zwischen Projektbeteiligten gefährden Projekte, egal, welche Prozesse verwendet werden.

Partizipation des Kunden bringt mehr als Vertrags-Verhandlungen. Eine enge Einbindung des Kunden macht Änderungsbedarf frühzeitig erkennbar und steigert damit den Nutzen des Produktes.

12.1.3 Scrum

Scrum [48] ist ein prominenter Vertreter agiler Projektmanagement-Verfahren. Zentral für Scrum ist der Sprint, ein kurzer Entwicklungszyklus (2 – 4 Wochen), welcher ein Produkt-Inkrement erzeugt. Ein solches Inkrement sollte einen Mehrwert für den Kunden darstellen. Während eines Sprints dürfen sich keine Änderungen der für den Sprint definierten Ziele ergeben, damit der Sprint geordnet abgearbeitet werden kann. Im schlimmsten Fall ist es möglich, einen Sprint vorzeitig abzubrechen und einen neuen Sprint aufzusetzen.

Figure 12.1 stellt einen Überblick über Scrum dar. Abgebildet sind die verschiedenen Rollen und Artefakte und ihre Einbettung in den Sprint. Zudem definiert Scrum einige Meetings. Alle diese Elemente werden im Folgenden vorgestellt.

Rollen

Der **Product Owner (PO)** stellt die Interessengruppen außerhalb des Teams dar. Insbesondere das Interesse des Kunden ist hier widergespiegelt, idealerweise aber auch andere, möglicherweise widersprüchliche Interessen. Der PO soll aus diesen Interessen die Vision

des Endproduktes formen und diese auf das Team übertragen. Dazu managt er mit dem Product Backlog eines der Artefakte.

Der **Scrum Master (SM)** coacht das Team in der Ausführung von Scrum, kann dazu die Moderation in den Meetings übernehmen und den PO in der Priorisierung des Product Backlog unterstützen. Außerdem löst er sämtliche Probleme (Impediments), die das Team von der Arbeit abhalten. Die Rolle des SM ist nicht gleichzusetzen mit einem Projektleiter mit Entscheidungsgewalt. Sämtliche Entscheidungen werden gemeinsam im Team getroffen.

Das **Team** übernimmt die Umsetzung eines Projektes. Dazu sollte es die Vision des Endproduktes verstehen. Es organisiert sich selbst, weshalb eine hohe Teilnahme der einzelnen Mitglieder gefordert ist. Die Möglichkeit, durch Selbstorganisation am Projekterfolg teilzuhaben, kann die Mitglieder motivieren und den Projekterfolg erhöhen. Idealerweise setzt sich das Team interdisziplinär aus $5 - 9$ Personen zusammen.

Artefakte

Das vom PO verwaltete **Product Backlog (PBL)** soll sämtliche gewünschte Features und Ergebnisse als User Stories vorhalten. Aufgrund sich ändernder Anforderungen ist das PBL aber jederzeit anpassbar.

User Stories erklären den Nutzen des jeweiligen Features für einen Endnutzer. Aufgrund dieses Nutzens lassen User Stories sich priorisieren. Außerdem lässt sich der Umfang jeden Features schätzen. Aufgrund von Umfang und Priorität lassen sich User Stories aus dem PBL auswählen, um im kommenden Sprint erledigt zu werden.

Für einen Sprint werden Teilaufgaben (Tasks) ausgewählter User Stories in den **Sprint Backlog (SBL)** übernommen. Für jeden Task ist eine Definition of Done (DoD) formuliert, die aussagt, wann der Task abgeschlossen ist. Das SBL stellt damit die Basis für die Organisation der Arbeit durch das Team dar. Es darf während eines Sprints nicht verändert werden.

Damit der SM die Behinderungen des Teams beseitigen kann, verwaltet er ein **Impediment Backlog (IBL)**, in welchem Teammitglieder Probleme einstellen und priorisieren können. Er kann diese Behinderungen selbst auflösen, oder deren Auflösung weiterdelegieren.

Meetings

Die verschiedenen Scrum-Meetings ermöglichen die Umsetzung des Verfahrens und eine Abschätzung des Projektfortschritts. Sie haben einen jeweils fest definierten Zweck, wodurch die Zeit, die für Meetings verwendet wird, reduziert werden soll.

Um einen kommenden Sprint zu planen, wird jeweils ein **Sprint Planning Meeting** abgehalten. Es beinhaltet die Schätzung (möglicherweise die Neu-Schätzung) der Items des PBL und eine Auswahl von Items für die Übernahme in den neuen Sprint. Die Auswahl wird auf Basis von Aufwand und Priorisierung der Elemente durch Konsens im Team getroffen. Darüber hinaus werden die Elemente des PBL in Tasks, wohldefinierte Arbeitspakete, zergliedert. Tasks werden Verantwortlichen zugewiesen und in das SBL eingetragen. Möglichst alle Termine für den kommenden Sprint werden festgelegt.

Um den Fortschritt des aktuellen Sprints festzustellen und Probleme (Impediments) zu identifizieren wird ein tägliches **Daily Meeting** oder kurz „Daily“, abgehalten. Es soll dort lediglich beantwortet werden, was zuletzt getan wurde und was als nächstes getan wird. Das Daily sollte eine Dauer von 15 Minuten nicht überschreiten.

Der Erfolg eines Sprints wird in einem **Review** und **Sprint Retrospective** ermittelt. Zum Review zählen die Vorstellung des Produkt-Inkrements sowie die Abnahme desselben durch den PO. In der Retrospektive wird die Qualität des Entwicklungsprozesses gemessen. Hier soll beantwortet werden, was gut und schlecht im letzten Sprint lief und wie möglicherweise Verbesserungen zu erreichen sind. Wie die Qualität gemessen werden soll, lässt Scrum offen. An dieser Stelle lässt sich Scrum hervorragend mit Kanban kombinieren, da Kanban die Messung der Prozessqualität stark fokussiert (siehe subsection 12.1.4).

12.1.4 Kanban

Kanban [47] ist, anders als Scrum, kein Vorgehensmodell. Es schreibt daher keinen Entwicklungsprozess vor, beinhaltet aber Praktiken, welche die Qualität bestehender Prozesse messen und verbessern können. Es wird ein Entwicklungsprozess angestrebt, der Inkremente regelmäßig, schnell und mit hoher Qualität ausliefern kann.

Das Verfahren modelliert dazu bestehende Prozesse als Kette von Arbeitsstationen, die jedes Produktinkrement durchlaufen muss (z.B. Analyse, Implementierung, Testing,. . .). Wichtig ist insbesondere, Abhängigkeiten innerhalb des Prozesses zu identifizieren, um Verzögerungen zu vermeiden. Dadurch lässt sich der Durchfluss optimieren, indem Bottlenecks identifiziert und aufgelöst werden.

Zentral für Kanban ist das Kanban-Board, auf dem der Prozess modelliert und sein Fortschritt sichtbar gemacht wird. Figure 12.2 zeigt einen Überblick über Kanban mit dem Board im Zentrum. Man erkennt die in Spalten angeordneten Stationen, sowie zusätzliche Spalten für Prozess-Input (in naher Zukunft geplante Features) und Prozess-Output (zur Abnahme freigegebene Features). Die Regeln von Kanban werden im Folgenden erläutert.

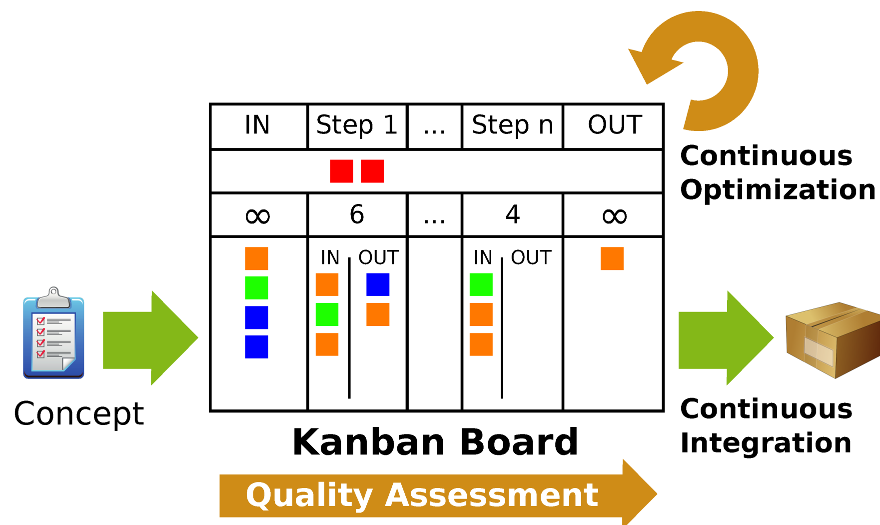


Abbildung 12.2: Das Kanban-Board

Regeln

Die grundlegende Regel in Kanban ist, dass die Anzahl Items in jeder Station, die **Work In Progress (WiP)**, streng limitiert ist. Die jeweiligen Obergrenzen sollten in jeder Spalte des Kanban-Boards eingetragen werden. Jede Station hat einen eigenen Input und Output. Im Input liegen aktuell bearbeitete Tickets, im Output fertige Tickets. Die Prozesskette funktionert nach dem Pull-Prinzip. Damit können Features nur weiterwandern, wenn die nachfolgende Arbeitsstation das Feature in seinen Input „zieht“.

Durch diese einfachen Regeln lassen sich Bottlenecks des Prozesses schnell identifizieren: Sollte ein Flaschenhals existieren, werden davor liegende Stationen aufgrund des Limits blockiert. Denn da die Station, die den Flaschenhals erzeugt, keine weiteren Tickets ziehen kann, dürfen auch frühere Stationen, wenn sie ihr Limit erreicht haben, keine weiteren Tickets annehmen. Dann kann die Ressourcenzuteilung zu den Stationen verbessert werden, sodass der Durchfluss steigt.

Damit die Anzahl der Tickets die tatsächliche Arbeit angemessen quantifiziert, sollten alle Tickets einen ähnlichen Arbeitsaufwand erzeugen. Dies kann z.B. durch Zergliederung von Features erreicht werden.

Optional können verschiedene Service-Klassen eingeführt werden, welche die Tickets priorisieren. Verbreitet ist z.B. eine Aufteilung in Standard, Expedite, Vague und Fixed. Expedite-Tickets wird eine eigene Bahn durch den Prozess zugeordnet, die nicht zu den Limits der Stationen zählt. So können z.B. wichtige Bugfixes vorrangig behandelt werden (siehe die roten Tickets in Figure 12.2). Vague-Tickets sollten nur durch die Kette wandern, wenn Kapazitäten des gesamten Prozesses frei sind. Fixed-Tickets können so durch den Prozess geführt werden, dass sie zu festen Terminen fertiggestellt sind.

Bewertung der Prozess-Qualität

Die Prozessqualität lässt sich zunächst daran messen, ob Bottlenecks in der Prozesskette existieren. Diese verringern den Durchfluss und weisen auf eine nicht-optimale Ressourcenverteilung hin. Wie bereits angemerkt, lassen sich Bottlenecks dadurch identifizieren, dass sie Tickets aufstauen und es dadurch vorigen Stationen nicht erlaubt ist, weitere Tickets anzunehmen.

Eine weitere Metrik zur Abschätzung der Qualität ist die Zeit, die für einzelne Tickets seit dem letzten Fortschritt vergangen ist. Solche Tickets sind möglicherweise blockiert, d.h., es sind Behinderungen aus dem Weg zu schaffen, damit das Ticket erfolgreich abgearbeitet werden kann. Weitere Metriken zur Messung des Durchflusses und dem Aufwand einzelner Tickets existieren darüber hinaus.

Wie Scrum verwendet auch Kanban Dailies und Reviews (siehe subsection 12.1.3), um den Projektfortschritt zu kommunizieren. Anders als in Scrum müssen Reviews aber nicht regelmäßig abgehalten werden.

12.2 Wahl des Verfahrens

Scrum und Kanban (siehe subsection 12.1.3 und subsection 12.1.4) stellen nur Rahmenwerke mit vielen Optionen zur Verfügung. Die Implementierung der Verfahren obliegt letztendlich dem Anwender. Für uns stellten sich folgende Fragen:

- Welches der Verfahren wählen wir? Nehmen wir eine Kombination vor?
- Wie lange sollen Sprints dauern?
- Wie sind die Rollen zu besetzen?
- Welche Software können wir für unser Verfahren verwenden?

Initial haben wir uns darauf geeinigt, lediglich Scrum zu verwenden und Kanban bei Bedarf zur Prozessbewertung und -optimierung hinzuzuziehen. Auf diese Weise wollten wir uns auf die Arbeit konzentrieren und die uns neuen agilen Projektmanagement-Verfahren nebenbei erlernen. Da Kanban kein Vorgehensmodell darstellt, sondern auf die Optimierung bestehender Prozesse abzielt, lässt sich ein solches Vorgehen gut implementieren.

Wir haben zwei Scrum-Master gewählt, um die Arbeit an den Impediments aufteilen zu können und bei Bedarf die Arbeit auf zwei Scrum-Teams aufzuteilen. Als Product Owner sollten die Betreuer herhalten. Sprints sollten zunächst eine Woche dauern, um dem hohen Abstimmungsaufwand am Anfang des Projektes zu begegnen, später sollten sie länger dauern.

Um das PBL zu pflegen, verwenden wir Atlassian JIRA. Über die Kommentar-Funktionen dieser Projektmanagement-Software für User Stories und Tasks können wir Lösungen diskutieren und unseren Fortschritt dokumentieren.

Teil III

Evaluation und Ausblick

Kapitel 13

Verteilte Streams-Prozesse

Die im Rahmen dieser PG umgesetzte Erweiterung des `streams`-Frameworks erlaubt die verteilte Ausführung von `streams`-Prozessen in einem Spark-Cluster (siehe chapter 9 bis section 9.6). Dabei ist es möglich, die Gesamtheit der Eingabedaten als Batch zu verarbeiten. Alternativ lassen sich Mini-Batches streamen, um kontinuierlich Ergebnisse zu erhalten.

Es soll hier zunächst konzeptionell evaluiert werden, welche Vorteile und Grenzen unsere Erweiterungen mit sich bringen (siehe section 13.1 und section 13.2). Darüberhinaus sollen die Skalierungseigenschaften des Systems beleuchtet werden, indem wir die Ausführungszeiten für verschiedene Konfigurationen des Clusters heranziehen (siehe section 13.3).

13.1 Batch-Prozesse

In einem Batch-Prozess wird für jeden zu verarbeitenden Datenstrom ein Spark-Task erstellt. Spark übernimmt das Scheduling solcher Tasks auf die verfügbaren Cores des Clusters. Sobald ein Task (beziehungsweise Datenstrom) abgearbeitet ist, wird dem freiwerdenden Core ein neuer Task zugeteilt (vgl. section 9.5).

13.1.1 Rechenleistung

Durch Sparks Task-Scheduling sind die Cores des Clusters sehr gut ausgelastet. Werden mehr Cores zur Verfügung gestellt, können auf sie ebenfalls Datenströme geschedulet werden und die Analysen werden früher fertig gestellt. Das System lässt sich also bezüglich der Rechenleistung, also der Anzahl und Geschwindigkeit der CPU-Kerne, horizontal skalieren - eine Schlüsseleigenschaft von Big Data-Systemen (vgl. section 2.3).

Allerdings werden natürlich nur maximal so viele Cores verwendet, wie Eingabe-Datenströme spezifiziert wurden. Die Anzahl der Eingabeströme stellt also eine Grenze für die

horizontale Skalierbarkeit dar. Da für Big Data ein hohes Datenvolumen angenommen wird (siehe chapter 2), besteht diese Grenze für praktische Anwendungen aber möglicherweise gar nicht. Im Zweifelsfall ist es möglich, durch eine Vorverarbeitung Daten auf mehrere Ströme aufzuteilen. Dadurch ist prinzipiell eine beliebige horizontale Skalierbarkeit der Batch-Prozesse möglich.

Problematisch ist auch, wenn die Größe der Eingabeströme stark variiert. Da Spark die Größe der Ströme nicht kennt, kann es die Größe beim Scheduling nicht heranziehen. Es kann dann passieren, dass zum Ende einer Batch-Analyse nicht mehr alle Kerne ausgelastet sind, weil manche Kerne ihre kurzen Datenströme abgearbeitet haben und andere mit längeren Datenströmen noch arbeiten müssen. Durch eine geeignete Vorverarbeitung können Eingabe-Datenströme mit annähernd gleicher Größe erzeugt werden.

13.1.2 Arbeitsspeicher

Eine schwerwiegendere Grenze der horizontalen Skalierbarkeit unseres Systems findet sich im Arbeitsspeicher des Driver-Knotens: Da sämtliche Ergebnisse als Batch, also zur gleichen Zeit gesammelt werden, kann der Speicher des Drivers beim Sammeln der Daten überlaufen. Dies ist der Fall, wenn das Ergebnis-Volumen die Kapazität des reservierten RAM überschreitet. Das Sammeln der Daten im Driver geschieht, damit sämtliche bestehenden Senken des `streams`-Frameworks wiederverwendet werden können.

Ist ein Speicherüberlauf zu befürchten, sollte man statt einer zentralen Senke zur Ausgabe Prozessoren verwenden, welche eine äquivalente Aufgabe übernehmen. Prozessoren werden in jedem Knoten erzeugt und erlauben somit eine verteilte Ausgabe der Ergebnisse. Beispielsweise können mit Prozessoren mehrere Dateien verteilt ins HDFS geschrieben werden, statt durch eine Senke zentral eine einzelne große Datei zu erzeugen. Ist es möglich, eine Analysekette auf diese Weise zu spezifizieren, müssen keine Daten im Driver gesammelt werden und ein Speicherüberlauf ist nicht zu befürchten. In diesem Falle skaliert der Arbeitsspeicher des Systems problemlos horizontal.

13.1.3 Fehlertoleranz und Generalisierbarkeit

Da Spark verfehlte Tasks (beispielsweise bei einem Hardware-Ausfall) neu schedulet, ist der Fehlertoleranz-Anforderung (siehe section 2.3) in diesem Sinne beigekommen. Weiterhin ist unsere Erweiterung generalisierbar, da sämtliche *streams*-Prozesse als Batch ausgeführt werden können.

13.2 Streaming-Prozesse

Streaming-Prozesse verarbeiten die Daten in Spark Receivern. Dadurch werden sämtliche Daten in derselben JVM verarbeitet, in der sie auch eingelesen werden. Es müssen also keine Daten (nachdem sie eingelesen wurden) durch das Cluster transportiert werden (siehe section 9.6). Durch die Wahl von 500ms als Länge eines Mini-Batches werden die Ergebnisse in subsekündlicher Aktualität gestreamt. Genügend Rechenleistung vorausgesetzt können also auch realzeitliche Analysen gefahren werden.

13.2.1 Rechenleistung

Die Spark Streaming Engine geht davon aus, dass zusätzlich zu den Receivern Cores für anschließende Transformationen der Daten zur Verfügung stehen. Leider müssen diese Cores zwingend alloziiert werden, auch wenn sie keine Transformation vornehmen und sich damit im Leerlauf befinden (siehe section 9.6). Da nicht alle Cores verwendet werden können, skalieren Streaming-Prozesse in unserer Implementierung also schlechter als Batch-Prozesse: Sie nutzen niemals die gesamte zur Verfügung stehende Rechenleistung.

Eigentlich sollen Receiver lediglich zum Einlesen der Daten verwendet werden und darauffolgende Tasks die Daten verarbeiten. In diesem Fall muss das Verhältnis von Receivern zu Tasks durch Ausprobieren eingestellt werden, damit nicht zu viele und nicht zu wenige Eingabedaten von den Receivern in das System gespeist werden. Wir haben uns gegen diesen üblichen Weg entschieden, weil die eingelesenen Daten schlicht zu groß waren und die Puffer, die zwischen Receivern und Tasks bestehen, in der Folge überliefen. Jetzt werden lediglich die (wesentlich kleineren) Ergebnisse verschickt und zwar zum Driver, wo sie ohnehin hin sollen. Die niemals ganz ausgenutzte Rechenleistung ist der Preis, den wir in Spark Streaming gegen einen Pufferüberlauf bezahlen müssen.

Auch die Leistung der durch die Receiver verwendeten Cores ist nicht immer optimal ausgenutzt: Da die Parameter eines Spark-Receivers nicht mehr geändert werden können, wenn er einmal läuft, muss die Verteilung der Eingabe-Datenströme vor dem Start der Receiver geschehen (siehe section 9.6). Sollte sich diese Verteilung als unausgewogen herausstellen, stellen die Receiver ihre Analysen zu auseinanderliegenden Zeitpunkten fertig. Zum Ende der Laufzeit befinden sich also mehr und mehr Rechenkerne im Leerlauf, die Geschwindigkeit der Analyse stagniert. Unausgewogene Verteilungen der Eingabeströme ergeben sich beispielsweise, falls als Eingabe Dateien stark variierender Größe verwendet werden.

13.2.2 Arbeitsspeicher

Einen besonderen Vorteil gegenüber den Batch-Prozessen genießen Streaming-Prozesse mit Blick auf den Arbeitsspeicher: Da die Ergebnisse als Mini-Batches alle 500ms zum Driver gestreamt werden, muss keiner der Nodes viel Arbeitsspeicher besitzen. Es müssen lediglich die Daten in den Speicher eine Node passen, die von ihr in einer halben Sekunde abgearbeitet werden können (plus natürlich für die Verarbeitung nötige Datenstrukturen). Das ein sehr geringes Volumen. Kann der Driver alle Daten der letzten halben Sekunde sofort wegschreiben, braucht auch er nicht viel RAM.

13.2.3 Fehlertoleranz

Spark Receiver sind nicht so fehlertolerant wie Tasks. Sollte ein Receiver fehlschlagen, wird auch er neu gestartet, da er aber bereits Daten in das System gespeist hat, kann es passieren, dass er dieselben Daten ein zweites Mal liefert. Dies ist beispielsweise der Fall, wenn die Eingabeströme aus Dateien im HDFS erzeugt werden: Wird ein Receiver neu gestartet, nachdem er eine halbe Datei gelesen hat, beginnt er wieder von vorn.

Erzeugt man einen Receiver für eine Datenquelle, die nicht sämtliche Daten ein zweites Mal liefern kann, besteht dieses Problem nicht. Dann trägt der Neustart eines fehlgeschlagenen Receivers zur Fehlertoleranz bei. Eine solche Datenquelle ist beispielsweise ein IP-Port, der online immer genau die gerade eintreffenden Daten liefert. Genau für solche Settings sind unsere Streaming-Prozesse geeignet.

13.3 Performanz der Erweiterungen

Die Erweiterungen, die wir in der PG an `streams` vorgenommen haben, zielen darauf ab, große Datenmengen effektiv zu verarbeiten. Daher sind wir insbesondere daran interessiert, welche Geschwindigkeit das Cluster für unterschiedliche Konfigurationen (Anzahl Cores und Menge an Arbeitsspeicher) erreicht. Wir messen diese Geschwindigkeit in Form von Eventraten, also wie viele Events durchschnittlich pro Sekunde verarbeitet werden.

Es werden die Eventraten unserer Erweiterungen für verschiedene Konfigurationen der Clusterressourcen verglichen. Für Teleskop-Daten beziehen wir auch die Geschwindigkeit zweier Konfigurationen im Torque-Cluster ein.

13.3.1 Feature Extraction auf MC-Daten

Zur Verfügung standen Monte-Carlo-simulierte Daten. Wir haben die Standard-Feature Extraction für MC-Daten mit verschiedenen Konfigurationen des Clusters für 2000 fits-

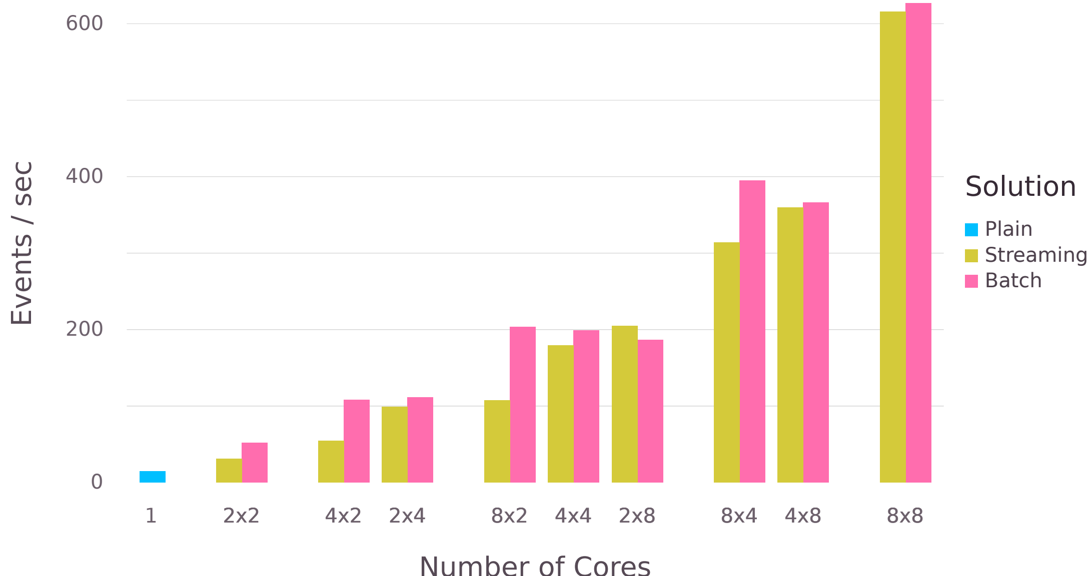


Abbildung 13.1: Eventraten der Feature Extraction auf MC-Daten

Dateien (22.789 Events) ausgeführt. Für jedes Event haben wir den Eintreffzeitpunkt der Ergebnisse am Driver aufgezeichnet.

Die Anzahl der Events geteilt durch die Gesamtzeit der Ausführung liefert uns eine mittlere Eventrate für die Standard-Feature Extraction auf MC-Daten. Figure 13.1 stellt die Eventraten für verschiedene Konfigurationen des Clusters dar. Die Konfigurationen sind nach dem Schema *Anzahl Workernodes* × *Anzahl Kerne pro Node* benannt und visuell nach der Gesamtanzahl Cores gruppiert. Der Umfang des Arbeitsspeichers machte für die Raten keinen Unterschied und ist deshalb nicht dargestellt.

Wie sich in Figure 13.1 erkennen lässt, erhöht sich die Geschwindigkeit der Analyse erwartungsgemäß mit der Anzahl der zur Verfügung stehenden Rechenkerne. Es zeigt sich dabei als durchaus relevant, auf wie viele Maschinen die Kerne verteilt sind. Insbesondere für Streaming-Prozesse ist eine Verteilung der Kerne auf wenige Maschinen vorzuziehen. Der Grund dafür liegt in dem Receiver-Konzept (siehe section 9.6): Da die Leerlauf-Cores pro Maschine alloziert werden müssen, bedeuten weniger Maschinen weniger Rechenkerne im Leerlauf. Für Batch-Prozesse erscheint es besser, mehr Maschinen mit weniger Kernen zu haben. Der Unterschied fällt hier aber geringer aus.

In subsection 13.1.1 und subsection 13.2.1 wurde bereits erörtert, dass bei einer unausgewogenen Verteilung der Eingabe-Datenströme zum Ende der Laufzeit die Eventraten stagnieren können. Da bei Batch-Prozessen die Tasks erst zugeordnet werden, wenn ein Core den vorigen Tasks abschließt, fällt der Effekt dort geringer aus, als bei Streaming-

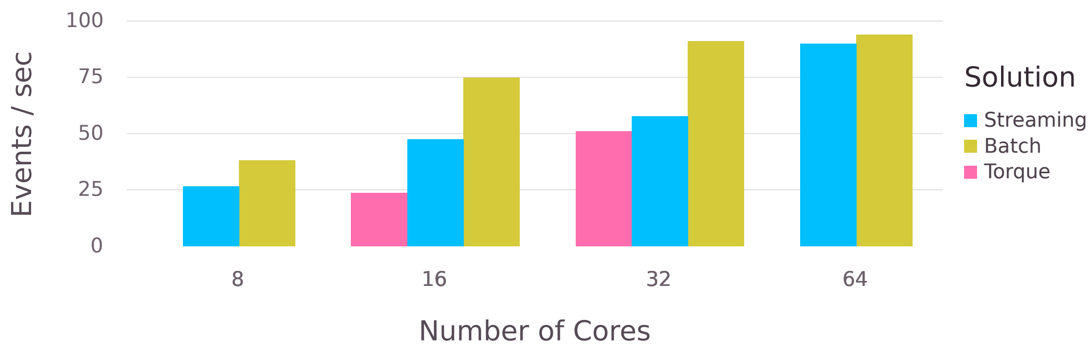


Abbildung 13.2: Eventraten der Feature Extraction auf Teleskop-Daten

Prozessen. Dort wird die Aufteilung der Eingabeströme bereits zu Beginn der Ausführung festgelegt.

Da bei jeder Ausführung eine andere Aufteilung der Datenströme vorgenommen werden kann, unterliegen die hier vorgestellten Ergebnisse also einer gewissen Schwankung, denn manche Aufteilungen sind besser als andere. Beispielsweise lässt die Ausführung 2×8 in Figure 13.1 vermuten, dass die Streaming-Variante schneller sei als der Batch. In den meisten Fällen sollte dies nicht der Fall sein. Aufgrund der Schwankungen kam es aber zu dem hier geplotteten Ergebnis. Auf eine umfassendere Messung mehrerer Ausführungen mit Bestimmung von Mittelwert und Standardabweichung der Raten mussten wir aus Zeitgründen leider verzichten.

13.3.2 Feature Extraction auf Teleskop-Daten

In einem zweiten Experiment haben wir die Standard-Feature Extraction für reale, vom Teleskop gelieferte Daten zur Messung der Performanz verwendet. Der verwendete Datensatz umfasst sämtliche Events, die im August 2013 aufgezeichnet wurden. Er setzt sich aus 38 Dateien (110.441 Events) zusammen, wobei die Dateigrößen zwischen 64, 5 MB und 10, 6 GB eine enorm große Spannweite abdecken.

Die Eventraten der Analyse sind in Figure 13.2 dargestellt. Da wir aus der Evaluation auf MC-Daten (siehe subsection 13.3.1) bereits wissen, wie die Rechenkerne für Batch- und Streaming-Prozesse am besten auf Maschinen zu verteilen sind, haben wir die jeweils beste Aufteilung gewählt. Im Vergleich zur MC-Analyse fallen die Eventraten insgesamt geringer aus, vermutlich weil die Feature Extraction hier aufwändiger ist.

Ebenfalls dargestellt ist die Geschwindigkeit zweier Konfigurationen des Torque-Clusters. Wie eindeutig ersichtlich ist, läuft das Torque-Cluster insgesamt langsamer als unsere Erweiterungen. Der hauptsächliche Grund dafür liegt in der Architektur von Torque: Da

die Daten nicht wie in Hadoop auf den Rechenknoten gespeichert werden – das Code-to-Data Prinzip also nicht umgesetzt werden kann – müssen große Datenvolumen durch das Netzwerk wandern. Damit wird das Einlesen und Abspeichern von Daten zum Flaschenhals für das Cluster. Wie wir beobachten konnten, wurden die einzelnen Rechenkerne so zu nur jeweils 50% ausgelastet. Die Auslastung der Kerne im Spark-Cluster lag stets bei über 80%.

Die stark variierende Größe der Dateien ist problematisch für eine vernünftige Verteilung der Dateiströme auf die zur Verfügung stehenden Rechenkerne (siehe subsection 13.1.1 und subsection 13.2.1). Damit sind auch stärker variierende Eventraten als für die MC-Daten zu erwarten, wo die Dateigrößen noch annähernd gleich waren. Der große Sprung der Raten des Streaming-Prozesses für 32 und 64 Cores lässt sich durch diese Varianz erklären. Auch hier war die Zeit für eine umfangreichere Evaluation nicht gegeben.

Ein weiteres Problem für die Verteilung der Dateien auf Rechenkerne ist deren geringe Anzahl. Da der Batch-Prozess nur so viele Cores nutzen kann, wie Eingabeströme zur Verfügung stehen, ergibt sich für 38 Dateien nur ein geringer Unterschied zwischen 32 und 64 Cores: Bei 64 Cores bleiben 26 Cores ungenutzt.

Diese Ergebnisse zeigen, wie wichtig eine geeignete Vorverarbeitung der Daten ist: Die zu prozessierenden Dateien sollten annähernd gleich groß sein, damit deren Verteilung gleichmäßig und mit geringer Varianz der Laufzeit geschehen kann.

Kapitel 14

Modellqualität in Spark ML

Mit unserer Software TELEPhANT wird streams um Funktionalitäten erweitert, die es erlauben, Spark ML Modelle zu trainieren und anzuwenden (vgl. chapter 10). Da Spark ML eine vielfältige Palette an nutzbaren Modellen und zugehörigen Parametern besitzt, wird in den nachfolgenden Abschnitten anhand einiger Beispiele gezeigt, wie sich die Modellvarianten von Spark ML auf den Daten des Anwendungsfalls verhalten. Insbesondere soll die Qualität der Klassifikations- und Regressionsmodelle sowie die benötigte Dauer des Trainiervorgangs in Abhängigkeit der Clusterressourcen gemessen werden. Gleichzeitig soll gezeigt werden, dass derartige Experimente recht einfach mit XMLs spezifiziert werden können, weshalb die zur Erhebung der Daten verwendeten XMLs im Anhang B eingesehen werden können.

14.1 Vergleich der Klassifikationsmodelle

Nachfolgend soll zunächst erläutert werden, in welcher Art und Weise die Qualität der jeweiligen Klassifikationsmodelle bestimmt wird, und daraufhin genauer auf die einzelnen Ergebnisse eingegangen werden.

Qualitätsbestimmung Zur Qualitätsbestimmung der Klassifikationsmodelle sind gelabelte Daten zu verwenden. In diesem Fall handelt es sich dabei um einen Datensatz von etwa 140.000 MonteCarlo-Events, welcher sich in etwa 80.000 Gamma- und etwa 60.000 Hadron-Events gliedert. Für die verteilte Verarbeitung wird der Datensatz in 20 Partitionen aufgeteilt und in einem HDFS, das sich über einen Cluster mit 10 Rechnern erstreckt, gespeichert. Als Replication Factor wurde hier 3 gewählt, um Wartezeiten durch das Versenden von Daten über das Netzwerk zu vermeiden.

Die eingesetzten Modelle sollen die korrekte Zuordnung der Events anhand von 32 ganzzahligen sowie reellen Features voraussagen, wobei zu beachten ist, dass Spark ML grund-

sätzlich alle Features als reellwertig interpretiert. Diese Eigenschaft stellt hinsichtlich des Anwendungsfalls jedoch kein Problem dar. Da lediglich zwischen Gamma- und Hadron-Events unterschieden wird, handelt es sich also um eine binäre Klassifikationsaufgabe (vgl. chapter 4), bei welcher Gamma-Events als „Positiv“ und Hadron-Events entsprechend als „Negativ“ definiert werden.

Zur Messung der Qualität der jeweiligen Modelle sind die True-Positive-Rate („Recall“), die True-Negative-Rate („Specifity“) sowie die Präzision der Vorhersage zu bestimmen. Dies geschieht in einer solchen Weise, dass 90% der Daten zum Trainieren des Modells verwendet werden, welches anschließend auf die verbleibenden 10% des Datensatzes angewendet wird. Die aus dem Modell resultierende Vorhersage wird daraufhin mit dem bereits bekannten Label verglichen, dessen Korrektheit als gegeben anzusehen ist. Jede Konfiguration wird fünfmal ausgeführt und aus den erhaltenen Ergebnissen ein Mittelwert gebildet. Zudem wird in Figure 14.1 die einfache Standardabweichung eingezeichnet, um die Schwankung der Werte zu visualisieren. Bei der Ausführung werden die Klassifikationsmodelle RandomForest sowie GradientBoostedTrees in den Standardeinstellungen belassen, es wird lediglich die Option `cacheNodeIds` zum Zwecke des Performanzgewinns aktiviert. Die konkreten Werte der Standardeinstellungen können ebenso wie die verwendete XML im Anhang B.1 nachgeschlagen werden. Bei der Verwendung des MultilayerPerceptron-Classifier muss die Netzgröße manuell festgelegt werden, in diesem Fall werden dort 20 Ebenen mit 32 Knoten sowie eine Ausgabeebene mit zwei Knoten verwendet.

Naive Bayes Da der in Spark ML unterstützte „Naive Bayes“-Klassifikator auf boolesche Features ausgelegt ist, kann dieser nicht auf sinnvolle Weise eingesetzt werden. Zwar ist in einschlägiger Fachliteratur eine Variante für reellwertige Features („Gaussian Naive Bayes“) zu finden, diese steht allerdings in Spark ML bis jetzt noch nicht zur Verfügung.

RF und GBT RandomForest und GradientBoostedTrees liefern an dieser Stelle vergleichbare Ergebnisse. GBT erkennt (unter Beibehaltung der Standardeinstellungen) Hadron-Events zuverlässiger als dies bei RF der Fall ist (vgl. True-Negative-Rate in Figure 14.1), ist jedoch hinsichtlich der Erkennung von Gamma-Events weniger verlässlich (vgl. True-Positive-Rate).

MPC Der MultilayerPerceptronClassifier legt ein sehr extremes Verhalten an den Tag, welches sich darin äußert, dass er alle Events als Gamma-Events klassifiziert. Dies resultiert zwar in einer perfekten True-Positive-Rate von 1.0, jedoch zugleich in einer True-Negative-Rate von 0.0 sowie einer Präzision von 57%, was dem Verhältnis von Gamma- und Hadron-Events in dem Testdatensatz entspricht. Da dieses Verhalten auch von anderen Anwendern beobachtet wurde [88], ist wie dort zu vermuten, dass zu viele nicht aussagekräftige Features betrachtet werden und zudem die Features nicht normiert wurden.

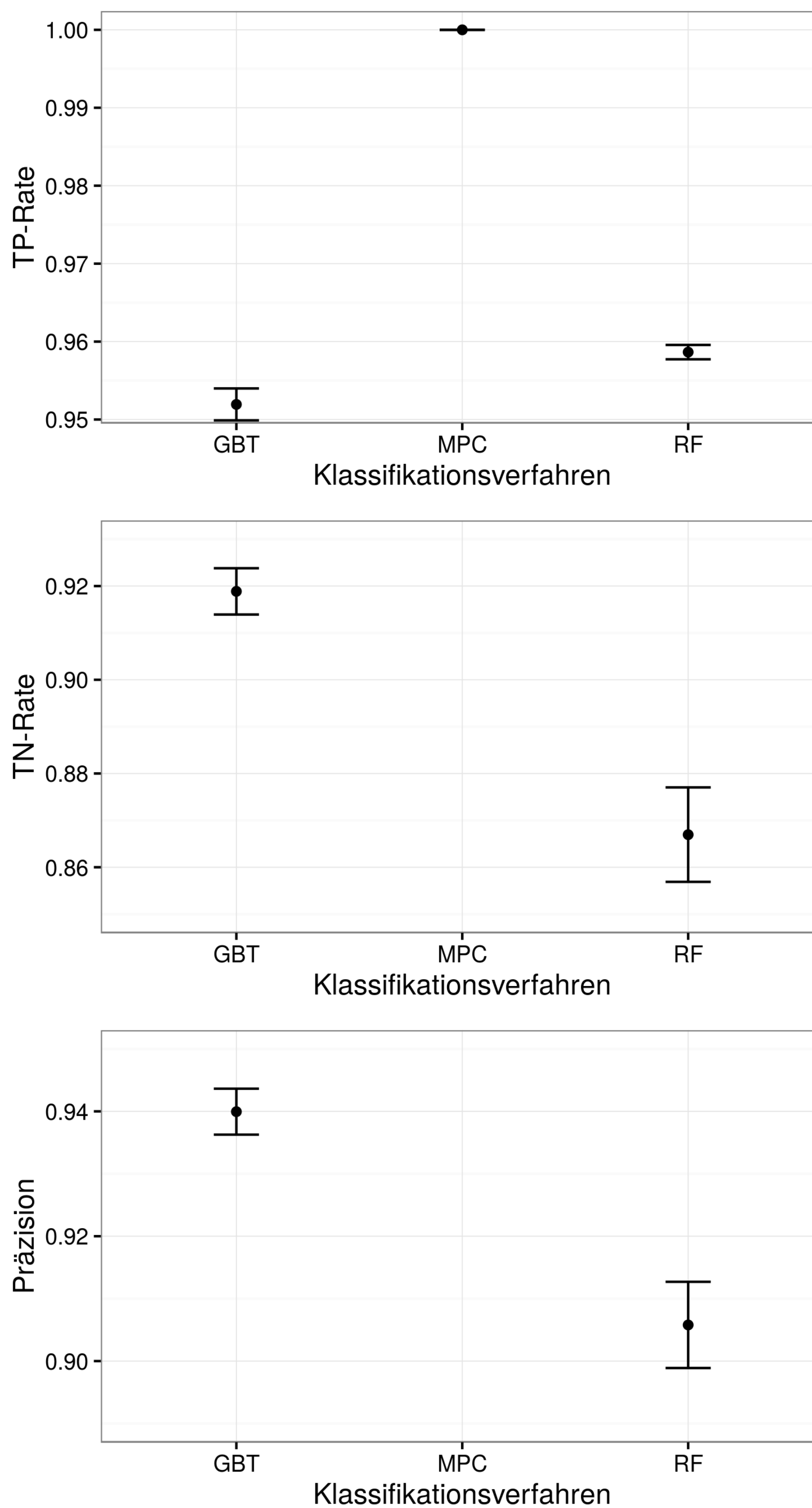


Abbildung 14.1: Die True-Positive-Rate, True-Negative-Rate und die Präzision der verschiedenen Klassifikationsverfahren. Eingezeichnet ist der Mittelwert aus 5 Replikationen und die einfache Standardabweichung. Die Werte für den MPC sind in den beiden letzten Grafiken erheblich schlechter und werden daher nicht aufgeführt.

14.2 Vergleich der Regressionsmodelle

Nach einem Vergleich der Klassifikationsmodelle sollen nun die verschiedenen Regressionsmodelle evaluiert werden. Zu diesem Zweck wird eine Energie-Abschätzung (vgl. subsection 1.3.2) auf Basis eines erweiterten Datensatzes mit etwa 550.000 Events durchgeführt. Dieser Datensatz weist ein erhebliches Ungleichgewicht zwischen Gamma- und Hadronen-Events auf und konnte daher bei den bisherigen Experimenten, die auf eine gleichermaßen präzise Erkennung von Gamma- und Hadronen-Events abzielen, nicht verwendet werden. Im Rahmen einer Regression eignet er sich aber aufgrund seines Umfangs an Trainingsdaten hervorragend. Analog zum vorherigen Datensatz wird auch dieser in 20 Partitionen geteilt und in einem HDFS verteilt gespeichert.

Der RandomForest-Regressor und der GradientBoostedTrees-Regressor werden hier in den jeweiligen Standardeinstellungen verwendet, welche im Anhang B.3 eingesehen werden können. Da Spark ML mit dem *root mean squared error* sowie R^2 zwei Bewertungsfunktionen zur Verfügung stellt, können anhand der Ergebnisse ebendieser die verwendeten Regressoren bewertet werden (vgl. Figure 14.2). Im betrachteten Fall schneidet der GBT-Regressor deutlich besser ab, da sein root mean squared error bei etwa 1410 liegt, wohingegen der RF-Regressor sich in einem Bereich von etwa 1575 verorten lässt. Hinsichtlich des Faktors R^2 weist GBT einen etwas besseren Wert auf, jedoch schneidet auch RF noch akzeptabel ab.

14.3 Trainingszeit von Modellen

Im Folgenden wird die Performanz des Trainierens von Modellen in Abhängigkeit von der Anzahl der verwendeten Rechenknoten und Prozessoren untersucht. Um den Effekt steigender Zahlen von Knoten und Prozessoren deutlicher sichtbar zu machen, wird an dieser Stelle in Form eines RandomForest-Klassifikatiors mit 80 Bäumen und einer Maximaltiefe von 25 ein recht aufwendiges Modell verwendet. Gleichzeitig wird wieder auf den erweiterten Datensatz zurückgegriffen, da hier nicht die Klassifikationsqualität gemessen wird.

Bei dieser Messreihe wird auf die Möglichkeit des Spark Shell-Scripts zurückgegriffen, die Anzahl der beteiligten Rechenknoten („Executors“) sowie die Anzahl an Prozessoren pro Rechenknoten („Cores“) einzustellen (vgl. chapter 17).

Aus Figure 14.3 ist ersichtlich, dass der Trainingsdurchsatz mit wachsender Knoten- und Prozessorzahl deutlich ansteigt. Allerdings resultiert aus doppeltem Ressourceneinsatz nicht eine Verdopplung der Leistung. Dieses Phänomen wird im Bereich der parallelen und verteilten Datenbanken als „sub-linear speed-up“ bezeichnet [83]. Gründe hierfür

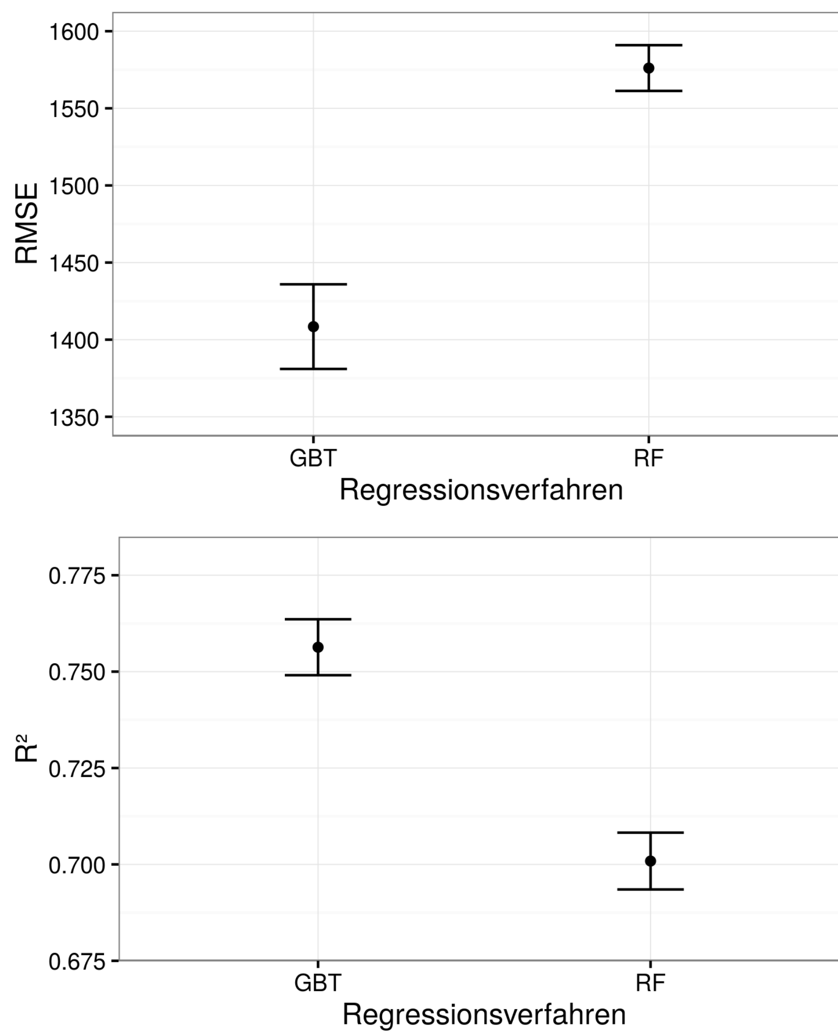


Abbildung 14.2: Der root mean squared error und der R^2 Wert für die Regressionsverfahren. Eingezeichnet ist der Mittelwert aus 5 Replikationen und die einfache Standardabweichung.

können in einer ungleichen Verteilung der Arbeitslast oder in hohen Anteilen an nicht-parallelisierbaren Aufgaben liegen.

Auffällig ist, dass die Performanz ab einer Zahl von vier Rechenknoten mit jeweils acht Prozessoren kaum noch wächst. Es wird vermutet, dass die Aufteilung des Datensatzes in 20 Partitionen hier limitierend wirkt. Laut [78] wird jede Partition zu einem eigenen „task“, wobei jeder Prozessor einen task gleichzeitig bearbeiten kann. Damit bräuchte eine Konfiguration von 8 Rechenknoten mit 8 Prozessoren einen Datensatz mit 64 Partitionen, um alle Prozessoren sinnvoll auszulasten. Dies sollte beim späteren Einsatz von Apache Spark beachtet werden.

Die Messung des Klassifikationsdurchsatzes ist schwieriger als beim Trainingsdurchsatz, weil Spark lazy evaluation anwendet. Dabei wird die Anweisung, das Modell auf den Datensatz anzuwenden, erst dann ausgeführt, wenn eine Operation mit anschließender Ausgabe diese Klassifikation benötigt. Eine Zeitmessung, die das nicht berücksichtigt, würde nur die Zeit zum Absenden des Klassifikationsauftrags, nicht jedoch die eigentliche Dauer der Klassifikation messen.

Daher wurde eine zusätzliche Leseoperation eingefügt, die zählt, wie oft die Klasse 1.0 vergeben wurde. Das Ergebnis dieser Operation ist nicht von Interesse, sie wird lediglich eingesetzt, um eine Klassifikation mithilfe des Modells zu garantieren. Es wird also die Dauer dieser beiden Operationen zusammen gemessen, wobei es im Nachhinein nicht trivial möglich ist, den Zeitanteil für die reine Klassifikation herauszurechnen. Die hier angegebenen Klassifikationsraten sind also als vorsichtige untere Schranken zu interpretieren. Wie in Figure 14.3 erkennbar steigt die Klassifikationsrate zunächst kaum merklich, wächst aber ab einer Knotenzahl von zwei mit jeweils acht Prozessoren rapide. Interessanterweise ist in diesem Fall keine Limitierung durch die Verwendung von 20 Partitionen zu beobachten.

14.4 Einfluss der Waldgröße auf die Modellqualität

Da durch die Standardeinstellungen des RandomForest-Klassifikators noch nicht die gewünschte Modellqualität erzielt wird, soll untersucht werden, wie die Qualität des Modells bei Beibehaltung des Datensatzes aus section 14.1 durch geeignete Parameterwahl optimiert werden kann. Zu diesem Zweck wird mittels der Variation der Anzahl von Bäumen der Einfluss ebendieser auf die True-Positive-Rate, True-Negative-Rate sowie auf die Präzision gemessen. Um realitätsnahe Ergebnisse zu erzielen, wird bei allen Konfigurationen die maximale Tiefe der Bäume auf einen Wert von 25 gesetzt.

Des Weiteren wird untersucht, wie die Anzahl der Bäume die Trainingszeit beeinflusst, was anhand des Trainingsdurchsatzes, der als die Anzahl von Trainingsevents dividiert durch die Trainingszeit in Sekunden definiert ist, beschrieben wird. Dabei wurden im Cluster

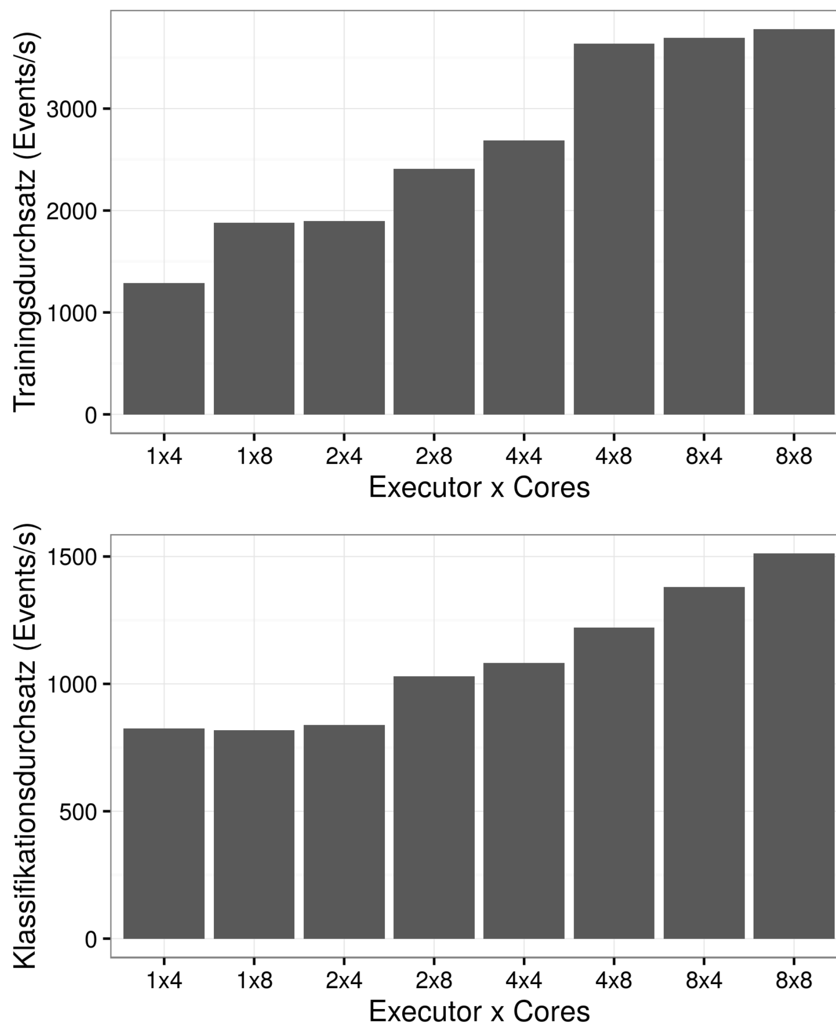


Abbildung 14.3: Trainingsdurchsatz und Klassifikationsdurchsatz in Abhängigkeit von verwendeten Rechenknoten und Prozessoren.

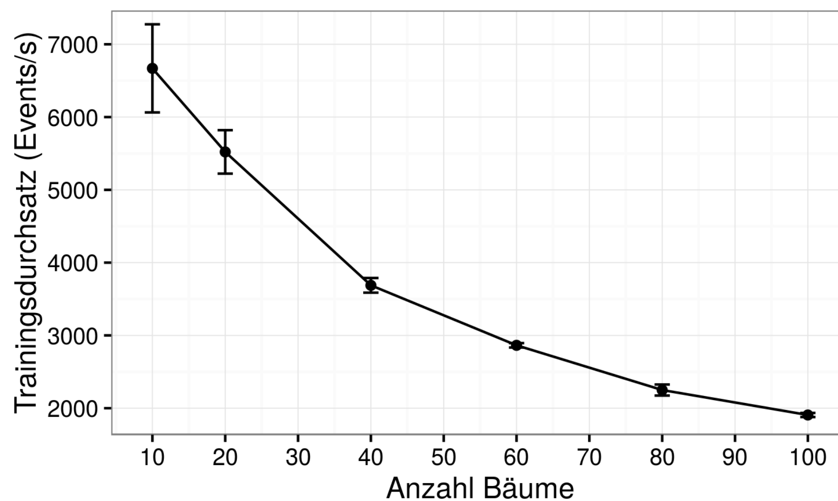


Abbildung 14.4: Durchsatz in Abhängigkeit der Größe des RandomForest. Eingezeichnet ist der Mittelwert aus 5 Replikationen und die einfache Standardabweichung.

8 Rechenknoten mit jeweils 8 Prozessoren eingesetzt. Wie im bisherigen Klassifikationsexperiment werden hier in fünf Replikationen 90% des Datensatzes als Trainingsdaten und 10% als Testdaten verwendet.

Aus Figure 14.5 ist ersichtlich, dass die Anzahl der Bäume sich nicht signifikant auf die True-Positive-Rate auswirkt. Die True-Negative-Rate hingegen steigt zunächst sehr stark, ab einer Waldgröße von 40 Bäumen jedoch nur noch langsam an. Dabei ist die stochastische Schwankung recht groß. Analog zur True-Negative-Rate steigt die Präzision des Modells zunächst stark, ab einer Anzahl von 40 Bäumen allerdings nur noch langsam an. Zudem ist aus Figure 14.4 ersichtlich, dass der Trainingsdurchsatz des Modells mit wachsender Zahl der Bäume rapide sinkt.

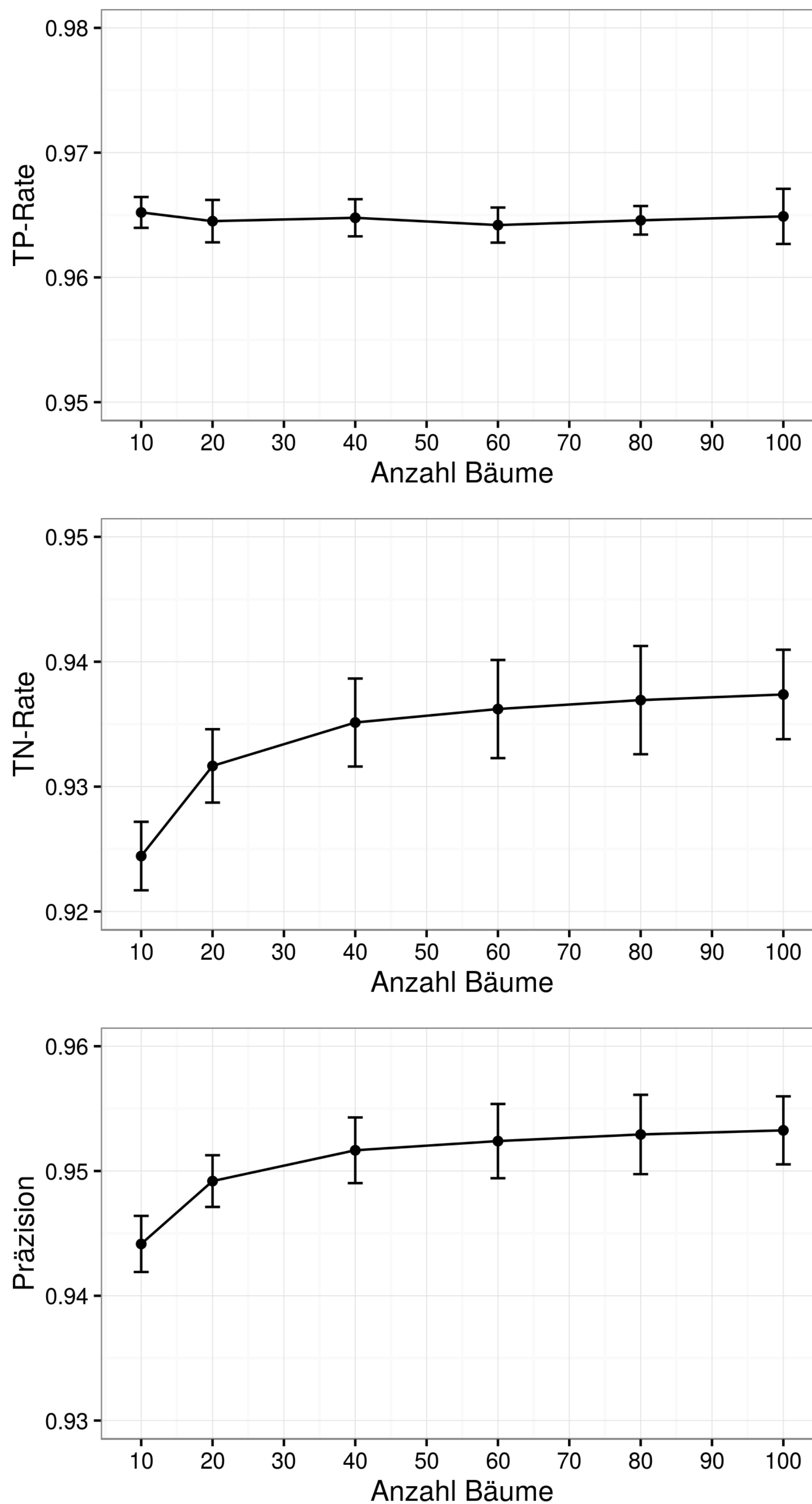


Abbildung 14.5: Die erzielte True-Positive-Rate, True-Negative-Rate und die Präzision in Abhängigkeit der Größe des RandomForest. Eingezeichnet ist der Mittelwert aus 5 Replikationen und die einfache Standardabweichung.

Kapitel 15

Fazit

15.1 Ergebnisse

Im Laufe der vergangenen beiden Semester konnte die Projektgruppe zahlreiche Erfolge verzeichnen, über welche im Folgenden in resümierender Weise ein kurzer Überblick gegeben werden soll. Die Leistungen der Projektgruppe lassen sich unter einem einzigen, bedeutenden Stichpunkt subsummieren, nämlich der Erweiterung des `streams`-Frameworks für die verteilte Ausführung mit Spark. Diese wiederum setzt sich aus verschiedenen Teilaspekten zusammen.

Zunächst ist hervorzuheben, dass die Rohdaten mittels einer MongoDB indexiert wurden (vgl. chapter 7), um einerseits die Durchsuchbarkeit der Events zu gewährleisten und andererseits eine Steigerung der Performanz zu erreichen. Einen nicht unbedeutenden Beitrag zu Letzterer leistet ebenfalls die verteilte Datenhaltung mit HDFS sowie die Realisierung des Code-to-Data-Prinzips, welche eine Neuerung im Gegensatz zum TORQUE-Cluster der Physiker darstellen (vgl. section 3.1).

Eine weitere Innovation besteht im beliebig skalierbaren DistributedProcess, der eine Verteilung der Rechenlast und abschließend eine automatisierte Zusammenführung der Teilergebnisse ermöglicht. Eine solche Verteilung ist sowohl für Streaming als auch für Batchverarbeitung möglich (vgl. chapter 9).

Um verteilte Lernalgorithmen verfügbar zu machen, wurde Spark ML als neues `streams`-Element integriert, sodass sich die dadurch bereitgestellte Pipeline nun via XML konfigurieren lässt. Auf Basis dieser Pipeline ist es nun möglich, Modelle zu trainieren, zu speichern, zu laden und anzuwenden. Im Zuge der Integration der ML-Pipeline in das `streams`-Framework wurde dieses zudem um die XML-Tags `task` und `operator` sowie die damit verbundenen Funktionalitäten erweitert. Einen besonderen Stellenwert nimmt dabei der `operator` ein, der als neue Schnittstelle zur Bearbeitung von DataFrames fungiert (vgl. chapter 10 und chapter 19).

Im Vergleich zum Status Quo der Physiker sollte erwähnt werden, dass durch die Einführung einer REST-API sowie die zur Nutzung dieser geschaffene Web-Oberfläche `TELEPhANT` um ein Vielfaches einfacher und komfortabler hinsichtlich der Bedienung konzipiert wurde. Ebendiese Web-Oberfläche ermöglicht nicht nur das unproblematische Starten von Jobs durch das Submitten einer XML, sondern zudem das Managen von Jobs im Cluster und bietet außerdem zahlreiche Konfigurationsoptionen. Details diesbezüglich lassen sich im Benutzerhandbuch finden (vgl. chapter 18).

Was den Umstieg auf `TELEPhANT` besonders einfach machen dürfte, ist die Tatsache, dass bestehende `streams`-Funktionalitäten einfach weiterverwendet werden können. Lediglich die Verwendung einiger neuer XML-Tags muss erlernt werden. Ein Überblick über diese wird im Benutzerhandbuch gegeben (vgl. subsection 19.3.3).

15.2 Ausblick

In den vergangenen beiden Semestern hat die Projektgruppe vieles erreicht - nichtsdestotrotz sind einige zusätzliche Erweiterungen der Funktionalitäten denkbar, für welche die verfügbare Zeit leider nicht mehr ausreichte. Die zur Indexierung der Rohdaten eingesetzte MongoDB läuft zurzeit lediglich auf einem einzigen Knoten, könnte jedoch geshardet, also auf mehreren Knoten verteilt eingesetzt werden, um die Ausfallsicherheit des Index zu erhöhen. Zudem erscheint es sinnvoll, die MongoDB der Projektgruppe sowie die der Physiker auf einen Nenner zu bringen, sofern `TELEPhANT` tatsächlich in der Forschung eingesetzt werden sollte. In welcher Weise dies geschehen sollte, ist noch zu klären. Ein weiteres mögliches Feature ist das Streaming von Daten nicht nur aus Fits-Dateien, sondern auch aus anderen Quellen, beispielsweise von einem Forschungszentrum in der Schweiz, was für die Physiker einige Arbeitsabläufe vereinfachen könnte. Die Grundvoraussetzungen für diese Funktionalität sind bereits vorhanden und müssten lediglich erweitert werden, falls gewünscht. Dies könnte darüberhinaus mit Realzeitanalysen ebendieser gestreamten Daten verbunden werden. Eine letzte Erweiterung, die wir in Betracht zogen, ist das Einführen einer umfangreichen Instrumentenmonitoring-Funktionalität, allerdings wäre es zur Realisierung dieser vonnöten, die genauen Bedürfnisse und Wünsche der Physiker in dieser Hinsicht zu kennen, um tatsächlich ein sinnvolles Software-Feature zu gestalten.

15.3 Retrospektive der Organisation

Wie in chapter 12 beschrieben, haben wir zur Organisation der PG das agile Vorgehensmodell SCRUM festgelegt. Wir haben unsere Umsetzung des Modells von der initialen Parametrisierung ausgehend in den letzten zwei Semestern ständig weiterentwickelt. Die

Änderungen, die wir an unserer Umsetzung von SCRUM vorgenommen haben, spiegeln die Lösung von organisatorischen Problemen wider, die wir mit der Zeit identifiziert haben. Sie zeigen, dass wir in organisatorischer Hinsicht in den letzten zwei Semestern viel lernen konnten.

Zu Beginn der PG hatten wir einige organisatorische Probleme, auf die zunächst in subsection 15.3.1 eingegangen werden soll. Anschließend beschreiben wir die Änderungen, die wir im ersten und zweiten Semester an unserer Umsetzung von SCRUM vorgenommen haben (siehe subsection 15.3.2 und subsection 15.3.3). Eine abschließende Bewertung wird in subsection 15.3.4 vorgenommen.

15.3.1 Projekt-Initialisierung

Scrum fordert, dass im Team ein tiefgehendes Verständnis über die Vision des Endproduktes vorliegt. Nur dadurch ist nachvollziehbar, was Teilziele für den Projekterfolg bedeuten, und umrissen, was möglicherweise im Vorhinein für zukünftige Arbeitspakete zu bedenken ist. Wir haben uns zu Beginn des ersten Semesters schwer damit getan, die Product Vision zu konkretisieren. Auch wenn abstrakt klar war, welche Prozesse zur Analyse der Daten abzubilden sind, lag der Weg dahin lange Zeit im Dunkeln. Ein Grund dafür war, dass wir mit den verwendeten Technologien nur wenig Erfahrung besaßen. Erst im Laufe des ersten Semesters konnten wir ein konkretes Bild der Product Vision schaffen.

Scrum nimmt weiterhin an, dass das Team die für das Projekt nötige Expertise bereits mitbringt, im Zweifelsfall durch im Vorhinein durchgeführte Schulungen. Dadurch werden Gliederungen auf geeigneter Abstraktionsstufe und zutreffende Aufwandsschätzungen ermöglicht. Auch lässt sich die Projektinitialisierung so schneller abwickeln. Für Projektgruppen kann die Annahme umfangreicher Expertise allerdings nicht vollends zutreffen, da dort das nötige Wissen erst vermittelt werden soll. Uns fehlten zu Anfang insbesondere Erfahrungen mit Spark und dem Streams-Framework.

15.3.2 Organisation im ersten Semester

Initial bestand aufgrund der inkonkreten Product Vision und den zunächst notwendigen Lernerfolgen bezüglich Spark und Streams ein besonders hoher Abstimmungsaufwand. Daher haben wir uns entschlossen, unsere Sprint-Meetings wöchentlich abzuhalten. Wir mussten dabei feststellen, dass Scrum für seine wöchentlichen Sprint Planning Meetings implizit einen höheren Arbeitsumfang, als für die Projektgruppe vorgesehen, annimmt: Während in einem regulären Arbeitsleben etwa acht tägliche Arbeitsstunden üblich sind, sieht das Modulhandbuch des Masterstudiengangs Informatik acht Semesterwochenstunden für die Projektgruppe vor [85]. Wir haben dadurch mit unserem wöchentlichen Sprint

Planning Meeting einen Umfang abgedeckt, für den von Scrum eigentlich ein Daily Meeting angedacht ist.

Durch diesen übersichtlichen Sprint-Umfang erschien es zunächst nicht zielführend, Scrum formal durchzuführen, also ein PBL, ein SBL oder ein IBL gewissenhaft zu führen. Dadurch wurde allerdings der Abstimmungsaufwand weiter erhöht und wir mussten in den wöchentlichen Meetings besonders viele Inhalte abhandeln. Die Treffen wurden länger als vielleicht nötig.

Zudem haben sich die meisten wöchentlichen Meetings zu Arbeitsmeetings ausgewachsen, die einzelne Probleme in einer Tiefe diskutiert haben, die nicht für alle Teilteams relevant war. Erst später im Semester haben wir regelmäßige Treffen der Teilteams etabliert, in denen die Arbeit erledigt und teilthemenbezogene Abstimmung erzielt wurde. Dadurch fielen die wöchentlichen Hauptmeetings sinnvollerweise kürzer aus.

Für Sprint-Retrospektiven („Was lief gut, wie können wir den Prozess verbessern?“) war eine Woche kein ausreichender Sprint-Umfang. Ein dediziertes Meeting zur Bewertung des Prozesses wurde auch nicht abgehalten. So haben wir nicht abstimmen können, wie wir unseren Entwicklungsprozess optimieren können.

15.3.3 Organisation im zweiten Semester

Durch die für den Zwischenbericht angefertigte erste Retrospektive haben wir die organisatorischen Probleme identifiziert, die sich im ersten Semester ergeben haben. Wir haben also Maßnahmen getroffen, um diese Probleme im zweiten Semester zu lösen.

Ein wesentlicher Punkt war es, die Gruppentreffen weiter zu optimieren. Wir konnten bereits zum Ende des ersten Semesters feststellen, dass Arbeitstreffen in den Teilteams die Planungstreffen mit der gesamten Gruppe fokussieren konnten. Wir haben darauf aufbauend beschlossen, weniger Planungstreffen zu veranstalten. Da wir so mehr Zeit zur Implementierung zwischen zwei Treffen besaßen, wurde die formale Durchführung eines Sprints erstmals attraktiv. Einem hohen Teil des Abstimmungsaufwandes konnten wir durch den intensivierten Einsatz von Atlassian JIRA beikommen.

Sicher wurden diese Entwicklungen dadurch begünstigt, dass wir durch die Formulierung des Zwischenberichtes die Product Vision ganz konkret festgelegt hatten. Der Ausblick des Zwischenberichtes diente uns als Zielformulierung im zweiten Semester, sodass wir ausgesprochen zielgerichtet arbeiten konnten. Arbeitspakete waren präziser planbar, weil sie im Kontext bestehender Ergebnisse standen. Ein genaues Bild des Endproduktes im Auge zu haben, hat überdies die Motivation der Gruppe regelrecht beflügeln können.

15.3.4 Abschließende Bewertung

Die angenommene Erfahrung mit verwendeten Technologien und die Annahme eines tiefgehenden Verständnisses der Product Vision haben Scrum für die Initialisierung des Projekts nicht so recht aufgehen lassen (siehe subsection 15.3.1). Wir sind dadurch erst recht spät aus dieser Findungsphase ausgetreten. Insbesondere waren einige Zeit lang keine sinnvollen Inkremente planbar.

Die von uns zunächst gewählte Sprintlaufzeit von einer Woche ließ eine formale Durchführung (PBL, SBL, IBL) von Scrum nicht sinnvoll erscheinen. Durch die nicht von Scrum vorgesehene Durchführung unserer Treffen haben wir wir viel Zeit in den Treffen verbraucht, wobei nicht immer alle von dieser Zeit profitieren konnten (siehe subsection 15.3.2).

Im zweiten Semester sind wir die organisatorischen Probleme des ersten Semesters angegangen. Seltenere fokussiertere Treffen mit der gesamten Gruppe machten die formale Durchführung von Scrum erstmals attraktiv. Die Formulierung des Zwischenberichtes konkretisierte die Product Vision, wodurch Arbeitspakete besser planbar wurden. Zusammen mit den im ersten Semester gewonnenen Erfahrungen mit Spark und Streams ermöglichten diese Umstände ein weitaus effizienteres Arbeiten als noch im ersten Semester. Auch sind wir heute als Team eingespielter als noch zu Beginn der Projektgruppe.

Es lässt sich festhalten, dass wir in der Projektgruppe allerhand über die praktische Durchführung von SCRUM gelernt haben. Insbesondere haben wir nun eine Vorstellung davon, unter welchen Umständen SCRUM sinnvoll ist und unter welchen nicht. Ein Projektmanagement-System wie JIRA zu kennen, kann den Einstieg in andere agil geführte Projekte erleichtern.

Da im zweiten Semester die in den Sprint Planning Meetings festgelegten Ziele meist erreicht wurden und wir dazu die von uns geforderte Arbeitszeit im Großen und Ganzen leisteten, hat sich nicht die dringende Notwendigkeit einer formal durchgeführten kontinuierlichen Prozessoptimierung ergeben. So haben wir Kanban nicht praktisch kennen lernen können.

Teil IV

Benutzerhandbuch

Kapitel 16

Vorbereitung eines Clusters

Die in dieser Projektgruppe entwickelte Bibliothek arbeitet gerade dann effizient, wenn die Berechnung in einem ganzen Cluster ausgeführt wird. Dazu müssen die folgenden Vorbereitungsschritte einmalig erfolgen.

1. **Vernetzung.** Alle Rechner des Clusters sollten so eingerichtet werden, dass sie sich in einem gemeinsamen, lokalen Netzwerk befinden.

2. **Hadoop und Ressourcenmanager einrichten.** Auf jedem Rechner des Clusters muss Apache Hadoop 2.6.2 (oder eine kompatible spätere Version) installiert und eingerichtet werden. Lediglich auf einem Rechner des Clusters wird ein Ressourcenmanager installiert, der zu bearbeitende Jobs annimmt und im Cluster verteilt. Im Rahmen der Projektgruppe wurde zu diesem Zweck Apache YARN eingesetzt.

3. **Verteiltes Dateisystem einrichten.** Damit alle Knoten des Clusters Zugriff auf alle Daten haben, empfiehlt sich die Nutzung eines verteilten Dateisystems. Hierfür bietet sich das HDFS an, weil das Zusammenspiel mit Hadoop und Spark gut funktioniert. Wie bei der zentralen Annahme der Jobs muss auch für das verteilte Dateisystem ein einzelner Rechner ausgewählt werden, der alle Anfragen entgegennimmt. Außerdem sollte vor dem Einspielen der Daten die Zahl der Replikationen geeignet gewählt werden. Eine große Anzahl an Replikationen führt zu einem hohem Speicherplatzbedarf, verringert aber potenziell die Bearbeitungsdauer der Jobs, weil weniger Dateien über das Netzwerk gesendet werden müssen.

4. **Weitere Software im Cluster installieren.** Nun können weitere, optionale Komponenten installiert werden. Es empfiehlt sich, ein Datenbankmanagementsystem auf jedem Rechner des Clusters zu installieren. Das Datenbankmanagementsystem darf seine Daten aber nicht im verteilten Dateisystem ablegen, da die Rechner des Clusters sonst ihre Datenbanken gegenseitig überschreiben! Wenn gewünscht ist, dass sich alle Rechner im Cluster eine Datenbank teilen, müssen die entsprechenden Funktionen des Datenbankmanagementsystems verwendet werden.

Anschließend muss jeder Rechner außerhalb des Clusters, der Jobs an diesen schicken soll, ebenfalls vorbereitet werden. Hierfür reicht es aus, Hadoop 2.6.2 sowie Spark 1.6.2 zu installieren und die jeweils genannten Einrichtungsschritte zu befolgen.

16.1 Verfügbarkeit von Dependencies

Für die Erweiterung von Streams existieren einige Abhängigkeiten zu verwendeten Bibliotheken. Die folgenden Dependencies müssen zur Laufzeit im Cluster vorhanden sein:

Streams Die Maven-Module `streams-core` und `streams-runtime` beinhalten alle für die Ausführung einer in XML spezifizierten Applikation nötigen Funktionen. `streams-hdfs` stellt einen Handler für URLs des HDFS-Protokolls zur Verfügung, was für das Öffnen von XML-Spezifikationen nötig ist.

FACT-Tools Das Projekt `fact-tools` ist eine Sammlung von Streams-Prozessoren und weiteren Funktionen zur Analyse der FACT-Daten im Streams-Framework.

Spark `spark-core` stellt die Basis-Konzepte von Spark zur Verfügung, die für eine verteilte Ausführung im Cluster nötig sind. `spark-mllib` und `spark-sql` werden für die Verwendung der Lernbibliothek MLlib benötigt.

Hadoop `hadoop-client` ist, neben `streams-hdfs` nötig, um Dateien aus dem HDFS zu lesen. `mongo-hadoop-core` ist für die Anbindung der MongoDB verantwortlich.

Damit nicht bei jeder Ausführung ein „Über-jar“, also ein Archiv mit sämtlichen Dependencies vom Client ins Cluster kopiert werden muss, haben wir ein Maven-Projekt für die Sammlung dieser Dependencies erstellt. Das aus diesem Projekt erstellte jar-Archiv kann dann für sämtliche Ausführungen, sofern keine Änderungen an den Abhängigkeiten nötig sind, verwendet werden.

Wir laden dazu die Dependency-Jar ins HDFS und übergeben ihren Pfad bei jeder Ausführung an `spark-submit`. Yarn erkennt den HDFS-Pfad und nimmt keine Kopie vom lokalen System vor. Um einen Job auszuführen, muss damit lediglich ein kleines Archiv mit dem aktuellen Stand unserer Streams-Erweiterung hochgeladen werden. Da die Abhängigkeiten in unserem Fall ein Archiv aus weit über 100MB ergeben, spart dieses Vorgehen eine Menge Zeit, insbesondere während der Entwicklung, wenn im Minutentakt eine neue Programmversion getestet werden muss.

16.2 Starten der REST API & Web-UI

Zum Starten der REST API auf dem Server gibt es im wesentlichen zwei Wege: „per Hand“

und über Docker. Beide werden im Folgenden kurz beschrieben.

16.2.1 Standard

Dies ist der klassische Weg, der keine besonderen Voraussetzungen an den oder die Server stellt.

MongoDB Zum Installieren der MongoDB sollte am besten der Anleitung auf der MongoDB-Website gefolgt werden [71]. Es ist nicht zwingend notwendig, dass die MongoDB und die REST API auf dem selben Server laufen, da die Verbindung zwischen den Beiden über einen Kommandozeilenparameter gesteuert werden kann.

REST API Die Jar-Datei der REST API enthält einen eingebauten Tomcat-Server, sodass es über einen einfachen Java Befehl gestartet werden kann, z.B.

```
java -jar target/apiwebapp-1.0-SNAPSHOT.jar [--server.port=8080]
      [--mongodb=mongodb://ls8cb01.cs.uni-dortmund.de:27017/fact]
```

Die MongoDB URI sollte hierbei natürlich für die MongoDB aus dem vorherigen Schritt gelten.

Da die REST API das Shell-Skript (vgl. chapter 17) verwendet, müssen die entsprechenden Umgebungsvariablen auch gesetzt werden. Alternativ können diese als weitere Parameter beim Jar-Datei-tart übergeben werden.

Nach dem Start der REST API ist diese unter dem Port *8080* (falls kein eigener Port angegeben wurde) auf dem Server zu erreichen.

Zum Beenden der REST API kann im Prozess einfach *Strg + C* gedrückt werden.

Damit die REST API im Hintergrund gestartet wird und auch läuft, wenn man nicht mehr mit dem Server verbunden ist, empfiehlt es sich das Programm *screen* zu verwenden. Dieses Werkzeug startet mehrere virtuelle Sitzungen in einer Verbindung und erlaubt es, diese von der eigentlichen Verbindung zu trennen und wieder aufzunehmen.

16.2.2 Docker

Alternativ zum Starten der Jar-Datei über den klassischen Weg kann auch Docker genutzt werden. Hierzu ist es erforderlich, dass Docker bereits auf dem Server installiert ist.

MongoDB Hierzu kann einfach das offiziell MongoDB Image von Docker Hub genommen werden. Dieses Image ist auch ausreichend dokumentiert.

REST API Zur REST API gibt es ein Dockerfile, welches es erlaubt ein Docker Image für die REST API zu erstellen.

Dazu ist es zunächst nötig die REST API mittels *mvn clean package* zu bauen. Anschließend sollte mit dem Befehl *cd docker* ins entsprechende Unterverzeichnis gewechselt werden. Als nächste Schritte müssen die notwendigen Komponenten vorbereitet werden: Durch *cp ../target/rest-api-1.0-SNAPSHOT.jar .*, *cp ../../streams-pg594/streams-submit.sh .* und *mkdir hadoop_conf*. Ins letzte Verzeichnis muss nun die Hadoop Konfiguration, also z.B. die *core-site.xml*, kopiert werden. Nun kann *docker build -t rest-api:latest .* aufgerufen werden um das Image zu bauen.

Abschließend muss dann dieses Image an den Server verschickt werden, da es nicht auf Docker Hub zu finden ist und somit nicht einfach von Docker heruntergeladen werden kann. Ist das Image auf dem Server verfügbar kann die REST API mit *docker run -p 8080:8080 rest-api:latest [–server.port=8080] [–mongodb=mongodb://ls8cb01.cs.uni-dortmund.de:27017/fact]* gestartet werden und ist unter *http://localhost:8080* dann erreichbar.

Kapitel 17

Shell-Script

Im Cluster des Lehrstuhls läuft Spark auf YARN, einem Tool zur Ressourcenverwaltung in Rechenclustern. Mit dem Shell-Kommando `spark-submit` können Spark-Applikationen YARN als Jobs übergeben werden, sodass sie mit zu spezifizierenden Ressourcen (Anzahl Cores, Hauptspeicher-Volumen, benötigte Dateien) ausgeführt werden.

Um YARN einen Spark-Job zu übergeben, muss `spark-submit` mit einigen Parametern (Ressourcen, auszuführende Datei, zu verwendende XML-Spezifikation) aufgerufen werden. Zudem muss sichergestellt sein, dass die gewünschte XML-Spezifikation im HDFS vorhanden ist. Um dem Benutzer die manuelle Spezifikation dieser Parameter und das Hochladen der XML zu ersparen, haben wir ein recht umfangreiches Shell-Script geschrieben, das diese Aufgaben übernimmt. Listing 17.1 stellt die Verwendung des Scriptes vor.

```
1  Usage: ./streams-submit.sh [options] <xml file>
2
3  Options:
4    --num-executors NUM          Number of executors
5    --driver-memory NUMg         GB of memory in driver
6    --executor-memory NUMg       GB of memory in executors
7    --executor-cores NUM         Number of cores per executor
8    --driver-cores NUM           Number of cores per driver
9    --max-result-size NUMg       GB of the result of batchProcess and
10                                distributedProcess
11   --name STRING                Name of the job displayed in YARN
12   --streams-jar STRING         Path to the dependecies jar
13   --hdfs-root STRING           URL of the hadoop cluster
14   --nowait                     Exit directly after the app is accepted
15
16 Example:
17   ./streams-submit.sh --driver-memory 4g example.xml
```

Listing 17.1: Verwendung des Shell-Scripts zur Ausführung im Cluster

Für jede der beschriebenen Optionen sind sinnvolle Standartwerte gesetzt. So wird etwa angenommen, dass die Jar mit den Abhängigkeiten im Hadoop Cluster unter einem fest definiertem Pfad zu finden ist. Für Entwicklungszwecke kann jedoch auch eine abweichende Jar-Datei mit der Option *–streams-jar* spezifiziert werden. Dies ist z.B. immer dann notwendig, wenn neue Streams getestet werden sollen.

Das Script prüft zunächst, ob alle Systemvariablen auf der ausführenden Maschine korrekt gesetzt sind. Nur so ist sichergestellt, dass `spark-submit` korrekt arbeitet. Dann wird ein temporäres Verzeichnis im HDFS-Home-Directory des Hadoop-Benutzers angelegt, in welches die lokal vorliegende XML kopiert wird. In die temporäre Kopie wird ein Zeitstempel in den Dateinamen geschrieben, um Konflikte zu verhindern.

Sind alle diese Vorarbeiten erledigt, kann `spark-submit` aufgerufen werden. Für die verwendeten Ressourcen bestehen niedrige Standard-Werte (2 Executor, 2GB Speicher pro Executor, ...), welche das Cluster nicht auslasten sollen. So können mehrere Entwickler gleichzeitig testen. Bei nicht zu aufwändig gestalteten Test-Konfigurationen reichen diese Ressourcen üblicherweise aus. Für aufwändigere Berechnungen können dem Script jedoch auch einige der `spark-submit`-Parameter (siehe „Options“ in Listing 17.1) übergeben werden. Es leitet diese weiter, sodass mehr Ressourcen verwendet werden.

Am Ende der Ausführung räumt das Script auf. Es löscht dazu die temporär verwendete XML-Konfiguration aus dem HDFS.

Kapitel 18

Web-UI

Neben dem Shell-Script ist während des zweiten Semesters eine Weboberfläche erstellt worden, mit der sich nicht nur Jobs komfortabel ausführen lassen, sondern auch das Managen von Jobs im Cluster vereinfachen lässt. Diese steht im direkten Zusammenhang mit der REST-API und wird als Teil dieser mitgeschickt. In diesem Kapitel soll nun die Bedienung der Oberfläche beschrieben werden.

Der gesamte Quellcode befindet sich im Order *rest-api*, wo sich auch das Dokument *README.md* befindet. Darin enthalten sind auch Informationen zur Installation per Standalone-jar oder per Docker, sowie die Systemvoraussetzungen zum Testen und Weiterentwickeln der beiden Komponenten.

Sobald die REST-API gestartet ist, kann die Weboberfläche über die Root-URL / der REST-API aufgerufen werden. Dort werden verschiedene Routen angeboten, die sich auf die folgenden Sektionen beziehen.

URL	**Aufgabe**
/#/config	Konfiguration der Hadoop URL
/#/jobs	Erstellen, starten und schedulen von Jobs
/#/events	Testen von Filter-Ausdrücken
/#/swagger	REST-Dokumentation

Daneben befinden sich in der rechts-oberen Ecke, wie beispielsweise auf Figure 18.1 zu sehen, zwei Navigationspunkte, die den Nutzer direkt zur Übersichtsseite des YARN-Clusters und die Startseite der Hadoop Weboberfläche weiterleitet.

18.1 Konfiguration

Zunächst sollte die Basis-URL von Hadoop konfiguriert werden, da der Parameter an das Shell-Script weitergegeben wird. Standardmäßig wird der Wert *hadoop.baseUrl* aus der

Abbildung 18.1: Konfigurationsseite der Web-UI

Konfigurationsdatei *application.properties* genutzt, dessen Wert zunächst auf

hdfs://s876cn01.cs.uni-dortmund.de:9000

gesetzt ist. Sollte sich die URL ändern, muss sie entsprechend über die Oberfläche über den Navigationspunkt *Config* (s. Figure 18.1) angepasst werden, da Jobs ansonsten nicht mehr korrekt gestartet werden können.

18.2 Starten und Managen von Jobs

Das Managen von Jobs beinhaltet das Erstellen, Ändern, Konfigurieren und Starten von Jobs. Hierfür bietet die Web-UI ein View an, das in Figure 18.2 abgebildet ist. Unter ❶ wird eine Liste aller gespeicherten Jobs angezeigt.

Abbildung 18.2: Interface zum Managen von Jobs

Soll ein neuer Job erstellt werden, muss unter ❷ die aus dem Projekt generierte *streams-pg594.jar* mit den entsprechenden Klassen per Drag-and-Drop oder per Klick ausgewählt werden. Dabei ist darauf zu achten, dass die gepackte jar-Datei die Klassen beinhalten muss, die im XML verwendet werden. Andernfalls schlägt der Job fehl. Anschließend wird in das vorgegebene Feld in ❸ die Definition des Jobs im XML-Format eingetragen. Der Editor zeigt etwaige Syntaxfehler an. Durch das Klicken auf den angezeigten Link *Show configuration* öffnet sich ein Formular (siehe Figure 18.3), um den Job näher zu konfigurieren. Die Parameter entsprechen den Argumenten des Shell-Scripts und haben den selben Effekt. Schließlich kann der Job über den Button *Start Job* in ❹ gestartet werden. Ein Popup informiert darüber, ob der Job erfolgreich gestartet wurde.

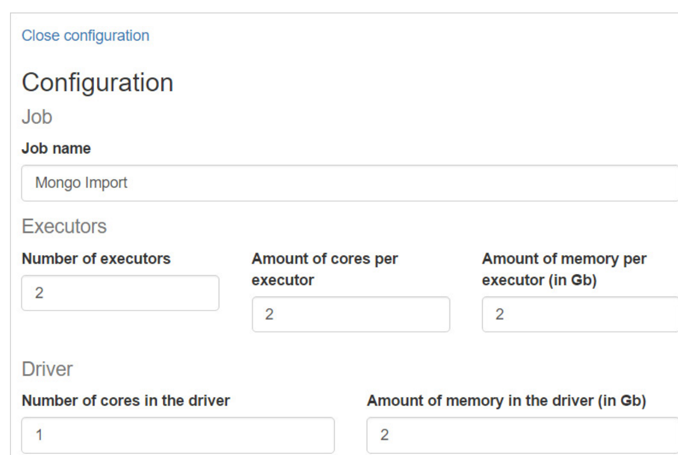


Abbildung 18.3: Konfiguration eines Jobs

Das Drücken des *Save*-Buttons speichert das Tripel aus Jar, XML und Konfiguration in der MongoDB, sodass bei einer erneuten Ausführung die Jar-Datei nicht noch einmal hochgeladen werden muss. Der entsprechende Eintrag erscheint dann in der Auflistung unter ❶. Es wird weiterhin empfohlen, einen aussagekräftigen Namen für den Job zu wählen.

Alle gestarteten Jobs lassen sich in der Übersicht anzeigen, die über den Tab *Job History* erreichbar ist. Der Inhalt, der auf Figure 18.4 zu sehen ist, listet alle Jobs auf, die gestartet wurden. Neben dem selbst gewählten Namen des Jobs werden auch Informationen zum Zustand und Fortschritt des Jobs angezeigt. Ein Klick auf den *Refresh*-Button aktualisiert die Liste der Jobs. Neben jedem Eintrag befindet sich ein Link *Details*, der auf die YARN-Übersichtsseite des Jobs verweist, wo genauere Informationen abzulesen sind, wie etwa einem Stacktrace bei einem fehlgeschlagenem Job.

PG594 Jobs Events Config REST API Hadoop YARN

Start a job Job History Tasks

Refresh

Job ID	Name	Started	Progress	State	Final Status	
application_1467202867577_0020	demo	2016/07/06 at 13:33:19	100.0%	FINISHED	SUCCEEDED	Details
application_1467202867577_0019	Demo123	2016/07/06 at 12:20:17	100.0%	FINISHED	FAILED	Details
application_1467202867577_0022	demo	2016/07/06 at 13:36:28	100.0%	FINISHED	SUCCEEDED	Details
application_1467202867577_0021	demo	2016/07/06 at 13:33:44	100.0%	FINISHED	SUCCEEDED	Details
application_1467202867577_0024	demo	2016/07/06 at 13:48:32	100.0%	FINISHED	FAILED	Details
application_1467202867577_0023	demo	2016/07/06 at 13:41:39	100.0%	FINISHED	SUCCEEDED	Details

Abbildung 18.4: Auflistung aller gestarteten Jobs

18.3 Schedulen von Jobs

Manchmal ist es hilfreich, einen Job zu einem bestimmten Zeitpunkt immer wieder auszuführen. So könnte man etwa den Import neu aufgenommener Events in die Datenbank manuell noch vor Arbeitsbeginn ausführen lassen. Für diesen Zweck bietet die Web-UI das Erstellen von immer wiederkehrenden Tasks an. Die Übersichtsseite ist über den Tab *Tasks* erreichbar und auf Figure 18.5 abgebildet.

In ❶ ist das Formular zu sehen, mit dem ein neuer Tast erstellt wird. Ein Task ist definiert

PG594 Jobs Events Config REST API Hadoop YARN

Start a job Job History Tasks

1 Create a new task

Name

Job

Cron Expression

Task is enabled.

Create Task

2 All tasks

Disabled

Mongo Import

0 30 7 1/1 * ? *

3 Enable

Delete

Abbildung 18.5: Übersichtseite der Tasks

durch seinen Namen, den Job der ausgeführt werden soll und einem Cron-Ausdruck, der bestimmt, in welchem Intervall der Job ausgeführt wird. So bedeutet der Ausdruck „*0 30 7 1/1 * ? **“ etwa, dass ein Job jeden Tag um 7:30 Uhr gestartet werden soll. Zusätzlich kann eingestellt werden, ob der Task direkt aktiviert werden soll. Durch das Klicken auf den Button *Create Task* wird der Task erstellt und erscheint in der Liste ❷.

Unter dem Menü ❸ können Tasks aktiviert oder deaktiviert sowie vollständig gelöscht werden. Falls ein aktiver Job aus der Datenbank entfernt wird, wird auch der entsprechende Schedule beendet. Gleiches gilt, falls ein Job gelöscht wird, der von Tasks referenziert werden.

Weiterhin ist anzumerken, dass nach einem erwarteten oder unerwarteten Beenden der Anwendung die Tasks nicht erneut manuell aktiviert werden müssen, da eine Routine in der Anwendung dafür sorgt, dass aktive Tasks beim Start der Anwendung automatisch gescheduled werden.

18.4 Testen von Filtern

Möchte man über die REST-API Events filtern, etwa mit dem dafür vorgesehenen *RESTfulEventStream*, muss ein Filterausdruck angegeben werden, dessen Syntax bereits in subsection 8.2.2 beschrieben wurde. In der Web-UI ist es möglich, diese Filterausdrücke zu testen. Die entsprechende Seite ist auf Figure 18.6 zu sehen. In dem dafür vorgesehen Feld wird ein Filterausdruck erwartet. Ein Klick auf dem Button *Count Events* führt diesen Ausdruck aus und liefert die Anzahl der gemachten Events zurück, was dann für den Filterausdruck

night.eq(20130801).and(eventNum.lt(10)).and(eventNum.gt(0))

wie in Figure 18.7 aussieht. Der Ausdruck filtert alle Events der Nacht des 01.08.2013 mit einer Eventnummer zwischen 0 und 10.

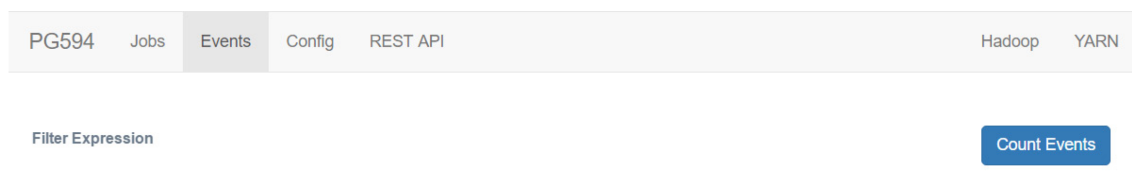


Abbildung 18.6: Übersichtseite des Event

Abbildung 18.7: Übersichtseite des Event Ergebnis

18.5 Einsehen der REST-API Dokumentation

Wie bereits in subsection 8.1.3 angesprochen, sind die Schnittstellen der REST-API mithilfe von *Swagger* dokumentiert worden. Über den Menüpunkt *REST API* ist diese, wie beispielhaft auf Figure 18.8 dargestellt, erreichbar. Dort sind alle Schnittstellen der REST-API mitsamt einer Beschreibung aufgelistet.

Sollte sich durch die Weiterentwicklung eine neue Schnittstelle ergeben, muss diese in der Datei *rest-api/src/main/resources/public/swagger.json* gemäß der Spezifikation [74] manuell definiert werden, damit sie in der Oberfläche auftaucht.

Abbildung 18.8: Konfigurationsseite der Web-UI

Kapitel 19

Maschinelles Lernen mit TELEPhANT

Nachdem in den vorherigen Kapiteln die Voraussetzungen für den Einsatz von `TELEPhANT` geklärt und das ShellScript sowie die Web-Oberfläche erläutert wurden, geht es im Folgenden um die XML-Gestaltung in Bezug auf das maschinelle Lernen. In diesem Kapitel wird auf die Besonderheiten von Apache Spark hingewiesen, welche im Umgang mit den von Spark ML bereitgestellten Lernverfahren beachtet werden sollten. Außerdem wird ein ausführliches Beispiel für das Modelltraining, die Evaluation und eine Parameterstudie gegeben. Abschließend folgt eine Erläuterung des in `TELEPhANT` enthaltenen *TreeParsers* für die genaue Analyse der von Spark ML erzeugten Baummodelle.

19.1 Datenaufbereitung

Zunächst soll es um die Datenaufbereitung gehen. Spark ML stellt sehr klare Anforderungen an die zu verarbeitenden Daten, sodass in den meisten Fällen Aufbereitungsschritte in der Pipeline nötig sind.

Numerische Merkmale Eine Voraussetzung für die korrekte Verarbeitung ist, dass in den Daten nur numerische Merkmale verwendet werden. Enthalten die gewünschten Daten allerdings kategorische Merkmale, müssen diese in passende numerische Merkmale konvertiert werden. Dabei ist zu beachten, dass diese Aufgabe keineswegs trivial ist. Nummeriert man beispielsweise alle Strings im Wertebereich einfach durch, kann es sein, dass einige Strings von Algorithmen als nahe beieinander liegend oder benachbart erkannt werden, weil ihre Zahlenwerte nahe zueinander sind. Dies muss aber nicht der Fall sein, wenn man nur die ursprünglichen Strings betrachtet. Daher ist die gemeinsame Verwendung des Estimators *StringIndexer* und des Transformers *OneHotEncoder* im erste Pipeline-Schritt empfehlenswert, wenn man kategorische Merkmale verwenden möchte.

Der *StringIndexer* bildet die Strings einer Spalte auf die Zahlenwerte 0.0 bis zur Anzahl ihrer verschiedenen Strings ab. Dabei werden die Zahlenwerte nach Häufigkeit absteigend vergeben, der häufigste String erhält daher den Zahlenwert 0.

```
1   <estimator stage="StringIndexer" inputCol="category"
       outputCol="categoryIndexed" />
```

Listing 19.1: Anwendung eines StringIndexer-Estimators

In Listing 19.1 ist die Verwendung des *StringIndexers* aufgeführt. Das folgende Beispiel für die Umwandlung kategorischer Merkmale in Zahlenwerte ist dem Apache Spark ML-Guide [6] entnommen und dient zur Demonstration:

id	category
0	a
1	b
2	c
3	a
4	a
5	c

$\xrightarrow{StringIndexer}$

id	category	categoryIndexed
0	a	0.0
1	b	2.0
2	c	1.0
3	a	0.0
4	a	0.0
5	c	1.0

Verwendet man nur den *StringIndexer* ergibt sich genau das Problem, dass nun die numerischen Kategorien a und c näher beieinander liegen als a und b. Deswegen folgt nun der Transformer *OneHotEncoder*, welcher eine Spalte mit Indizes in eine Spalte mit Binärvektoren umwandelt. Diese Binärvektoren enthalten genauso viele Komponenten, wie es verschiedene Kategorien gibt, sodass genau eine Komponente für eine Kategorie steht. Für jede Zeile enthält der Binärvektor an der Komponente eine 1, welche für die Kategorie steht, zu welcher diese Zeile gehört. Alle anderen Komponenten sind 0. Da nun jede Ausprägung ihre eigene Dimension im Vektor hat, gibt es keine Ähnlichkeiten mehr zwischen den eigentlich kategorischen Merkmalen.

Der *OneHotEncoder* nutzt eine spärliche Darstellung des Vektors: Anstatt einen Vektor der Form $(0.0, 0.0, 1.0, 0.0, 0.0)$ auszuschreiben wird dieser als $(5, [2], [1.0])$ dargestellt. Im Vektor mit fünf Komponenten steht an Index 2 die 1.0, der Rest wird mit 0.0 gefüllt. Bei vielen verschiedenen Merkmalsausprägungen ist diese Darstellung platzsparender als ein Vektor mit zig Einträgen.

```
1   <transformer stage="OneHotEncoder" inputCol=
2   "categoryIndexed" outputCol="categoryVec" />
```

Listing 19.2: Anwendung eines OneHotEncoder-Transformers

In Listing 19.2 ist die XML-Verwendung in der Pipeline aufgeführt. Insgesamt können mit Hilfe dieser Kodierung kategorische Merkmale umgewandelt und in Spark ML korrekt verarbeitet werden. Zum besseren Verständnis und zur Erklärung einer weiteren Eigenheit von Spark ML ein kurzes Beispiel:

id	**category**	**categoryIndexed**
0	a	0.0
1	b	2.0
2	c	1.0
3	a	0.0
4	a	0.0
5	c	1.0

$\xrightarrow{OneHotEncoder}$

id	**categoryVec**
0	(2, [0], [1.0])
1	(2, [], [])
2	(2, [1], [1.0])
3	(2, [0], [1.0])
4	(2, [0], [1.0])
5	(2, [1], [1.0])

Auffällig ist, dass die Vektoren nur zwei Komponenten haben, aber insgesamt drei Kategorien vertreten sind. Um sicherzustellen, dass die Vektoren linear abhängig sind, bildet genau die Kategorie mit den wenigsten zugehörigen Zeilen (in diesem Fall *b*) auf den Vektor $(0.0, 0.0)$ ab, anstatt auf $(0.0, 0.0, 1.0)$. Möchte man genau die lineare Unabhängig zwischen den Binärvektoren erreichen, fügt man in die XML in Listing 19.2 zusätzlich den Paramter `dropLast="false"` ein.

Merkmalsvektor Verfügt man nun ausschließlich über Merkmale beziehungsweise Spalten mit numerischen oder booleschen Werten oder Vektoren, müssen all diese Merkmale zu einem großen Merkmalsvektor zusammengefasst werden. Die in Spark ML bereitgestellten Lernverfahren verlangen nämlich als Eingabe genau eine Merkmalsspalte. Dies kann nur geliefert werden, wenn man alle Merkmale zu einem Vektor zusammenfasst. Genau zu diesem Zweck wird der Transformer *VectorAssembler* bereitgestellt.

```
1 <transformer stage="VectorAssembler" outputCol="features"
2 inputCols=
3 "categoryVec,numberA,numberB,boolA,..." />
```

Listing 19.3: Anwendung eines VectorAssembler-Transformers

In Listing 19.3 ist ein Beispiel für die Einbindung eines solchen *VectorAssemblers* in die XML-Pipeline gegeben. Alle Spalten, die im Parameter `inputCols` angegeben werden, werden in einen Vektor zusammengefasst. Wichtig ist dabei, dass diese Spalten wirklich nur numerische, boolesche Werte oder Vektoren enthalten, sonst bricht die Prozedur mit einem Fehler ab. Die Angabe des gebildeten Merkmalsvektors wird im späteren Verlauf des Kapitels in section 19.2 genauer erläutert.

Benennung der Label Auch die Spalte mit den wahren Klassen (Labels) bei den Trainingsdaten unterliegt bestimmten Voraussetzungen: Die Klassen sollen durch ganzzahlige Double-Werte startend bei 0.0 repräsentiert werden. In unserem Fall mit zwei Klassen würde dies bedeuten, *gamma* und *hadron* als 0.0 und 1.0 darzustellen. Für binäre Klassifikationsprobleme tauchen oftmals auch die Bezeichnungen -1 und 1 für die beiden möglichen Klassen auf. Auch diese müssen auf 0.0 und 1.0 abgebildet werden. Hilfe bei dieser Aufgabe bietet wieder der oben bereits erwähnte *StringIndexer*. Um garantieren zu können, dass die Label vom Lernverfahren richtig erkannt werden, sollte der *StringIndexer* **immer** auf die Klassenspalte angewendet werden. Sollte das Label schon numerisch gewesen sein, wird dieser numerische Wert in einen String umgewandelt und von dort aus auf gewohnte Weise indiziert. Eine Einbindung des *StringIndexers* ist wie in Listing 19.1 gezeigt möglich und eine klare Empfehlung von unserer Seite. Auch wenn die Klassen schon mit 0.0 und 1.0 benannt sind, ist die Verwendung eines *StringIndexers* ratsam, um einen reibungslosen Ablauf garantieren zu können.

Sollen nach der Klassifikation die Klassenlabel in die ursprünglichen zurück konvertiert werden, steht auch dazu ein Operator zur Verfügung. Dieser Schritt kann jedoch nicht in die Pipeline integriert werden, da für die korrekte Ausführung eine Datenstruktur notwendig ist, welche erst bei der Ausführung des in der Pipeline definierten *StringIndexers* erzeugt wird. Daher sollte der Operator *IndexToStringConversion* erst **nach** der Pipeline und dem Training des Pipeline-Modells (siehe Abschnitt 19.2) aufgerufen werden.

```
1 <pipeline modelName="model"> ... </pipeline>
2 <!-- load test or raw data and apply model -->
3 <stream.pg594.operators.IndexToStringConversion modelName="
    model" inputCol="prediction" outputCol="predictedLabel"/>
```

Listing 19.4: Umkehrung eines StringIndexers

Listing 19.4 zeigt eine beispielhafte XML-Gestaltung für die Integration einer Rückkonvertierung.

Weitere Möglichkeiten zur Datenvorverarbeitung Spark ML bietet nicht nur Transformer und Estimator zur Aufbereitung der Daten an, damit sie die von den bereitgestellten Lernalgorithmen gestellten Voraussetzungen erfüllen können. Mithilfe der implementierten Stages kann man auch eigene Ansprüche und Wünsche an die Daten umsetzen. Dazu sei auf die Seite `http://spark.apache.org/docs/latest/ml-features.html` verwiesen, auf welcher alle Stages für die Merkmalsanpassung vorgestellt werden.

Abschließend ist noch zu sagen, dass alle Aufbereitungs- und Vorverarbeitungsschritte, die in einer Pipeline aufgeführt werden, auch auf die späteren Test- beziehungsweise Klassifikationsdaten angewendet werden. Daher ist keine neue Aufbereitung der zu klassifizierenden

Daten nötig. Solange dieselben Spalten mit denselben Namen wie im Trainingsdatensatz existieren, ausgenommen natürlich die Spalte mit den Klassen, treten keine Probleme auf. Wurde beim Design der Pipeline darauf geachtet, dass kategorische in numerische Merkmale konvertiert wurden und dass pro Zeile ein gesamter Merkmalsvektor gebildet wurde, werden diese Schritte automatisch auch auf die zu klassifizierenden Daten angewendet, bevor sie wirklich durch das Modell klassifiziert werden. Es ist daher nur essentiell darauf zu achten, dass auch dieselben Merkmale mit denselben Namen vorhanden sind wie in den Trainingsdaten.

19.2 Modelltraining und Evaluation

Nachdem die nötigen Schritte zur Datenaufbereitung erläutert wurden, geht es in diesem Abschnitt um das Herzstück des maschinellen Lernens, nämlich den Einsatz des gewünschten Lerners und dessen Evaluation. Dieser Abschnitt ist in drei Unterabschnitte gegliedert. Zunächst werden Training und Klassifikation eines Lernverfahrens thematisiert, anschließend die Evaluation eines Modells und die Möglichkeiten der Durchführung von Parameterstudien.

19.2.1 Training und Klassifikation

Zu Beginn des Trainings ist es ratsam, die gegebenen Daten in Trainings- und Testdatensatz zu splitten. Meist wird ein Verhältnis von circa 70 zu 30 Prozent gewählt. Auf dem größeren Trainingsdatensatz wird das Pipeline-Modell trainiert, während die restlichen Daten zur Evaluation des Lerners genutzt werden.

```
1 <stream.pg594.operators.SplitDataFrame ratio="0.3" newName="
    testData"/>
```

Listing 19.5: Splitten in Trainings- und Testdaten

In Listing 19.5 ist zu sehen, wie diese Aufteilung im XML umgesetzt werden kann. Der Operator *SplitDataFrame* schneidet in diesem Fall 30% des geladenen DataFrames ab und speichert diesen unter dem Namen *testData*. Auf dem verbliebenen DataFrame wird das Pipeline-Modell trainiert.

In der Pipeline stehen zuerst die verschiedenen Aufbereitungs- und Vorverarbeitungsschritte, wie sie im vorangegangenen Abschnitt erklärt wurden. Anschließend können ein oder mehrere Lernverfahren definiert werden, welche trainiert werden sollen. Dabei wird in Klassifikation und Regression unterschieden. Im Folgenden werden die von Spark ML bereitgestellten Lernverfahren aus den jeweiligen Kategorien kurz vorgestellt.

Klassifikation Für die Klassifikation werden folgende Lernverfahren bereitgestellt: Logistische Regression, Entscheidungsbaum, Random Forest, Gradient-Boosted Trees, Multilayer Perceptron, One-vs-Rest-Klassifikator und Naive Bayes. Für die Gamma-Hadron-Separation empfehlen sich besonders die baumbasierten Lernverfahren [6].

Regression Für die Regression werden folgende Lernverfahren bereitgestellt: Lineare Regression, Entscheidungsbaum, Random Forest, Gradient-Boosted Trees, Survival Regression und Isotonic Regression [6]. Für die Energieschätzung haben sich in unseren Tests erneut die baumbasierten Methoden am besten verhalten.

Die jeweiligen Parameter lassen sich aus der Spark ML-API [7] ablesen. Wie dies genau funktioniert, wird am Beispiel des *RandomForestClassifiers* [8] erläutert: In der API befindet sich eine Zusammenfassung aller Methoden. Alle `set`-Methoden beschreiben Parameter, die gesetzt werden können. Soll ein bestimmter Parameter in der XML gesetzt werden, muss das `set` weggelassen und der erste Buchstabe des Parameters kleingeschrieben werden.

```
1 <estimator stage="RandomForestClassifier" maxDepth="20"
    numTrees="30" featuresCol="features" labelCol="
    labelIndexed" predictionCol="prediction"/>
```

Listing 19.6: Definition eines Random Forest

In Listing 19.6 wird ein *RandomForestClassifier* in der Pipeline definiert. In der API zu diesem Klassifikator kann nachgelesen werden, dass dieser unter anderem über Methoden `setMaxDepth(int value)` und `setNumTrees(int value)` verfügt. In Klammern wird angegeben, wie viele Parameter von welchem Datentyp benötigt werden. Wie im Beispiel gezeigt können die gewünschten Parameter in der XML beschrieben werden.

Besonders wichtig ist es hier, immer die Parameter *featuresCol* und *labelCol* anzugeben, damit das jeweilige Lernverfahren weiß, auf welchen Merkmalen und Klassen er trainieren soll. Diese Parameter existieren unabhängig davon, ob es sich um ein Klassifikations- oder Regressionsverfahren handelt. Außerdem kann der Parameter *predictionCol* angegeben werden, um die Spalte zu benennen, in welcher die Anwendungsergebnisse des Lerners gespeichert werden. Der Standardwert für diesen Parameter ist *prediction"*.

Auch die Pipeline enthält Parameter. Neben dem Modellnamen, welcher relativ selbsterklärend ist, spielt der Parameter `automaticTraining` eine wichtige Rolle. Ist dieser auf `true` gesetzt, wird das Pipeline-Modell automatisch trainiert und kann danach abgespeichert werden und neue Daten klassifizieren. Es gibt jedoch auch Anwendungsfälle, in denen das Modell nicht sofort trainiert werden sollte und der Parameter auf `false` gestellt werden kann. Anwendungsfälle dafür sind beispielsweise eine auf die Pipeline folgende

Kreuzvalidierung oder ein gewünschtes Training mit jeweils anderen Daten. Beide Fälle werden am Ende dieses Unterabschnittes besprochen. Zunächst folgt ein XML-Ausschnitt mit zugehöriger Erläuterung für das automatische Training:

```
1 <pipeline modelName="model" automaticTraining="True">
2 <!-- beliebige Transformer und Estimator zur Aufbereitung -->
3 <estimator stage="RandomForestClassifier" numTrees="30"
    featuresCol="features" labelCol="labelIndexed"/>
4 </pipeline>
5 <stream.pg594.operators.LoadDataFrame name="testData" />
6 <stream.pg594.operators.ApplyModel modelName="model" />
7 <stream.pg594.operators.ExportModelToBinaryFile modelName="
    model" url="hdfs://..." />
```

Listing 19.7: Training und Anwendung eines Modells

Wie in Listing 19.7 zu sehen ist, wird zunächst eine Pipeline gebaut, welche nach beliebigen Aufbereitungsschritten einen Klassifikator trainiert, in diesem Fall einen *RandomForestClassifier* mit 30 Bäumen. Anschließend wird der Testdatensatz geladen. Alternativ können in diesem Schritt, wenn nicht getestet werden soll, die Rohdaten geladen werden, welche klassifiziert werden sollen. Das Modell, welches am Ende der Pipeline automatisch trainiert wurde, wird auf die Testdaten angewendet und an einen beliebigen Ort im HDFS exportiert, sodass es für spätere Anwendungen wieder geladen werden kann. Das Modell wird als `modelName.model` abgespeichert und kann mit Hilfe des *ImportModelFromBinaryFile*-Operators durch Angabe der URL und des gewünschten Modelnamens wieder geladen werden.

Es wird nun der Fall betrachtet, in welchem die Pipeline nicht automatisch trainiert wird. Der erste Anwendungsfall ist eine Kreuzvalidierung, welche auf die Pipeline folgen soll.

```
1 <stream.pg594.operators.CrossValidator type="classification"
    modelName="CrossValidatorModel" folds="4" labelCol="
    labelIndexed" predictionCol="prediction" metricName="
    recall"/>
```

Listing 19.8: Kreuzvalidierung

Wie in Listing 19.8 zu sehen ist, muss bei einer Kreuzvalidierung der Typ, also Klassifikation oder Regression, angegeben werden. In diesem Fall wird eine vierfache Kreuzvalidierung durchgeführt. Dabei wird das Modell viermal auf den gleichen Daten trainiert und sofort evaluiert. Genau das Modell, welches bei den Evaluationen am besten abschneidet,

wird zurückgegeben und als *CrossValidatorModel* abgespeichert. Danach kann dieses wie gewohnt zur Klassifikation genutzt, abgespeichert und evaluiert werden. Die Kreuzvalidierung ist hilfreich, um statistische Schwankungen zu reduzieren. Aufgrund der Nutzung von *random seeds* innerhalb der meisten Algorithmen kann es durchaus vorkommen, dass sich Modelle, die mit denselben Parametern und auf denselben Daten trainiert werden, in ihrer Qualität unterscheiden.

Diese Schwankungen können mit Hilfe unserer Operatoren außerdem zu Analysezwecken beobachtet werden. Dazu gibt es den *foreach*-Operator, welcher es möglich macht, das Training eines Pipeline-Modells beliebig oft zu wiederholen, sofern `automaticTraining` auf `false` gesetzt war. Werden in jedem Durchlauf die entsprechenden Evaluationsergebnisse abgespeichert, werden die Schwankungen sichtbar. Der folgende XML-Auszug gibt einen Einblick in das Konzept des *foreach*-Operators, die Evaluation und Speicherung ebendieser Ergebnisse wird in subsection 19.2.2 diskutiert.

```
1 <stream.pg594.operators.ForEach header="i in [1,2,3,4,5]">
2 <stream.pg594.operators.LoadDataFrame name="trainingData" />
3 <stream.pg594.operators.TrainModel modelName="model" />
4 <stream.pg594.operators.LoadDataFrame name="testData" />
5 <stream.pg594.operators.ApplyModel modelName="model" />
6 <! -- evaluation -->
7 </stream.pg594.operators.ForEach>
```

Listing 19.9: Mehrfaches Training eines Modells auf denselben Daten zur Beobachtungen der statistischen Schwankungen

In Listing 19.9 wird fünfmal trainiert und getestet. Voraussetzung für das Funktionieren dieser XML ist, dass vor dem *foreach*-Operator die kompletten Daten geladen und in zwei DataFrames aufgeteilt wurden, welche unter den Namen *trainingData* und *testData* abgespeichert wurden.

Dieses abwechselnde Laden von Trainings- und Testdaten bildet eine gute Verbindung zum zweiten Anwendungsfall für ein manuelles Training: Es kann durchaus sein, dass ein Anwender das gleiche Modell mit mehreren verschiedenen Trainingsdatensätzen trainieren möchte. Auch dafür kann man einen *foreach*-Operator einsetzen und immer wieder andere DataFrames für das Training laden, sofern man diese vorher entsprechend abgespeichert hat.

19.2.2 Evaluation

Nachdem Training und Anwendung eines Modells im vorherigen Unterabschnitt ausführlich erläutert wurden, geht es im Folgenden um die Evaluation eines Modells. Zunächst werden

die Operatoren von TELEPhANT zur Evaluation vorgestellt, anschließend wird erläutert, wie Evaluationsergebnisse gespeichert werden können.

Evaluation einer binären Klassifikation Da es sich bei unserem Anwendungsfall um eine binäre Klassifikation in Gamma- und Hadronstrahlungen handelt, stellt unser Produkt Möglichkeiten zur Evaluation eines binären Klassifikators zur Verfügung. Zunächst können mithilfe des *EvaluateBinaryClassifier*-Operators die Werte TP (true positives), TN (true negatives), FP (false positives) und FN (false negatives) ausgerechnet werden. Dieser Operator berechnet auch gleichzeitig die jeweils zugehörigen Raten. Wichtig ist, dass es bei diesem Operator zwei Parameter gibt: Die Spalte mit den korrekten Klassenbezeichnungen und die Spalte mit den Klassifikationsergebnissen. Diese beiden Spalten werden miteinander verglichen und sollten deswegen vom Typ her übereinstimmen. Es ist daher ratsam bereits die indexierte Label-Spalte als Eingabe zu verwenden, da sonst zunächst eine Rückkonvertierung der Klassifikationsergebnisse zu den ursprünglichen Klassenbezeichnungen stattfinden muss. Die Ergebnisse des Operators werden in die Logdateien geschrieben und zusätzlich im Kontext abgelegt, sodass sie später komfortabel in eine CSV-Datei geschrieben werden können. Es folgt eine kurze Beispielanwendung:

```
1 <stream.pg594.operators.EvaluateBinaryClassifier labelCol="
    labelIndexed" predictionCol="prediction" />
```

Listing 19.10: Evaluation eines binären Klassifikators (erstes Beispiel)

Der zweite Operator zur Evaluation eines binären Klassifikators ist der *EvaluateBinaryClassifierRaw*-Operator, welcher zusätzliche Metriken bereitstellt. Dies sind die *Area Under ROC*, oft auch ROC-Kurve genannt, und die *Area under Precision-Recall-Curve*. Auch hier werden zwei Parameter gefordert: Die Spalte mit den korrekten Klassenbezeichnungen und die Spalte mit noch nicht normalisierten Wahrscheinlichkeiten für jede Klasse. Bei den meisten Klassifikationsproblemen (Entscheidungsbäumen, Random Forests, Logistischer Regression und Naive Bayes) erzeugt die Anwendung eines Modells nicht nur eine neue Spalte mit Namen *prediction*, sondern noch zwei weitere, nämlich *rawPrediction* und *probability*. Diese beiden Spalten enthalten jeweils einen Vektor der Länge 2 (für genau 2 Klassen bei einem binären Klassifikationsproblem). In *prediction* stehen jeweils die genauen Wahrscheinlichenkeiten dafür, dass diese Zeile Klasse 0.0 beziehungsweise 1.0 zugeordnet werden sollte. Die Wahrscheinlichkeiten sind normalisiert, sodass deren Summe genau 1 ergibt. Die Spalte *rawPrediction*, welche für die Evaluation benötigt wird, enthält diese Wahrscheinlichkeiten in noch nicht normalisiertem Zustand. Für Random Forests sind beispielsweise in den Vektorkomponenten jeweils zu sehen, wie viele Bäume im Wald für die jeweilige Klasse gestimmt haben. Nur für Klassifikatoren, welche diese “rohen”Vorhersagen ausgeben, kann der *EvaluateBinaryClassifierRaw*-Operator genutzt

werden. Auch hier werden die beiden Werte in den Log geschrieben und im Kontext abgespeichert. Außerdem sollte nach Möglichkeit wieder die bereits indexierte Spalte mit Klassenbezeichnungen genutzt werden. Eine beispielhafte Anwendung sieht folgendermaßen aus:

```
1 <stream.pg594.operators.EvaluateBinaryClassifierRaw labelCol=
    "labelIndexed" rawPredictionCol="rawPrediction" />
```

Listing 19.11: Evaluation eines binären Klassifikators (zweites Beispiel)

Natürlich kann man die Spalten *predictionCol* und *rawPredictionCol* nach Belieben bei der Definition des Klassifikators umbenennen, indem man den jeweiligen Parameter in der Pipeline setzt.

Evaluation eines Regressors Für die Energieschätzung wird eine Regression durchgeführt, deswegen stellt `TELEPhANT` auch Evaluationsmöglichkeiten für die Regression bereit. Der *EvaluateRegressor*-Operator funktioniert ganz ähnlich und benötigt zwei Parameter: Die Spalte mit der wahren Energie und die mit der geschätzten. Auf dieser Basis werden die Werte RMSE (Root-Mean Squared Error) und R^2 berechnet, ins Log geschrieben und im Kontext abgelegt. Das folgende Beispiel zeigt die XML-Anwendung:

```
1 <stream.pg594.operators.EvaluateRegressor labelCol="energy"
    predictionCol="prediction" />
```

Listing 19.12: Evaluation eines Regressors

Zum Abschluss dieses Unterabschnittes wird im Folgenden erläutert, wie die Ergebnisse aus dem Kontext in eine CSV geschrieben werden können. Dazu wird der *AppendToCSV*-Operator genutzt. Dieser nimmt zwei Parameter entgegen, zum Einen die URL, unter welcher die CSV abgelegt werden soll, zum Anderen die Werte, die in der CSV gespeichert werden sollen. Dabei hat der Anwender die Möglichkeit einen beliebigen Wert abzulegen (zum Beispiel `type=RandomForest`) oder einen Wert aus dem Kontext zu holen (zum Beispiel `AreaUnderROC`).

```
1 <stream.pg594.operators.AppendToCSV columns="type=
    RandomForest,numTrees=30,AreaUnderROC,
    AreaUnderPrecisionRecallCurve,TP,TN,FP,FN,TPrate,TNrate,
    FPrate,FNrate" url="hdfs://.../eval.csv"/>
```

Listing 19.13: Darstellen der Evaluationsergebnisse in einer CSV

Dieser in Listing 19.13 beschriebene Schritt kann durchaus mehrfach ausgeführt werden, sodass die unter der URL angegebene CSV-Datei beim ersten Zugriff erzeugt und bei danach folgenden Zugriffen nur noch erweitert wird. Mithilfe eines *foreach*-Operators wie in Listing 19.9 können so statistische Schwankungen bei der Modellerzeugung beobachtet und gesichert werden. Auch für Parameterstudien kann die Dokumentation der Ergebnisse in einer CSV-Datei vorteilhaft sein, wie der nächste Unterabschnitt zeigen wird.

19.2.3 Parameterstudie

Mit Hilfe von großen Parameterstudien kann dasselbe Lernverfahren auf verschiedenen Parametern getestet werden. Eine besonders komfortable Möglichkeit zum Design solcher Studien gibt es dafür leider nicht, dafür ist die Auswertung solcher Studien mit Hilfe des *AppendToCSV*-Operators relativ elegant.

In der Pipeline können tatsächlich beliebig viele Estimator-Stages stehen. Jeder Klassifikator wird trainiert und kann evaluiert werden. Wichtig ist bei der Verwendung von mehreren Lernverfahren in einer Pipeline allerdings die korrekte Parameterbenennung. Üblicherweise entscheiden sich die Lerner dafür, ihre Vorhersage in eine Spalte namens *prediction* zu schreiben. Für den ersten Lerner der Pipeline funktioniert das auch noch sehr gut, für den zweiten jedoch kommt es zum Problem, da diese Spalte, die er eigentlich an den DataFrame anfügen sollte, schon existiert. Es muss also darauf geachtet werden, dass die Parameter *predictionCol, rawPredictionCol* und *probabilityCol* für Klassifikatoren und lediglich *predictionCol* für Regressoren immer wieder für jeden Lerner individuell benannt werden müssen. Der folgende XML-Auszug zeigt einen möglichen Aufbau für eine Parameterstudie:

```
1 <pipeline modelName="gammaHadronModel" automaticTraining="
    true">
2 <!-- beliebige Aufbereitungsschritte -->
3 <estimator stage="RandomForestClassifier" numTrees="5"
    featuresCol="features" labelCol="IsGammaIndexed"
    rawPredictionCol="raw1" probabilityCol="prob1"
    predictionCol="IsGamma1" />
```

```
4 <!-- Trainiere N Random Forests mit jeweils unterschiedlicher
     Waldgroesse -->
5 <estimator stage="RandomForestClassifier" numTrees="50"
    featuresCol="features" labelCol="IsGammaIndexed"
    rawPredictionCol="rawN" probabilityCol="probN"
    predictionCol="IsGammaN" />
6 </pipeline>
7 <!-- Lade Testdaten und wende Modell an -->
8 <stream.pg594.operators.EvaluateBinaryClassifier modelName="
    gammaHadronModel" predictionCol="IsGamma1" labelCol="
    IsGammaIndexed" />
9 <stream.pg594.operators.EvaluateBinaryClassifierRaw modelName
    ="gammaHadronModel" rawPredictionCol="raw1" labelCol="
    IsGammaIndexed" />
10 <stream.pg594.operators.AppendToCSV columns="type=
    RandomForestClassifier,numTrees=${numTrees1},AreaUnderROC,
    AreaUnderPrecisionRecallCurve,TP,TN,FP,FN,TPrate,TNrate,
    FPrate,FNrate" url="hdfs://.../eval.csv"/>
11 <!-- Evaluiere N Random Forests und schreibe Ergebnis in CSV
    wie gezeigt -->
12 </task>
```

Listing 19.14: Parameterstudie

Zu beachten ist, dass bei der Ausführung von Listing 19.14 N Zufallswälder trainiert werden. Anschließend wird auf diesem großen Modell klassifiziert, sodass insgesamt $3 \cdot N$ Spalten angefügt werden: Für jeden Zufallswald die Spalten *rawPrediction, probability* und *prediction*. Außerdem werden bei jeder Evaluation die Ergebnisse im Kontext (also FP, TP, AreaUnderROC,...) überschrieben.

So mühsam auch der Aufbau einer solchen Studie ist, so komfortabel lassen sich anschließend die verschiedenen Evaluationsergebnisse und darauf basierend die perfekten Parameter für den Lerner aus der CSV-Datei ablesen. Mit Hilfe der `property`-Tags müssen die Parameter nicht hart in die Pipeline kodiert werden, so wird wenigstens ein wenig Flexibilität in den Studien erreicht. Im Folgenden wird nun der in `TELEPhANT` enthaltene *TreeParser* vorgestellt, mit welchem baumbasierte Modelle weiterführend analysiert werden können, wenn die bis hier vorgestellten Evaluationsmethoden nicht ausreichen.

19.3 Der TreeParser

In unseren Experimenten stellte sich das Verwenden von Random Forests als sehr erfolg-

reiche Methode zur Klassifikation heraus. Da Spark ML es erlaubt, gewonnene Random-Forest-Modelle in Textform zu speichern, kann der **TreeParser** benutzt werden um die Textdatei für weitere Analysen wieder in eine Datenstruktur zu überführen.

19.3.1 Struktur der Lernbäume

Betrachten wir zunächst als Beispiel einen Auszug aus einem beliebigen Random Forest. Jeder Baum des Waldes beginnt mit der Zeile "Tree X (weight Y.Z)", dann folgt die Wurzel des Baumes in der nächsten Zeile. Die Knoten des Baumes sind entweder Entscheidungsknoten, an denen eine Bedingung geprüft und dann eines der Kinder gewählt wird, oder Vorhersageknoten, die die Blätter des Baumes darstellen und eine Klasse wählen.

In unserem Fall ist die Wurzel des Baumes ein Entscheidungsknoten, der zwischen den Fällen "Ist Feature 30 kleiner als 49.42404654253062" oder "Ist Feature 30 größer als 49.42404654253062", entscheidet. Zu dem **If** eines Knotens gehört also immer ein **Else**, das allerdings meistens sehr viel später im Baum folgt. Leider sorgt dieses Format dafür, dass die Bäume für Menschen relativ schwierig zu lesen sind, dafür sind sie mit dem **TreeParser** umso leichter zu verarbeiten. Betrachtet man die **If**-Zweige als rechte und die **Else**-Zweige als linke Kinder des Knotens, so kann der Baum innerhalb eines Durchlaufs analysiert werden. Dazu werden, von der Wurzel ausgehend, zunächst alle **If**-Zeilen rekursiv als rechte Kinder des Vorgängers in den Baum eingefügt, bis eine Vorhersage-Zeile auftritt. Diese stellt den Rekursionsanker dar. Nach der **Else**-Zeile folgt ein rekursiver Aufruf für den linken Teilbaum des letzten Entscheidungsknotens. Ist dieser abgearbeitet folgt das **Else**, und damit der linke Teilbaum des nächsthöheren Knotens.

```
1  Tree 0 (weight 1.0):
2    If (feature 30 <= 49.42404654253062)
3     If (feature 17 <= 0.082960642675512)
4      Predict: 0.0
5     Else (feature 17 > 0.082960642675512)
6      Predict: 1.0
7    Else (feature 30 > 49.42404654253062)
8      Predict: 0.0
9  Tree 1 (weight 1.0):
10   If (feature 30 <= 13)
11     Predict: 0.0
12    Else (feature 30 > 13)
13     Predict: 1.0
```

Listing 19.15: Auszug aus einem kleinen RandomForest

19.3.2 Die Parser-Klasse

Der TreeParser kann nun genutzt werden, um eine genauere Analyse des Random Forests durchzuführen. Dazu wird der Baum zunächst wieder, wie beschrieben, in eine Datenstruktur überführt. Nach dem Aufruf der Main-Methode der Parser Klasse, legt diese ein Array mit den Bäumen des Random Forests an, die jeweils ihren Wurzel-Knoten als auch Gewicht und Nummer enthalten. Außerdem können diese erweitert werden, um in ihnen weitere Daten, die für die Analyse wichtig sind, zu speichern. Die Knoten selber unterteilen sich in **DecisionNodes** und **PredictionNodes**, die den Entscheidungs- und Vorhersageknoten entsprechen. Das Einlesen des Baumes geschieht, wie bereits beschrieben, rekursiv, die **DecisionNodes** erstellen also ihre Kinderknoten, sobald sie erstellt werden. Der Parser muss lediglich mit der ersten Zeile des jeweiligen Baumes den Wurzelknoten erstellen. Ist die TreeList erstellt worden, kann nun die eigentliche Analyse beginnen.

19.3.3 CombinedTreeFeatures

Zuletzt betrachten wir noch **CombinedTreeFeatures**, ein Beispiel für ein Analyseverfahren das auf dem durch den Parser gewonnenen Random Forest durchgeführt werden kann. Angenommen, wir wollen herausfinden, wie häufig ein Feature pro Baum oder Wald vorkommt. Hierzu erweitern wir die Tree Klasse um die Klasse **Tupel**, die zwei Integer enthält sowie eine HashMap und ein Array aus Tupeln.
Wird nun die Methode `featureCounting()` des Baumes aufgerufen, zählt dieser zunächst rekursiv alle Features die ihn ihm vorkommen mit Hilfe der HashMap. Dabei sind die **Feature-Nummern** die Keys, während die Values die Häufigkeit des Features repräsentieren. Sind die Features in der HashMap gesammelt werden sie dann zwecks besserer Sortierbarkeit in das Array übertragen, welches dann sortiert wird. Ebenso können wir jetzt im Parser die Methode `combinedTreeFeatures()` verwenden, um die HashMaps des gesamten Random Forest nach demselben Prinzip zu kombinieren und in einen Array zu übertragen.
Es ist also zu erkennen, dass die rekursive Datenstruktur, in der der Random Forest nun gespeichert ist, es relativ problemlos erlaubt, neue Analyseverfahren zu implementieren. Somit ist die Grundlage gegeben, auch interessantere Analyseverfahren als das einfache Zählen von Features durchzuführen.

Anhang A

Liste der Operatoren

Im Folgenden werden alle verwendeten Operatoren mit Name und Funktion aufgelistet. Die Namen der zu verwendenden Parameter sind dabei jeweils in Anführungszeichen angegeben.

AppendToCSV	Fügt einen String (”columns”) an eine CSV-Datei (”url”) an.
ApplyModel	Lädt das angegebene Modell (”modelName”) und wendet es auf den InputFrame an.
ComputeAbsoluteError	Berechnet den absoluten Fehler aus zwei vorgegebenen Spalten (”correctCol” und ”predictionCol”) und fügt ihn als zusätzliche Spalte (”outputCol”) hinzu
CountRows	Zählt die Reihen des DataFrames
CountRowsWhere	Zählt die Reihen des DataFrames, an denen eine Bedingung (”condition”) gilt.
CrossValidator	Führt entweder eine Klassifikation oder eine Regression (”type”) mit wahren (”labelcol”) und vorhergesagten (”predictionCol”) Werten durch. Dabei wird die Metrik (”metricName”) verwendet.
EvaluateBinaryClassifier	Evaluiert eine binäre Klassifikation mit wahren (”labelCol”) und vorhergesagten(”predictionCol”) Werten.
EvaluateRegressor	Evaluiert eine Regression mit wahren (”labelCol”) und vorhergesagten(”predictionCol”) Werten.
ExplainPipeline	Gibt die aktuelle Pipeline mit Erklärungen aus.
ExportDataFrame	Speichert den DataFrame auf die gewünschte Art (”saveMode”) im Dateisystem (”url”).

ExportDataFrameParquet	Speichert den DataFrame als Parquet-File im Dateisystem (”url”).
ExportModel	Speichert ein Modell (”modelName”) im Dateisystem (”url”).
ExportModelToBinaryFile	Speichert ein Modell (”modelName”) als binäre Datei im Dateisystem (”url”).
FilterDataFrame	Entfernt alle Zeilen aus dem DataFrame, die eine Bedingung (”condition”) nicht erfüllen.
ForEach	Führt die enthaltenen Prozessoren mehrfach aus. Der Parameter ”header” muss folgende Form haben: „%variable% in [val1,val2,...,valn]“. Beispiel: header=„i in [10,20,30]“
ImportModel	Importiert ein Modell (”modelName”) aus dem Dateisystem (”url”).
ImportModelFromBinaryFile	Importiert ein als binäre Datei gespeichertes Modell (”modelName”) aus dem Dateisystem (”url”).
IndexToStringConversion	Wenn ein Modell (”modelName”) mit StringIndexer verwendet wurde, übernimmt dieser Operator die Rückübersetzung der Spalte ”inputCol” zu ”outputCol”.
LoadDataFrame	Lädt einen DataFrame (”url”) aus dem Dateisystem.
LogTimestamp	Schreibt eine Nachricht (”message”) und die aktuelle Zeit in Millisekunden in den Log.
PrintDataFrame	Gibt den Inhalt des DataFrames aus.
RenameColumn	Ändert den Namen einer Spalte im DataFrame vom alten (”oldName”) in einen neuen (”newName”).
SafeDataFrame	Speichert den DataFrame im DateiSystem (”url”).
SelectColumnsSQL	Wählt die vorgegeben Spalten (”columns”) aus. Die einzelnen Spalten sind mit Kommata zu trennen, z.B. (col1, col2, col3), zusätzlich können SQL Ausdrücke verwendet werden.
SplitDataFrame	Erstellt eine Partition des DataFrame und speichert diese unter neuem Namen (”newName”). Die Größe (”ratio”) muss festgelegt werden.
StartTimeMeasurement	Legt einen Zeitmesser (”timerName”) an und setzt seine Startzeit.
StopTimeMeasurement	Gibt die Differenz zwischen der aktuellen Zeit und dem Start-Zeitmesser des selben Namens (”timerName”) an.
SwitchDataFrame	Speichert gegebenfalls den aktuellen DataFrame (”safeCurrentDataFrameAs”), und lädt dann einen neuen (”loadDataFrame”).
TrainModel	Wendet die aktuelle Pipeline auf den DataFrame an. Das so gewonnene Modell wird gespeichert (”modelName”).
TrainModelWithParameters	Trainiert die Pipeline und setzt für diese Ausführung zusätzliche Parameter. ”parameters” muss folgende Form haben: ”%stage%.%Param%=%variable%”

Anhang B

XMLs zum Kapitel „Modellqualität“

B.1 Experiment 14.1: Vergleich von Klassifikationsmodellen

```
1  <container>
2  
3      <input id="1" class="stream.io.CSVInput" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/datasets/
            gammahadron-balanced/" />
4  
5      <task id="2" input="1">
6  
7          <pipeline modelName="gammaHadronModel" automaticTraining="false">
8              <transformer stage="VectorAssembler" inputCols="numPixelInShower,Size,M3Long,M3Trans,M4Long,M4Trans
                    ,COGx,COGy,Length,Width,Delta,m3l,m3t,numIslands,Concentration_onePixel,Concentration_twoPixel
                    ,ConcCore,concCOG,Leakage,Leakage2,Timespread,Timespread_weighted,Slope_long,Slope_trans,
                    Slope_spread,Slope_spread_weighted,Disp,Alpha,Distance,CosDeltaAlpha,Theta,Energy" outputCol="
                    features" />
9              <estimator stage="StringIndexer" inputCol="IsGamma" outputCol="IsGammaIndexed" />
10 
11             <estimator stage="RandomForestClassifier" cacheNodeIds="true"  featuresCol="features" labelCol="
                    IsGammaIndexed" rawPredictionCol="raw1" probabilityCol="prob1" predictionCol="prediction1" />
12             <estimator stage="GBTClassifier" cacheNodeIds="true" featuresCol="features" labelCol="
                    IsGammaIndexed" predictionCol="prediction2" /> <!-- no rawPredictionCol, no probabilityCol -->
13             <estimator stage="MultilayerPerceptronClassifier" featuresCol="features" labelCol="IsGammaIndexed"
                     predictionCol="prediction3" />
14 
15         </pipeline>
16 
17         <stream.pg594.operators.GenerateLayersMPC stage="4" numFeatures="32" numLabels="2" numInnerLayers="20"
                />
18         <stream.pg594.operators.SaveDataFrame name="original" />
19 
20         <stream.pg594.operators.ForEach header="i in [1,2,3,4,5]">
21             <stream.pg594.operators.LoadDataFrame name="original" />
22             <stream.pg594.operators.SplitDataFrame ratio="0.1" newName="testData"/>
23             <stream.pg594.operators.SaveDataFrame name="trainingData" />
24             <stream.pg594.operators.CountRowsWhere name="trainingEvents" />
25             <stream.pg594.operators.TrainModel modelName="gammaHadronModel" />
26             <stream.pg594.operators.LoadDataFrame name="testData" />
27             <stream.pg594.operators.ApplyModel modelName="gammaHadronModel" />
28 
29 
30             <!-- Evaluate every forest and append result to csv -->
31             <stream.pg594.operators.IndexToStringConversion modelName="gammaHadronModel" inputCol="prediction1"
                     outputCol="IsGamma1"/>
32             <stream.pg594.operators.EvaluateBinaryClassifierRaw labelColIndexed="IsGammaIndexed"
                    rawPredictionCol="raw1"/>
33             <stream.pg594.operators.EvaluateBinaryClassifier labelCol="IsGamma" predictionCol="IsGamma1" />
```

```
34            <stream.pg594.operators.AppendToCSV columns="type=RF,trainingEvents,AreaUnderROC,AreaUnderPrecisionRecallCurve,TP,TN,FP,FN,TPrate,TNrate,FPrate,FNrate" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/classificationexperiment.csv"/>
35
36            <stream.pg594.operators.IndexToStringConversion modelName="gammaHadronModel" inputCol="prediction2" outputCol="IsGamma2"/>
37            <!-- GBT doesn't have rawPrediction <stream.pg594.operators.EvaluateBinaryClassifierRaw labelColIndexed="IsGammaIndexed" rawPredictionCol="raw2"/> -->
38            <stream.pg594.operators.EvaluateBinaryClassifier labelCol="IsGamma" predictionCol="IsGamma2" />
39            <stream.pg594.operators.AppendToCSV columns="type=GBT,trainingEvents,AreaUnderROC=0,AreaUnderPrecisionRecallCurve=0,TP,TN,FP,FN,TPrate,TNrate,FPrate,FNrate" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/classificationexperiment.csv"/>
40
41            <stream.pg594.operators.IndexToStringConversion modelName="gammaHadronModel" inputCol="prediction3" outputCol="IsGamma3"/>
42            <!-- MPC doesn't have rawPrediction ... -->
43            <stream.pg594.operators.EvaluateBinaryClassifier labelCol="IsGamma" predictionCol="IsGamma3" />
44            <stream.pg594.operators.AppendToCSV columns="type=MPC,trainingEvents,AreaUnderROC=0,AreaUnderPrecisionRecallCurve=0,TP,TN,FP,FN,TPrate,TNrate,FPrate,FNrate" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/classificationexperiment.csv"/>
45        </stream.pg594.operators.ForEach>
46    </task>
47
48 </container>
```

Die von den einzelnen Modellen verwendeten Parameter können der folgenden Übersicht entnommen werden:

```
1  Pipeline parameters:
2  stages: stages of the pipeline (current: [Lorg.apache.spark.ml.PipelineStage;@778eb85f)
3
4  stage 0: class org.apache.spark.ml.feature.VectorAssembler
5  inputCols: input column names (current: [Ljava.lang.String;@6ac94488)
6  outputCol: output column name (default: vecAssembler_3b2ea5eb1698__output, current: features)
7
8  stage 1: class org.apache.spark.ml.feature.StringIndexer
9  handleInvalid: how to handle invalid entries. Options are skip (which will filter out rows with bad values), or
       error (which will throw an errror). More options may be added later. (default: error)
10 inputCol: input column name (current: IsGamma)
11 outputCol: output column name (default: strIdx_af29aa3fde7a__output, current: IsGammaIndexed)
12
13 stage 2: class org.apache.spark.ml.classification.RandomForestClassifier
14 cacheNodeIds: If false, the algorithm will pass trees to executors to match instances with nodes. If true, the
      algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. (default:
      false, current: true)
15 checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache
      will get checkpointed every 10 iterations (default: 10)
16 featureSubsetStrategy: The number of features to consider for splits at each tree node. Supported options: auto
      , all, onethird, sqrt, log2 (default: auto)
17 featuresCol: features column name (default: features, current: features)
18 impurity: Criterion used for information gain calculation (case-insensitive). Supported options: entropy, gini
      (default: gini)
19 labelCol: label column name (default: label, current: IsGammaIndexed)
20 maxBins: Max number of bins for discretizing continuous features.  Must be >=2 and >= number of categories for
      any categorical feature. (default: 32)
21 maxDepth: Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf node; depth 1 means 1 internal node + 2
      leaf nodes. (default: 5)
22 maxMemoryInMB: Maximum memory in MB allocated to histogram aggregation. (default: 256)
23 minInfoGain: Minimum information gain for a split to be considered at a tree node. (default: 0.0)
24 minInstancesPerNode: Minimum number of instances each child must have after split.  If a split causes the left
      or right child to have fewer than minInstancesPerNode, the split will be discarded as invalid. Should be
      >= 1. (default: 1)
25 numTrees: Number of trees to train (>= 1) (default: 20)
26 predictionCol: prediction column name (default: prediction, current: prediction1)
27 probabilityCol: Column name for predicted class conditional probabilities. Note: Not all models output well-
      calibrated probability estimates! These probabilities should be treated as confidences, not precise
      probabilities (default: probability, current: prob1)
28 rawPredictionCol: raw prediction (a.k.a. confidence) column name (default: rawPrediction, current: raw1)
29 seed: random seed (default: 207336481, current: -2221372633937965731)
30 subsamplingRate: Fraction of the training data used for learning each decision tree, in range (0, 1]. (default:
       1.0)
31 thresholds: Thresholds in multi-class classification to adjust the probability of predicting each class. Array
      must have length equal to the number of classes, with values >= 0. The class with largest value p/t is
      predicted, where p is the original probability of that class and t is the class' threshold. (undefined)
32
```

```
33 stage 3: class org.apache.spark.ml.classification.GBTClassifier
34 cacheNodeIds: If false, the algorithm will pass trees to executors to match instances with nodes. If true, the
       algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. (default:
       false, current: true)
35 checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache
       will get checkpointed every 10 iterations (default: 10)
36 featuresCol: features column name (default: features, current: features)
37 impurity: Criterion used for information gain calculation (case-insensitive). Supported options: entropy, gini
       (default: gini)
38 labelCol: label column name (default: label, current: IsGammaIndexed)
39 lossType: Loss function which GBT tries to minimize (case-insensitive). Supported options: logistic (default:
       logistic)
40 maxBins: Max number of bins for discretizing continuous features.  Must be >=2 and >= number of categories for
       any categorical feature. (default: 32)
41 maxDepth: Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf node; depth 1 means 1 internal node + 2
       leaf nodes. (default: 5)
42 maxIter: maximum number of iterations (>= 0) (default: 20)
43 maxMemoryInMB: Maximum memory in MB allocated to histogram aggregation. (default: 256)
44 minInfoGain: Minimum information gain for a split to be considered at a tree node. (default: 0.0)
45 minInstancesPerNode: Minimum number of instances each child must have after split.  If a split causes the left
       or right child to have fewer than minInstancesPerNode, the split will be discarded as invalid. Should be
       >= 1. (default: 1)
46 predictionCol: prediction column name (default: prediction, current: prediction2)
47 seed: random seed (default: -1287390502, current: 6076594278666838564)
48 stepSize: Step size to be used for each iteration of optimization. (default: 0.1)
49 subsamplingRate: Fraction of the training data used for learning each decision tree, in range (0, 1]. (default:
       1.0)
50
51 stage 4: class org.apache.spark.ml.classification.MultilayerPerceptronClassifier
52 blockSize: Block size for stacking input data in matrices. Data is stacked within partitions. If block size is
       more than remaining data in a partition then it is adjusted to the size of this data. Recommended size is
       between 10 and 1000 (default: 128)
53 featuresCol: features column name (default: features, current: features)
54 labelCol: label column name (default: label, current: IsGammaIndexed)
55 layers: Sizes of layers from input layer to output layer E.g., Array(780, 100, 10) means 780 inputs, one hidden
       layer with 100 neurons and output layer of 10 neurons. (default: [I@7b50010, current: [I@1666fc8e)
56 maxIter: maximum number of iterations (>= 0) (default: 100)
57 predictionCol: prediction column name (default: prediction, current: prediction3)
58 seed: random seed (default: -763139545, current: 2387178402411159171)
59 tol: the convergence tolerance for iterative algorithms (default: 1.0E-4)
```

B.2 Experiment 14.4: Parameterstudie zur Waldgröße von Random Forests

```
1  <container>
2      <input id="1" class="stream.io.CSVInput" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/datasets/
           gammahadron-balanced/"/>
3
4      <task id="2" input="1">
5
6          <pipeline modelName="gammaHadronModel" automaticTraining="false">
7              <transformer stage="VectorAssembler" inputCols="numPixelInShower,Size,M3Long,M3Trans,M4Long,M4Trans
                   ,COGx,COGy,Length,Width,Delta,m3l,m3t,numIslands,Concentration_onePixel,Concentration_twoPixel
                   ,ConcCore,concCOG,Leakage,Leakage2,Timespread,Timespread_weighted,Slope_long,Slope_trans,
                   Slope_spread,Slope_spread_weighted,Disp,Alpha,Distance,CosDeltaAlpha,Theta,Energy" outputCol="
                   features"/>
8              <estimator stage="StringIndexer" inputCol="IsGamma" outputCol="IsGammaIndexed" />
9              <estimator stage="RandomForestClassifier" maxDepth="25" featuresCol="features" labelCol="
                   IsGammaIndexed" cacheNodeIds="true"/>
10         </pipeline>
11
12         <stream.pg594.operators.SaveDataFrame name="original" />
13
14         <stream.pg594.operators.ForEach header="i in [1,2,3,4,5]">
15             <stream.pg594.operators.LoadDataFrame name="original" />
16             <stream.pg594.operators.SplitDataFrame ratio="0.1" newName="testData" />
17             <stream.pg594.operators.SaveDataFrame name="trainingData" />
18             <stream.pg594.operators.CountRowsWhere condition="" name="trainingEvents" />
19
20             <stream.pg594.operators.ForEach header="numTrees in [10,20,40,60,80,100]">
21                 <stream.pg594.operators.LoadDataFrame name="trainingData" />
22                 <stream.pg594.operators.StartTimeMeasurement timerName="trainingTime" />
23                 <stream.pg594.operators.TrainModelWithParameters modelName="model" parameters="2.numTrees=
                       numTrees"/>
24                 <stream.pg594.operators.StopTimeMeasurement timerName="trainingTime" />
25
26                 <stream.pg594.operators.LoadDataFrame name="testData" />
27                 <stream.pg594.operators.ApplyModel modelName="model" />
28                 <stream.pg594.operators.IndexToStringConversion modelName="model" inputCol="prediction"
                       outputCol="IsGammaPrediction"/>
29                 <stream.pg594.operators.EvaluateBinaryClassifier rawPredictionCol="rawPrediction"
                       labelColIndexed="IsGammaIndexed" predictionCol="IsGammaPrediction" labelCol="IsGamma" />
30                 <stream.pg594.operators.AppendToCSV columns="type=RandomForestClassifier,trainingEvents,
                       numTrees,maxDepth=25,trainingTime,AreaUnderROC,AreaUnderPrecisionRecallCurve,TP,TN,FP,FN,
                       TPrate,TNrate,FPrate,FNrate" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/
                       parameterstudyexperiment1.csv" />
31             </stream.pg594.operators.ForEach>
32         </stream.pg594.operators.ForEach>
33
34     </task>
35
36 </container>
```

B.3 Experiment 14.2: Vergleich von Regressionsmodellen

```
<container>

    <input id="1" class="stream.io.CSVInput" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/datasets/
        gammahadron/" />

    <task id="2" input="1">
        <pipeline modelName="energyestimator" automaticTraining="false">
            <transformer stage="VectorAssembler" inputCols="numPixelInShower,photonchargeMean,arrivalTimeMean,
                phChargeShower_mean,phChargeShower_max,phChargeShower_min,phChargeShower_kurtosis,
                phChargeShower_variance,phChargeShower_skewness,arrTimeShower_mean,arrTimeShower_max,
                arrTimeShower_min,arrTimeShower_kurtosis,arrTimeShower_variance,arrTimeShower_skewness,
                maxSlopesShower_mean,maxSlopesShower_max,maxSlopesShower_min,maxSlopesShower_kurtosis,
                maxSlopesShower_variance,maxSlopesShower_skewness,arrTimePosShower_mean,arrTimePosShower_max,
                arrTimePosShower_min,arrTimePosShower_kurtosis,arrTimePosShower_variance,
                arrTimePosShower_skewness,maxSlopesPosShower_mean,maxSlopesPosShower_max,
                maxSlopesPosShower_min,maxSlopesPosShower_kurtosis,maxSlopesPosShower_variance,
                maxSlopesPosShower_skewness,maxPosShower_mean,maxPosShower_max,maxPosShower_min,
                maxPosShower_kurtosis,maxPosShower_variance,maxPosShower_skewness,Size,M3Long,M3Trans,M4Long,
                M4Trans,COGx,COGy,Length,Width,Delta,m3l,m3t,numIslands,Concentration_onePixel,
                Concentration_twoPixel,ConcCore,concCOG,Leakage,Leakage2,Timespread,Timespread_weighted,
                Slope_long,Slope_trans,Slope_spread,Slope_spread_weighted,Disp,AzTracking,ZdTracking,
                AzPointing,ZdPointing,AzSourceCalc,ZdSourceCalc,Alpha,Distance,Alpha_Off_1,Distance_Off_1,
                Alpha_Off_2,Distance_Off_2,Alpha_Off_3,Distance_Off_3,Alpha_Off_4,Distance_Off_4,Alpha_Off_5,
                Distance_Off_5,CosDeltaAlpha,CosDeltaAlpha_Off_1,CosDeltaAlpha_Off_2,CosDeltaAlpha_Off_3,
                CosDeltaAlpha_Off_4,CosDeltaAlpha_Off_5,Theta,Theta_Off_1,Theta_Off_2,Theta_Off_3,Theta_Off_4,
                Theta_Off_5" outputCol="features" />
            <estimator stage="RandomForestRegressor" cacheNodeIds="true"  featuresCol="features" labelCol="
                Energy" />
        </pipeline>

        <stream.pg594.operators.SaveDataFrame name="original" />

        <stream.pg594.operators.ForEach header="i in [1,2,3,4,5]">
            <stream.pg594.operators.LoadDataFrame name="original" />
            <stream.pg594.operators.SplitDataFrame ratio="0.1" newName="testData"/>
            <stream.pg594.operators.SaveDataFrame name="trainingData" />
            <stream.pg594.operators.CountRowsWhere name="trainingEvents" />
            <stream.pg594.operators.StartTimeMeasurement timerName="trainingTime" />
            <stream.pg594.operators.TrainModel modelName="energyestimator" />
            <stream.pg594.operators.StopTimeMeasurement timerName="trainingTime" />
            <stream.pg594.operators.LoadDataFrame name="testData" />
            <stream.pg594.operators.CountRowsWhere name="testEvents" />
            <stream.pg594.operators.StartTimeMeasurement timerName="regressionTime" />
            <stream.pg594.operators.ApplyModel modelName="energyestimator" />

            <!-- Evaluate every forest and append result to csv -->
            <stream.pg594.operators.EvaluateRegressor labelCol="Energy" predictionCol="prediction" />
            <stream.pg594.operators.StopTimeMeasurement timerName="regressionTime" />
            <stream.pg594.operators.AppendToCSV columns="type=RF,trainingEvents,trainingTime,testEvents,
                regressionTime,rmse,r2" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/
                regressionexperiment.csv"/>
        </stream.pg594.operators.ForEach>
    </task>

</container>
```

```
...
stage 1: class org.apache.spark.ml.regression.RandomForestRegressor
cacheNodeIds: If false, the algorithm will pass trees to executors to match instances with nodes. If true, the
    algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. (default:
    false, current: true)
checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache
    will get checkpointed every 10 iterations (default: 10)
featureSubsetStrategy: The number of features to consider for splits at each tree node. Supported options: auto
    , all, onethird, sqrt, log2 (default: auto)
featuresCol: features column name (default: features, current: features)
impurity: Criterion used for information gain calculation (case-insensitive). Supported options: variance (
    default: variance)
labelCol: label column name (default: label, current: Energy)
```

```
9  maxBins: Max number of bins for discretizing continuous features.  Must be >=2 and >= number of categories for
      any categorical feature. (default: 32)
10 maxDepth: Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf node; depth 1 means 1 internal node + 2
      leaf nodes. (default: 5)
11 maxMemoryInMB: Maximum memory in MB allocated to histogram aggregation. (default: 256)
12 minInfoGain: Minimum information gain for a split to be considered at a tree node. (default: 0.0)
13 minInstancesPerNode: Minimum number of instances each child must have after split.  If a split causes the left
      or right child to have fewer than minInstancesPerNode, the split will be discarded as invalid. Should be
      >= 1. (default: 1)
14 numTrees: Number of trees to train (>= 1) (default: 20)
15 predictionCol: prediction column name (default: prediction)
16 seed: random seed (default: 235498149, current: 8821372440605673783)
17 subsamplingRate: Fraction of the training data used for learning each decision tree, in range (0, 1]. (default:
       1.0)
```

```
1  <container>
2
3      <input id="1" class="stream.io.CSVInput" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/datasets/
          gammahadron/" />
4
5      <task id="2" input="1">
6          <pipeline modelName="energyestimator" automaticTraining="false">
7              <transformer stage="VectorAssembler" inputCols="numPixelInShower,photonchargeMean,arrivalTimeMean,
                   phChargeShower_mean,phChargeShower_max,phChargeShower_min,phChargeShower_kurtosis,
                   phChargeShower_variance,phChargeShower_skewness,arrTimeShower_mean,arrTimeShower_max,
                   arrTimeShower_min,arrTimeShower_kurtosis,arrTimeShower_variance,arrTimeShower_skewness,
                   maxSlopesShower_mean,maxSlopesShower_max,maxSlopesShower_min,maxSlopesShower_kurtosis,
                   maxSlopesShower_variance,maxSlopesShower_skewness,arrTimePosShower_mean,arrTimePosShower_max,
                   arrTimePosShower_min,arrTimePosShower_kurtosis,arrTimePosShower_variance,
                   arrTimePosShower_skewness,maxSlopesPosShower_mean,maxSlopesPosShower_max,
                   maxSlopesPosShower_min,maxSlopesPosShower_kurtosis,maxSlopesPosShower_variance,
                   maxSlopesPosShower_skewness,maxPosShower_mean,maxPosShower_max,maxPosShower_min,
                   maxPosShower_kurtosis,maxPosShower_variance,maxPosShower_skewness,Size,M3Long,M4Long,COGx,COGy
                   ,Length,Width,Delta,m3l,m3t,numIslands,Concentration_onePixel,Concentration_twoPixel,ConcCore,
                   concCOG,Leakage,Leakage2,Timespread,Timespread_weighted,Slope_long,Slope_trans,Slope_spread,
                   Slope_spread_weighted,Disp,AzTracking,ZdTracking,AzPointing,ZdPointing,AzSourceCalc,
                   ZdSourceCalc,Alpha,Distance,Alpha_Off_1,Distance_Off_1,Alpha_Off_2,Distance_Off_2,Alpha_Off_3,
                   Distance_Off_3,Alpha_Off_4,Distance_Off_4,Alpha_Off_5,Distance_Off_5,CosDeltaAlpha,
                   CosDeltaAlpha_Off_1,CosDeltaAlpha_Off_2,CosDeltaAlpha_Off_3,CosDeltaAlpha_Off_4,
                   CosDeltaAlpha_Off_5,Theta,Theta_Off_1,Theta_Off_2,Theta_Off_3,Theta_Off_4,Theta_Off_5"
                   outputCol="features" />                    <!-- M3Trans, M4Trans raised exception: java.lang.
                   RuntimeException: No bin was found for continuous feature. This error can occur when given
                   invalid data values (such as NaN). Feature index: 42-->
8
9              <estimator stage="GBTRegressor" cacheNodeIds="true" featuresCol="features" labelCol="Energy" />
10         </pipeline>
11
12         <stream.pg594.operators.SaveDataFrame name="original" />
13
14         <stream.pg594.operators.ForEach header="i in [1,2,3,4,5]">
15             <stream.pg594.operators.LoadDataFrame name="original" />
16             <stream.pg594.operators.SplitDataFrame ratio="0.1" newName="testData"/>
17             <stream.pg594.operators.SaveDataFrame name="trainingData" />
18             <stream.pg594.operators.CountRowsWhere name="trainingEvents" />
19             <stream.pg594.operators.StartTimeMeasurement timerName="trainingTime" />
20             <stream.pg594.operators.TrainModel modelName="energyestimator" />
21             <stream.pg594.operators.StopTimeMeasurement timerName="trainingTime" />
22             <stream.pg594.operators.LoadDataFrame name="testData" />
23             <stream.pg594.operators.CountRowsWhere name="testEvents" />
24             <stream.pg594.operators.StartTimeMeasurement timerName="regressionTime" />
25             <stream.pg594.operators.ApplyModel modelName="energyestimator" />
26
27
28                 <!-- Evaluate every forest and append result to csv -->
29             <stream.pg594.operators.EvaluateRegressor labelCol="Energy" predictionCol="prediction" />
30             <stream.pg594.operators.StopTimeMeasurement timerName="regressionTime" />
31             <stream.pg594.operators.AppendToCSV columns="type=GBT,trainingEvents,trainingTime,testEvents,
                   regressionTime,rmse,r2" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/
                   regressionexperiment.csv"/>
32
33         </stream.pg594.operators.ForEach>
34     </task>
35
36 </container>
```

```
...
stage 1: class org.apache.spark.ml.regression.GBTRegressor
cacheNodeIds: If false, the algorithm will pass trees to executors to match instances with nodes. If true, the algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. (default: false, current: true)
checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache will get checkpointed every 10 iterations (default: 10)
featuresCol: features column name (default: features, current: features)
impurity: Criterion used for information gain calculation (case-insensitive). Supported options: variance (default: variance)
labelCol: label column name (default: label, current: Energy)
lossType: Loss function which GBT tries to minimize (case-insensitive). Supported options: squared, absolute (default: squared)
maxBins: Max number of bins for discretizing continuous features.  Must be >=2 and >= number of categories for any categorical feature. (default: 32)
maxDepth: Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf node; depth 1 means 1 internal node + 2 leaf nodes. (default: 5)
maxIter: maximum number of iterations (>= 0) (default: 20)
maxMemoryInMB: Maximum memory in MB allocated to histogram aggregation. (default: 256)
minInfoGain: Minimum information gain for a split to be considered at a tree node. (default: 0.0)
minInstancesPerNode: Minimum number of instances each child must have after split.  If a split causes the left or right child to have fewer than minInstancesPerNode, the split will be discarded as invalid. Should be >= 1. (default: 1)
predictionCol: prediction column name (default: prediction)
seed: random seed (default: -131597770, current: 9103996764841619817)
stepSize: Step size to be used for each iteration of optimization. (default: 0.1)
subsamplingRate: Fraction of the training data used for learning each decision tree, in range (0, 1]. (default: 1.0)
```

B.4 Experiment 14.3: Trainingszeit in Abhängigkeit der Clusterressourcen

```
<container>

    <property name="executors" value="8" />
    <property name="cores"     value="8" />

    <input id="1" class="stream.io.CSVInput" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/datasets/gammahadron/" />

    <task id="2" input="1">

        <pipeline modelName="gammaHadronModel" automaticTraining="false">
            <transformer stage="VectorAssembler" inputCols="numPixelInShower,Size,M3Long,M3Trans,M4Long,M4Trans,COGx,COGy,Length,Width,Delta,m3l,m3t,numIslands,Concentration_onePixel,Concentration_twoPixel,ConcCore,concCOG,Leakage,Leakage2,Timespread,Timespread_weighted,Slope_long,Slope_trans,Slope_spread,Slope_spread_weighted,Disp,Alpha,Distance,CosDeltaAlpha,Theta,Energy" outputCol="features" />
            <estimator stage="StringIndexer" inputCol="IsGamma" outputCol="IsGammaIndexed" />
            <estimator stage="RandomForestClassifier" cacheNodeIds="true" maxDepth="25" numTrees="80" featuresCol="features" labelCol="IsGammaIndexed" predictionCol="prediction" />
        </pipeline>


        <stream.pg594.operators.SplitDataFrame ratio="0.1" newName="testData"/>
        <stream.pg594.operators.SaveDataFrame name="trainingData" />
        <stream.pg594.operators.CountRowsWhere name="trainingEvents" />
        <stream.pg594.operators.StartTimeMeasurement timerName="trainingTime" />
        <stream.pg594.operators.TrainModel modelName="gammaHadronModel" />
        <stream.pg594.operators.StopTimeMeasurement timerName="trainingTime" />
        <stream.pg594.operators.LoadDataFrame name="testData" />
        <stream.pg594.operators.CountRowsWhere name="testEvents" />
        <stream.pg594.operators.StartTimeMeasurement timerName="classificationTime" />
        <stream.pg594.operators.ApplyModel modelName="gammaHadronModel" />
        <stream.pg594.operators.CountRowsWhere condition="prediction > 0.5" name="counter" />
        <stream.pg594.operators.StopTimeMeasurement timerName="classificationTime" />

        <stream.pg594.operators.AppendToCSV columns="type=RF,executors=${executors},cores=${cores},trainingEvents,trainingTime,testEvents,classificationTime" url="hdfs://s876cn01.cs.uni-dortmund.de:8020/pg594/distributionexperiment.csv"/>
    </task>

</container>
```

Anhang C

Liste der referenzierten Software

AngularJS	https://angularjs.org/
Apache Cassandra	https://cassandra.apache.org/
Apache Hadoop	https://hadoop.apache.org/
Apache Spark	https://spark.apache.org/
Apache Storm	https://storm.apache.org/
Apache YARN	https://hadoop.apache.org/docs/r2.6.2/hadoop-yarn/hadoop-yarn-site/YARN.html
Atlassian JIRA	https://de.atlassian.com/software/jira
Docker	https://www.docker.com/
Elasticsearch	https://www.elastic.co/de/products/elasticsearch
FACT Tools	https://sfb876.de/fact-tools/
MongoDB	https://www.mongodb.org/
MongoDB Docker-Image	https://hub.docker.com/_/mongo/
MVEL	https://github.com/mvel/mvel
OpenAPI Initiative	https://openapis.org/
OpenAPI Spezifikationen	https://github.com/OAI/OpenAPI-Specification
PostgreSQL	http://www.postgresql.org/
Postgres-XL	http://www.postgres-xl.org/
QueryDSL	https://github.com/querydsl/querydsl
RapidMiner	https://rapidminer.com/
Spring Boot	http://projects.spring.io/spring-boot/
streams Framework	https://sfb876.de/streams/
Swagger Editor	http://swagger.io/swagger-editor
Swagger Projekt	http://swagger.io
Swagger Tools	http://swagger.io/open-source-integrations
Swagger UI	http://swagger.io/swagger-ui

Abkrzungsverzeichnis

API Application Programming Interface

CRUD Create, Read, Update and Delete

DAG Directed Acyclic Graph

DoD Definition of Done

FACT First G-APD Cherenkov Telescope

FITS Flexible Image Transport System

GUI Graphical User Interface

HDFS Hadoop Distributed File System

HTTP Hyper Text Transfer Protocol

IBA Index of balanced accuracy

IBL Impediment Backlog

JSON JavaScript Object Notation

NASA National Aeronautics and Space Administration

NoSQL Not only SQL

NPM Node Package Manager

PBL Product Backlog

PG Projektgruppe

PO Product Owner

REST Representational State Transfer

ROC Receiver Operating Characteristic

SBL Sprint Backlog

SM Scrum Master

URL Uniform Resource Locator

WiP Work In Progress

XML Extensible Markup Language

YARN Yet Another Resource Negotiator

Abbildungsverzeichnis

1.1 Visuelle Darstellung eines Gamma-Showers (oben links), welcher von Teleskopen aufgezeichnet wird (unten links) und in Grafiken der einzelnen Aufnahmen dargestellt werden kann (rechts) [17] . . . 3
1.2 Beispielhafte Verwaltung mit TORQUE und FhGFS (jetzt BeeGFS) . . . 4
1.3 Code-2-Data mit Hadoop und Spark . . . 5
1.4 Analysekette . . . 10

2.1 Veranschaulichung der ersten vier *Vs* von Big Data. Von links nach rechts: Volume, Velocity, Variety und Veracity [91] . . . 14
2.2 Arten der Skalierung . . . 15

3.1 Lambda-Architektur . . . 18
3.2 Architektur des Apache Hadoop Projekts [73] . . . 20
3.3 Funktionsweise eines HDFS Clusters . . . 21
3.4 Apache Spark Resilient Distributed Datasets . . . 24
3.5 Maschinelles Lernen mit Spark MLlib . . . 26
3.6 Pipeline-Struktur von Spark ML . . . 26
3.7 Konkretes Beispiel für eine Pipeline in Spark ML . . . 27
3.8 Beispiel einer Storm Topologie als DAG. Zu sehen sind `Spouts` (links, erste Ebene) und `Bolts` (rechts, ab zweite Ebene) [63] . . . 28
3.9 Aufbau eines Storm Clusters [63] . . . 30
3.10 Beispielhafte Trident Topologie [64] . . . 31
3.11 Figure 3.10 als kompilierte Storm Topologie [64] . . . 32
3.12 Apache Spark Streaming . . . 32
3.13 Spark Streaming - DStream . . . 33

3.14 Schematischer Aufbau eines *Container* [15] 34
3.15 Funktionsweise eines *Stream* [15] 35
3.16 Arbeitsschritte eines *Process* [15] 35
3.17 Veranschaulichung des Gossip Protocol [2] 38
3.18 Ein typisches Datenbankschema nach dimensionaler Modellierung, hier am Beispiel einer Vertriebsdatenbank [53] 41

4.1 Beispielhafter Entscheidungsbaum 48
4.2 Unterscheidung Realer Drift vs. Virtueller Drift [37] 61
4.3 Schematische Darstellung vom unterschiedlichen Auftreten von Concept Drift [37] 62
4.4 Schematischer Aufbau einer Wahrheitsmatrix 63
4.5 Eine ROC Kurve [36] 64
4.6 Korrelation als Heuristik 71
4.7 Beispiel-Ausführung CFS [80] 72
4.8 Berechnung von Ensemble-Korrelationen in Fast-Ensembles 74
4.9 Beispiel-Ausführung Fast-Ensembles [80] 74
4.10 k-fache Kreuzvalidierung [68] 76
4.11 Active learning als Kreislauf [81] 79

5.1 Überblick über die verwendeten Software-Komponenten 84

6.1 Event vor (links) und nach (rechts) der DRS Kalibrierung. Die Spitzen entsprechen den Signalen einer einzelnen Fotodiode [5] 89
6.2 Statistik zur Luftfeuchtigkeit in der Nacht des 21.09.2013 aufgenommen von zwei Sensoren: TNG (oben) und MAGIC (unten) 90

8.1 Die Rückgabeformate der REST API 96

9.1 Datenfluss bei verteilten Batch-Prozessen 121
9.2 Datenfluss nach dem Konzept von Spark Streaming 124
9.3 Datenfluss von unserer Streaming-Lösung 125

10.1 Klassendiagramm mit zugehörigen Klassen für `DataFrameInput` und `DataFrameStream` 136

10.2 Übersicht der Klassen zuständig für die Erstellung von Pipelines und Stages 138
10.3 Übersicht der Klassen zuständig für die Verarbeitung von Pipelines und Stages 139

12.1 Der Sprint in Scrum 149
12.2 Das Kanban-Board 152

13.1 Eventraten der Feature Extraction auf MC-Daten 161
13.2 Eventraten der Feature Extraction auf Teleskop-Daten 162

14.1 Die True-Positive-Rate, True-Negative-Rate und die Präzision der verschiedenen Klassifikationsverfahren. Eingezeichnet ist der Mittelwert aus 5 Replikationen und die einfache Standardabweichung. Die Werte für den MPC sind in den beiden letzten Grafiken erheblich schlechter und werden daher nicht aufgeführt. 167
14.2 Der root mean squared error und der R^2 Wert für die Regressionsverfahren. Eingezeichnet ist der Mittelwert aus 5 Replikationen und die einfache Standardabweichung. 169
14.3 Trainingsdurchsatz und Klassifikationsdurchsatz in Abhängigkeit von verwendeten Rechenknoten und Prozessoren. 171
14.4 Durchsatz in Abhängigkeit der Größe des RandomForest. Eingezeichnet ist der Mittelwert aus 5 Replikationen und die einfache Standardabweichung. . 172
14.5 Die erzielte True-Positive-Rate, True-Negative-Rate und die Präzision in Abhängigkeit der Größe des RandomForest. Eingezeichnet ist der Mittelwert aus 5 Replikationen und die einfache Standardabweichung. 173

18.1 Konfigurationsseite der Web-UI 190
18.2 Interface zum Managen von Jobs 190
18.3 Konfiguration eines Jobs 191
18.4 Auflistung aller gestarteten Jobs 192
18.5 Übersichtseite der Tasks 192
18.6 Übersichtseite des Event 193
18.7 Übersichtseite des Event Ergebnis 194
18.8 Konfigurationsseite der Web-UI 194

Literaturverzeichnis

[1] Adaptive Computing: *TORQUE*. http://www.adaptivecomputing.com/products/open-source/torque/ (13.08.2016).

[2] Alberto Diegeo Prieto Löfkrantz: *Do you know Cassandra?*. http://blogs.atlassian.com/2013/09/do-you-know-cassandra/ (19.10.2016).

[3] Anderhub, H., M. Backes, A. Biland, V. Boccone, I. Braun, T. Bretz, J. Buss, F. Cadoux, V. Commichau, L. Djambazov, D. Dorner, S. Einecke, D. Eisenacher, A. Gendotti, O. Grimm, H. von Gunten, C. Haller, D. Hildebrand, U. Horisberger, B. Huber, K. S. Kim, M. L. Knoetig, J. H. Köhne, T. Krähenbühl, B. Krumm, M. Lee, E. Lorenz, W. Lustermann, E. Lyard, K. Mannheim, M. Meharga, K. Meier, T. Montaruli, D. Neise, F. Nessi-Tedaldi, A. K. Overkemping, A. Paravac, F. Pauss, D. Renker, W. Rhode, M. Ribordy, U. Röser, J. P. Stucki, J. Schneider, T. Steinbring, F. Temme, J. Thaele, S. Tobler, G. Viertel, P. Vogler, R. Walter, K. Warda, Q. Weitzel und M. Zänglein: *Design and operation of FACT – the first G-APD Cherenkov telescope*. Journal of Instrumentation, 8(06):P06008, 2013.

[4] Anderhub, H., M. Backes, A. Biland, A. Boller, I. Braun, T. Bretz, V. Commichau, L. D. Jambazov, D. Dorner, C. Farnier, A. Gendotti, O. Grimm, H. von Gunten, D. Hildebrand, U. Horisberger, B. Huber, K.-S. Kim, J.-H. Köhne, T. Krähenbühl, B. Krumm, M. Lee, J.-P. Lenain, E. Lorenz, W. Lustermann, E. Lyard, K. Mannheim, M. Meharga, D. Neise, F. Nessi-Tedaldi, A.-K. Overkemping, F. Pauss, D. Renker, W. Rhode, M. Ribordy, R. Rohlfs, U. Röser, J.-P. Stucki, J. Schneider, J. Thaele, O. Tibolla, G. Viertel, P. Vogler, R. Walter, K. Warda und Q. Weitzel: *Status of the First G-APD Cherenkov Telescope (FACT)*. In: *Proceedings of the 32nd International Cosmic Ray Conference (ICRC2011)*, Bd. 9, S. 203–206, Peking, China, 2011.

[5] Anderhub, H., A. Biland, I. Braun, S. Commichau, V. Commichau, O. Grimm, H. v. Gunten, D. M. Hildebrand, U. Horisberger,

T. Krähenbühl, W. Lustermann, F. Pauss et al.: *Calibrating the camera for the First G-APD Cherenkov Telescope (FACT).* In: *Proceedings of the 32nd International Cosmic Ray Conference (ICRC2011)*, Bd. 9, S. 30–33, Peking, China, 2011.

[6] Apache Software Foundation: *Apache Spark Machine Learning Library Guide*, 2016. `https://spark.apache.org/docs/1.6.1/mllib-guide.html` (08.07.2016).

[7] Apache Software Foundation: *Overview.* Apache Spark 1.6.1 Java API, 2016. `http://spark.apache.org/docs/1.6.1/api/java/index.html` (08.10.2016).

[8] Apache Software Foundation: *RandomForestClassifier*. Apache Spark 1.6.1 Java API, 2016. `http://spark.apache.org/docs/1.6.1/api/java/org/apache/spark/ml/classification/RandomForestClassifier.html` (08.10.2016).

[9] Bandyopadhyay, S., C. Giannella, U. Maulik, H. Kargupta, K. Liu und S. Datta: *Clustering distributed data streams in peer-to-peer environments*. Information Sciences, 176(14):1952–1985, 2006.

[10] Beck, K. et al.: *Manifesto for Agile Software Development*, 2001. `http://www.agilemanifesto.org/` (08.02.2016).

[11] Berger, K., T. Bretz, D. Dorner, D. Hoehne und B. Riegel: *A robust way of estimating the energy of a gamma ray shower detected by the magic telescope*. In: *Proceedings of the 29th International Cosmic Ray Conference*, S. 100–104, 2005.

[12] Bifet, A., G. Holmes, R. Kirkby und B. Pfahringer: *Moa: Massive online analysis*. The Journal of Machine Learning Research, 11:1601–1604, 2010.

[13] Birant, D. und A. Kut: *ST-DBSCAN: An algorithm for clustering spatial–temporal data*. Data & Knowledge Engineering, 60(1):208–221, 2007.

[14] Bock, R., A. Chilingarian, M. Gaug, F. Hakl, T. Hengstebeck, M. Jiřina, J. Klaschka, E. Kotrč, P. Savickỳ, S. Towers et al.: *Methods for multidimensional event classification: a case study using images from a Cherenkov gamma-ray telescope*. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 516(2):511–528, 2004.

[15] Bockermann, C.: *The streams Framework.* `https://sfb876.de/streams/` (17.01.2016).

[16] Bockermann, C. und H. Blom: *The streams Framework*. Techn. Ber. 5, TU Dortmund, Dezember 2012.

[17] Bockermann, C., K. Brügge, J. Buss, A. Egorov, K. Morik, W. Rhode und T. Ruhe: *Online Analysis of High-Volume Data Streams in Astroparticle Physics*.

In: Bifet, A., M. May, B. Zadrozny, R. Gavalda, D. Pedreschi, F. Bonchi, J. Cardoso und M. Spiliopoulou (Hrsg.): *Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2015)*, Bd. 3, S. 100–115. Springer, 2015.

[18] Boulicaut, J.-F., K. Morik und A. Siebes (Hrsg.): *Local Pattern Detection - International Seminar Dagstuhl Castle, Germany, April 12-16, 2004, Revised Selected Papers*. Springer Berlin Heidelberg, 2005.

[19] Bradley, J. K.: *Decision Trees on Spark*, September 2014. `https://speakerdeck.com/jkbradley/mllib-decision-trees-at-sf-scala-baml-meetup` (15.08.2016).

[20] Bradley, J. K. und M. Amde: *Scalable Decision Trees in MLlib*. Databricks Blog, September 2014. `https://databricks.com/blog/2014/09/29/scalable-decision-trees-in-mllib.html` (15.08.2016).

[21] Cappellaro, E. und M. Turatto: *Supernova types and rates*. In: Vanbeveren, D. (Hrsg.): *The influence of binaries on stellar population studies*, S. 199–214. Springer, 2001.

[22] Chang, C.-C. und C.-J. Lin: *LIBSVM: A library for support vector machines*. ACM Transactions on Intelligent Systems and Technology, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[23] Corpet, F.: *Multiple sequence alignment with hierarchical clustering*. Nucleic acids research, 16(22):10881–10890, 1988.

[24] Crockford, D.: *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627, RFC Editor, Juli 2006. `http://www.rfc-editor.org/rfc/rfc4627.txt` (08.07.2016).

[25] Datta, S., C. Giannella und H. Kargupta: *K-Means Clustering Over a Large, Dynamic Network*. In: Ghosh, J., D. Lambert, D. B. Skillicorn und J. Srivastava (Hrsg.): *Proceedings of the Sixth SIAM International Conference on Data Mining, April 20-22, 2006, Bethesda, MD, USA*, S. 153–164. SIAM, 2006.

[26] Dean, J. und S. Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*. Communications of the ACM, 51(1):107–113, Januar 2008.

[27] Dietterich, T. G.: *Ensemble methods in machine learning*. In: *Multiple classifier systems*, S. 1–15. Springer, 2000.

[28] Douglas, K. und S. Douglas: *PostgreSQL: a comprehensive guide to building, programming, and administering PostgreSQL databases*. SAMS, 2003.

[29] Dries, A. und U. Rückert: *Adaptive concept drift detection*. Statistical Analysis and Data Mining, 2(5-6):311–327, 2009.

[30] Fawcett, T.: *An introduction to ROC analysis.* Pattern recognition letters, 27(8):861–874, 2006.

[31] Fielding, R. T.: *Architectural styles and the design of network-based software architectures.* Doktorarbeit, University of California, Irvine, 2000.

[32] Fielding, R. T., J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach und T. Berners-Lee: *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616, RFC Editor, Juni 1999. http://www.rfc-editor.org/rfc/rfc2616.txt (08.07.2016).

[33] FITS Working Group et al.: *Definition of the flexible image transport system (FITS), version 3.0*, 2008. http://fits.gsfc.nasa.gov/fits_standard.html (11.07.2016).

[34] Freund, Y. und R. E. Schapire: *A Short Introduction to Boosting.* Journal of Japanese Society for Artificial Intelligence, 14(5):771–780, 1999.

[35] Friedman, J., T. Hastie und R. Tibshirani: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer Series in Statistics, 2001.

[36] Galar, M., A. Fernandez, E. Barrenechea, H. Bustince und F. Herrera: *A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches.* IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, 42(4):463–484, 2012.

[37] Gama, J., I. Žliobaitė, A. Bifet, M. Pechenizkiy und A. Bouchachia: *A survey on concept drift adaptation.* ACM Computing Surveys (CSUR), 46(4):44, 2014.

[38] García, V., R. A. Mollineda und J. S. Sánchez: *Index of balanced accuracy: A performance measure for skewed class distributions.* In: Araujo, H., A. M. Mendonça, A. J. Pinho und M. I. Torres (Hrsg.): *Pattern Recognition and Image Analysis (IbPRIA 2009)*, S. 441–448. Springer, 2009.

[39] García, V., J. S. Sánchez und R. A. Mollineda: *On the effectiveness of preprocessing methods when dealing with different levels of class imbalance.* Knowledge-Based Systems, 25(1):13–21, 2012.

[40] Ghemawat, S., H. Gobioff und S.-T. Leung: *The Google File System.* In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, S. 29–43, New York, NY, USA, 2003. ACM.

[41] Guyon, I., C. Aliferis und A. Elisseeff: *Causal feature selection.* In: Liu, H. und H. Motoda (Hrsg.): *Computational methods of feature selection*, Data Mining

and Knowledge Discovery Series, S. 63–86. Chapman and Hall/CRC, Boca Raton, FL, 2007.

[42] Guyon, I. und A. Elisseeff: *An introduction to variable and feature selection*. The Journal of Machine Learning Research, 3:1157–1182, 2003.

[43] Hall, M. A.: *Correlation-based feature selection for machine learning*. Doktorarbeit, The University of Waikato, 1999.

[44] Heck, D., G. Schatz, J. Knapp, T. Thouw und J. Capdevielle: *CORSIKA: A Monte Carlo code to simulate extensive air showers*. Wissenschaftlicher Bericht FZKA 6019, Forschungszentrum Karlsruhe GmbH, Karlsruhe, 1998.

[45] Helf, M.: *Gamma-Hadron-Separation im MAGIC Experiment durch verteilungsgestütztes Sampling*. Diplomarbeit, TU Dortmund, 2011.

[46] Herrera, F., C. J. Carmona, P. González und M. J. del Jesus: *An overview on subgroup discovery: foundations and applications*. Knowledge and information systems, 29(3):495–525, 2011.

[47] InterFace AG: *Das KANBAN-Plakat*. `http://www.kanban-plakat.de/` (03.11.2015).

[48] InterFace AG: *Das SCRUM-Plakat*. `http://www.scrum-plakat.de/` (03.11.2015).

[49] Jain, A. K., M. N. Murty und P. J. Flynn: *Data Clustering: A Review*. ACM Computing Surveys (CSUR), 31(3):264–323, September 1999.

[50] Jain, R.: *How Lambda Architecture Can Analyze Big Data Batches in Near Real-Time*, Oktober 2013. `http://data-informed.com/lambda-architecture-can-analyze-big-data-batches-near-real-time/` (01.03.2016).

[51] Japkowicz, N. und S. Stephen: *The Class Imbalance Problem: A Systematic Study*. Intelligent Data Analysis, 6(5):429–449, Oktober 2002.

[52] Kargupta, H. und B. Park: *A Fourier Spectrum-Based Approach to Represent Decision Trees for Mining Data Streams in Mobile Environments*. IEEE Transactions on Knowledge and Data Engineering, 16(2):216–229, Februar 2004.

[53] Kimball, R. und M. Ross: *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[54] Klösgen, W.: *Explora: a multipattern and multistrategy discovery assistant*. In: Fayyad, U. M., G. Piatetsky-Shapiro, P. Smyth und R. Uthurusamy (Hrsg.): *Advances in Knowledge Discovery and Data Mining*, S. 249–271. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[55] KLÖSGEN, W.: *Applications and research problems of subgroup mining*. In: RAŚ, Z. W. und A. SKOWRON (Hrsg.): *Foundations of Intelligent Systems (ISMIS'99)*, Bd. 1609 d. Reihe *Lecture Notes in Computer Science*, S. 1–15. Springer Berlin Heidelberg, 1999.

[56] KOLLER, D. und M. SAHAMI: *Toward Optimal Feature Selection*. In: SAITTA, L. (Hrsg.): *Proceedings of the Thirteenth International Conference on Machine Learning (ICML'96)*, S. 284–292. Morgan Kaufmann, 1996.

[57] KSHEMKALYANI, A. D. und M. SINGHAL: *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, März 2011.

[58] LAVRAČ, N., B. KAVŠEK, P. FLACH und L. TODOROVSKI: *Subgroup discovery with CN2-SD*. The Journal of Machine Learning Research, 5:153–188, 2004.

[59] LIAW, A. und M. WIENER: *Classification and Regression by RandomForest*. R News, 2(3):18–22, Dezember 2002.

[60] LICHMAN, M.: *UCI Machine Learning Repository*. University of California, Irvine, School of Information and Computer Sciences, 2013. `http://archive.ics.uci.edu/ml` (12.07.2016).

[61] MAMPAEY, M., S. NIJSSEN, A. FEELDERS und A. KNOBBE: *Efficient algorithms for finding richer subgroup descriptions in numeric and nominal data*. In: *IEEE International Conference on Data Mining*, S. 499–508, 2012.

[62] MARR, B.: *Big Data: The 5 Vs Everyone Must Know*, 2014. `https://www.linkedin.com/pulse/20140306073407-64875646-big-data-the-5-vs-everyone-must-know` (13.03.2016).

[63] MARZ, N.: *A Storm is coming: more details and plans for release*. Twitter Blog, August 2011. `https://blog.twitter.com/2011/a-storm-is-coming-more-details-and-plans-for-release` (25.03.2016).

[64] MARZ, N.: *Trident: a high-level abstraction for realtime computation*. Twitter Blog, August 2012. `https://blog.twitter.com/2012/trident-a-high-level-abstraction-for-realtime-computation` (25.03.2016).

[65] MARZ, N. und J. WARREN: *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications, Mai 2015.

[66] MASSE, M.: *REST API design rulebook*. O'Reilly Media, 2011.

[67] MATTERN, F.: *Verteilte Basisalgorithmen*, Bd. 226 d. Reihe *Informatik-Fachberichte*. Springer, 1989.

[68] McCormick, C.: *K-Fold Cross-Validation, With MATLAB Code*, August 2013. http://mccormickml.com/2013/08/01/k-fold-cross-validation-with-matlab-code/ (08.07.2016).

[69] Meier, K. J.: *FACT - The First G-APD Cherenkov Telescope*, Mai 2014. http://www.astro.uni-wuerzburg.de/en/research/fact/fact-introduction (23.02.2016).

[70] MongoDB, Inc.: *MongoDB CRUD Operations*. https://docs.mongodb.org/manual/crud/ (25.03.2016).

[71] MongoDB Inc: *Install MongoDB Community Edition*, 2016. https://docs.mongodb.com/manual/administration/install-community/ (07.10.2016).

[72] Morik, K. und C. Weihs: *Wissensentdeckung in Datenbanken*. Folien zur gleichnamigen Vorlesung an der TU Dortmund, 2015.

[73] Murthy, A. C.: *Apache Hadoop YARN - Enabling Next Generation Data Applications*, 2013. http://de.slideshare.net/hortonworks/apache-hadoop-yarn-enabling-nex (23.03.2016).

[74] OpenAPI Initiative: *OpenAPI Specification Version 2.0*, 2014. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md (07.10.2016).

[75] Quinlan, J. R.: *Bagging, Boosting, and C4.5*. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Bd. 1, S. 725–730, 1996.

[76] Richardson, L. und S. Ruby: *RESTful web services*. O'Reilly Media, 2008.

[77] Rijsbergen, C. J. V.: *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2. Aufl., 1979.

[78] Ryza, S.: *How-to: Tune Your Apache Spark Jobs (Part 2)*. Cloudera Engineering Blog, März 2015. http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/ (06.09.2016).

[79] Saeys, Y., T. Abeel und Y. Van de Peer: *Robust feature selection using ensemble feature selection techniques*. In: Walter Daelemans, Bart Goethals, K. M. (Hrsg.): *Machine learning and knowledge discovery in databases (ECML PKDD 2008)*, Bd. 2, S. 313–325. Springer, 2008.

[80] Schowe, B. und K. Morik: *Fast-ensembles of minimum redundancy feature selection*. In: Okun, O., G. Valentini und M. Re (Hrsg.): *Ensembles in Machine Learning Applications*, S. 75–95. Springer, 2011.

[81] Settles, B.: *Active Learning Literature Survey*. Computer Sciences Technical Report 1648, University of Wisconsin-Madison, 2009.

[82] Sharma, M., J. Nayak, M. K. Koul, S. Bose und A. Mitra: *Gamma/hadron segregation for a ground based imaging atmospheric Cherenkov telescope using machine learning methods: Random Forest leads*. Research in Astronomy and Astrophysics, 14(11):1491, 2014.

[83] Taniar, D., C. H. C. Leung, J. W. Rahayu und S. Goel: *High Performance Parallel Database Processing and Grid Databases*. John Wiley & Sons, 2008.

[84] Todorovski, L., P. Flach und N. Lavrač: *Predictive performance of weighted relative accuracy*. In: Zighed, D. A., J. Komorowski und J. Żytkow (Hrsg.): *Principles of Data Mining and Knowledge Discovery (PKDD 2000)*, S. 255–264. Springer, 2000.

[85] TU Dortmund, Fakultät für Informatik: *Modulhandbuch Master-Studiengänge Informatik und Angewandte Informatik*, April 2016. http://www.cs.tu-dortmund.de/nps/de/Studium/Ordnungen_Handbuecher_Beschluesse/Modulhandbuecher/Master_Inf/gesamtes_Modulhandbuch/Modulhandbuch_MSc_INF_1.pdf (08.07.2016).

[86] Turatto, M.: *Classification of supernovae*. In: Weiler, K. W. (Hrsg.): *Supernovae and Gamma-Ray Bursters*, S. 21–36. Springer, 2003.

[87] Tyagi, U.: *Data Science & Spark :- Logistic Regression implementation for spam dataset*. knoldus Blog, Mai 2015. https://blog.knoldus.com/2015/05/25/data-science-spark-logistic-regression-implementation-for-spam-dataset/ (11.07.2016).

[88] Unyelioglu, K.: *Using Artificial Neural Networks to Predict Emergency Department Deaths*. DZone, Mai 2016. https://dzone.com/articles/apache-spark-machine-learning-using-artificial-neu (18.08.2016).

[89] Vavilapalli, V. K., A. C. 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 und E. Baldeschwieler: *Apache Hadoop YARN: Yet Another Resource Negotiator*. In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC'13)*, S. 5:1–5:16, New York, NY, USA, 2013. ACM.

[90] Wagstaff, K., C. Cardie, S. Rogers, S. Schrödl et al.: *Constrained k-means clustering with background knowledge*. In: *Proceedings of the Eighteenth International Conference on Machine Learning (ICML '01)*, S. 577–584. Morgan Kaufmann, 2001.

[91] Walker, M.: *Data Veracity*, 2012. http://www.datasciencecentral.com/profiles/blogs/data-veracity (13.03.2016).

[92] WAMPLER, D.: *Apache Spark Resilient Distributed Datasets.* `http://www.lightbend.com/activator/template/spark-workshop` (01.03.2016).

[93] WIKIPEDIA: *Scalability*, 2016. `https://en.wikipedia.org/wiki/Scalability` (22.02.2016).

[94] XIN, R. S., J. ROSEN, M. ZAHARIA, M. J. FRANKLIN, S. SHENKER und I. STOICA: *Shark: SQL and rich analytics at scale.* In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, S. 13–24. ACM, 2013.

[95] ZHANG, W.: *Complete Anytime Beam Search.* In: *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence (AAAI '98/IAAI '98)*, S. 425–430, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[96] ZHOU, Z.: *Ensemble Methods: Foundations and Algorithms*. Chapman and Hall/CRC, 2012.